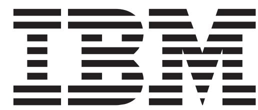
IBM WebSphere Application Server for z/OS, Version 8.5

Securing WebSphere applications



SA32-1082-00

Note

Before using this information, be sure to read the general information under “Notices” on page 1047.

Compilation date: June 1, 2012

© Copyright IBM Corporation 2012.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

How to send your comments . . . vii

Using this PDF . . . ix

Chapter 1. Securing Client applications . . . 1
Configuring secure access to resources for applet clients . . . 1
  Applet client security requirements . . . 1
Example: Running the thin or pluggable application client with security enabled . . . 2
Configuring secure access for stand-alone clients . . . 3

Chapter 2. Securing Data access resources . . . 5
Securing data sources . . . 5
  Java EE connector security . . . 5
  Enabling trusted context for DB2 databases . . . 9
  Configuring the application server and DB2 to authenticate with Kerberos . . . 15
  Connection thread identity . . . 17
  Using thread identity support . . . 19
Securing optimized local adapters . . . 21
  Security considerations using optimized local adapters with IMS . . . 21
  Securing optimized local adapters for inbound support . . . 22
  Securing optimized local adapters for outbound support . . . 23

Chapter 3. Securing EJB applications . . . 27
Securing Enterprise JavaBeans applications . . . 27
  Securing enterprise bean applications . . . 27

Chapter 4. Securing Messaging resources . . . 31
Securing messaging . . . 31
  Configuring security for message-driven beans that use activation specifications . . . 32
  Configuring security for message-driven beans that use listener ports . . . 33

Chapter 5. Securing Mail, URLs, and other Java EE resources . . . 35
Securing applications that use the JavaMail API . . . 35
  JavaMail API security permissions best practices . . . 35

Chapter 6. Securing OSGi applications . . . 37

Chapter 7. Securing Portlet applications . . . 39
Portlet URL security . . . 39
  Portlet URL security . . . 39

Chapter 8. Securing Service integration . . . 43
Securing service integration. . . . 43
  Securing buses . . . 44
  Disabling bus security . . . 53
  Enabling client SSL authentication . . . 53
  Adding unique names to the bus authorization policy . . . 55
  Administering authorization permissions . . . 56
  Administering permitted transports for a bus . . . 82
  Securing messages between messaging buses . . . 85
  Securing access to a foreign bus. . . . 86
  Securing links between messaging engines . . . 86
  Controlling which foreign buses can link to your bus. . . . 87

© Copyright IBM Corp. 2012

Securing database access 87
Securing mediations 88
Auditing the service integration security infrastructure 90

Chapter 9. Securing Session Initiation Protocol (SIP) applications 93
Securing SIP applications 93
Configuring security for the SIP container 93
Configuring digest authentication for SIP 94
Developing a custom trust association interceptor 97

Chapter 10. Securing web applications 101
Web application security components and settings 101
Web component security 101
Securing web applications using an assembly tool 101
Security constraints in web applications 104
Security settings 105
Assigning users and groups to roles 106
Securing applications during assembly and deployment 118
Session security support 121

Chapter 11. Securing web services 123
Securing JAX-RS web applications 123
Securing JAX-RS applications within the web container 123
Securing JAX-RS resources using annotations 129
Securing downstream JAX-RS resources 133
Securing JAX-RS clients using SSL 134
Administering secure JAX-RS applications 137
Defining and managing secure policy set bindings 138
Configuring the SSL transport policy 138
Configuring SCA web service binding for transport layer authentication 140
Transformation of policy and binding assertions for WSDL 141
Securing message parts using the administrative console 143
Signing and encrypting message parts using policy sets 145
Configuring the callers for general and default bindings 150
Changing the order of the callers for a token or message part 150
Configuring SCA web service binding to use SSL 151
Configuring web service binding for LTPA authentication 152
Policy set bindings settings for WS-Security 153
Keys and certificates 155
WS-Security authentication and protection 163
Caller settings 183
Caller collection 186
Message expiration settings 187
Actor roles settings 188
Securing web services 188
Securing web services applications at the transport level 188
Authenticating web services clients using HTTP basic authentication 190
Securing JAX-WS web services using message-level security 191
Securing JAX-RPC web services using message-level security 192
Securing web services using Security Markup Assertion Language (SAML) 193
Authenticating web services using generic security token login modules 193
Web Services Security concepts 194
Migrating Web Services Security 362
Developing applications that use Web Services Security 378
Configuring Web Services Security during application assembly 600
Administering Web Services Security 671

Deploying applications that use SAML . . . 1014
Tuning Web Services Security . . . 1021
Securing WSIF . . . 1023
Configuring UDDI registry security . . . 1024
Configuring the UDDI registry to use WebSphere Application Server security. . . . 1024
Configuring UDDI security with WebSphere Application Server security enabled . . . 1026
Configuring UDDI Security with WebSphere Application Server security disabled . . . 1027
Access control for UDDI registry interfaces . . . 1027
UDDI registry security and UDDI registry settings . . . 1028
Securing bus-enabled web services . . . 1030
Overriding the default security configuration between bus-enabled web services and a secure bus 1031
Configuring secure transmission of SOAP messages by using WS-Security . . . 1034
Working with password-protected components . . . 1035
Invoking outbound services over HTTPS . . . 1043
Securing WS-Notification . . . 1044
Configuring secure access to WS-Notification service points by using SOAP over HTTPS . . . 1045

Notices . . . 1047

Trademarks and service marks . . . 1049

Index . . . 1051

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

- To send comments on articles in the WebSphere Application Server Information Center
  1. Display the article in your Web browser and scroll to the end of the article.
  2. Click on the **Feedback** link at the bottom of the article, and a separate window containing an email form appears.
  3. Fill out the email form as instructed, and submit your feedback.
- To send comments on PDF books, you can email your comments to: **wasdoc@us.ibm.com**.

  Your comment should pertain to specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. Be sure to include the document name and number, the WebSphere Application Server version you are using, and, if applicable, the specific page, table, or figure number on which you are commenting.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer. When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about your comments.

© Copyright IBM Corp. 2012

Using this PDF

Links

Because the content within this PDF is designed for an online information center deliverable, you might experience broken links. You can expect the following link behavior within this PDF:

- Links to Web addresses beginning with http:// work.
- Links that refer to specific page numbers within the same PDF book work.
- The remaining links will *not* work. You receive an error message when you click them.

Print sections directly from the information center navigation

PDF books are provided as a convenience format for easy printing, reading, and offline use. The information center is the official delivery format for IBM WebSphere Application Server documentation. If you use the PDF books primarily for convenient printing, it is now easier to print various parts of the information center as needed, quickly and directly from the information center navigation tree.

To print a section of the information center navigation:

1. Hover your cursor over an entry in the information center navigation until the **Open Quick Menu** icon is displayed beside the entry.
2. Right-click the icon to display a menu for printing or searching your selected section of the navigation tree.
3. If you select **Print this topic and subtopics** from the menu, the selected section is launched in a separate browser window as one HTML file. The HTML file includes each of the topics in the section, with a table of contents at the top.
4. Print the HTML file.

For performance reasons, the number of topics you can print at one time is limited. You are notified if your selection contains too many topics. If the current limit is too restrictive, use the feedback link to suggest a preferable limit. The feedback link is available at the end of most information center pages.

© Copyright IBM Corp. 2012

Chapter 1. Securing Client applications

This page provides a starting point for finding information about application clients and client applications. Application clients provide a framework on which application code runs, so that your client applications can access information on the application server.

For example, an insurance company can use application clients to help offload work on the server and to perform specific tasks. Suppose an insurance agent wants to access and compile daily reports. The reports are based on insurance rates that are located on the server. The agent can use application clients to access the application server where the insurance rates are located.

Configuring secure access to resources for applet clients

By default, the applet client is configured to have security enabled. If you have administrative security turned on at the server from which you are accessing resources, then you can use secure sockets layer (SSL) when needed.

About this task

If you decide that the security requirements for applet client applications differ from other types of client applications, then create a new version of the `sas.client.props` and `ssl.client.props` files.

Procedure

1. Make a copy of the following files so that you can use them for an applet:
   - `<app_client_root>\properties\sas.client.props`
   - `<app_client_root>\properties\ssl.client.props`
2. Edit the copies of the `sas.client.props` and `ssl.client.props` files that you made with your changes.
3. Display the IBM® Control Panel for Java. Click **Start** > **Control panel**, then select the product Java Plug-In.
4. To use the files you created in step 1, modify the following Java Run-Time parameter values. Click the Advanced tab, then edit the parameters in the Java Runtime Parameters field:
   - `-Dcom.ibm.CORBA.ConfigURL=file:<app_client_root>\properties\sas.client.props`
   - `-Dcom.ibm.SSL.ConfigURL=file:<app_client_root>\properties\ssl.client.props`
5. To save your changes, click **Apply**

Applet client security requirements

When code is loaded, it is assigned permissions based on the security policy in effect. This policy specifies the permissions that are available for code from various locations. You can initialize this policy from an external policy file.

By default, the client uses the *app_server_root*/`properties/client.policy` file. You must update this file with the following permission:

SocketPermission grants permission to open a port and make a connection to a host machine, which is your WebSphere® Application Server. In the following example, `yourserver.yourcompany.com` is the complete host name of your WebSphere Application Server:

```
permission java.util.PropertyPermission "*", "read";
permission java.net.SocketPermission "yourserver.yourcompany.com ,"connect";
```

© IBM Corporation 2009

Example: Running the thin or pluggable application client with security enabled

Your Java thin application client no longer needs additional code to set security providers if you have enabled security for your WebSphere Application Server instance. This code found in iSeries® Java thin or pluggable application clients should be removed to prevent migration and compatibility problems. The java.security file from your WebSphere instance in the properties directory is now used to configure the security providers.

The security providers were set programmatically in the main() method and occurred prior to any code that accessed enterprise beans:

```
import java.security.*;
  ...
  if (System.getProperty("os.name").equals("OS/400")) {

    // Set the default provider list first.
    Provider jceProv = null;
    Provider jsseProv = null;
    Provider sunProv = null;

    // Allow for when the Provider is not needed, when
    // it is not in the client application's classpath.
    try {
      jceProv = new com.ibm.crypto.provider.IBMJCE();
    }
    catch (Exception ex) {
 ex.printStackTrace();
    throw new Exception("Unable to acquire provider.");
 }

    try {
      jsseProv = new com.ibm.jsse.JSSEProvider();
    }
    catch (Exception ex) {
 ex.printStackTrace();
    throw new Exception("Unable to acquire provider.");
 }

    try {
      sunProv = new sun.security.provider.Sun();
    }
    catch (Exception ex) {
 ex.printStackTrace();
    throw new Exception("Unable to acquire provider.");
 }

    // Enable providers early and ahead of other providers
    // for consistent performance and function.
    if ( (null != sunProv) && (1 != Security.insertProviderAt(sunProv, 1)) ) {
      Security.removeProvider(sunProv.getName());
      Security.insertProviderAt(sunProv, 1);
    }
    if ( (null != jceProv) && (2 != Security.insertProviderAt(jceProv, 2)) ) {
      Security.removeProvider(jceProv.getName());
      Security.insertProviderAt(jceProv, 2);
    }
    if ( (null != jsseProv) && (3 != Security.insertProviderAt(jsseProv, 3)) ) {
      Security.removeProvider(jsseProv.getName());
      Security.insertProviderAt(jsseProv, 3);
    }

    // Adjust default ordering based on admin/startstd properties file.
    // Maximum allowed in property file is 20.
    String provName;
```

```
    Class  provClass;
    Object provObj =  null;

    for (int i = 0; i < 21; i++) {
      provName = System.getProperty("os400.security.provider."+ i);

      if (null != provName) {

        try {
          provClass = Class.forName(provName);
          provObj = provClass.newInstance();
        }
        catch (Exception ex) {
          // provider not found
          continue;
        }

        if (i != Security.insertProviderAt((Provider) provObj, i)) {

          // index 0 adds to end of existing list
          if (i != 0) {
            Security.removeProvider(((Provider) provObj).getName());
            Security.insertProviderAt((Provider) provObj, i);
          }
        }
      } // end if (null != provName)
    } // end for (int i = 0; i < 21; i++)
  } // end if ("os.name").equals("OS/400")
```

Configuring secure access for stand-alone clients

The Thin Client for JMS with WebSphere Application Server and the Resource Adapter for JMS with WebSphere Application Server use the standard Java Secure Socket Extension (JSSE) that all supported JREs provide for making Secure Sockets Layer (SSL) connections.

About this task

When you are configuring secure connections for the Thin Client for JMS with WebSphere Application Server or the Resource Adapter for JMS with WebSphere Application Server, you can choose between the following approaches:

- A global configuration approach that affects all stand-alone outbound connections from the process.
- A private approach applies only to client or resource adapter connections from the process.

For further information refer to the Securing JMS client and JMS resource adapter connections topic.

Procedure

Decide on which configuration approach you want to use, then configure the secure connections for the Thin Client for JMS with WebSphere Application Server or the Resource Adapter for JMS with WebSphere Application Server according to your selected approach.

Chapter 2. Securing Data access resources

This page provides a starting point for finding information about data access. Various enterprise information systems (EIS) use different methods for storing data. These backend data stores might be relational databases, procedural transaction programs, or object-oriented databases.

The flexible IBM WebSphere Application Server provides several options for accessing an information system backend data store:

- Programming directly to the database through the JDBC 4.0 API, JDBC 3.0 API, or JDBC 2.0 optional package API.
- Programming to the procedural backend transaction through various J2EE Connector Architecture (JCA) 1.0 or 1.5 compliant connectors.
- Programming in the bean-managed persistence (BMP) bean or servlets indirectly accessing the backend store through either the JDBC API or JCA-compliant connectors.
- Using container-managed persistence (CMP) beans.
- Using the IBM data access beans, which also use the JDBC API, but give you a rich set of features and function that hide much of the complexity associated with accessing relational databases.

Service Data Objects (SDO) simplify the programmer experience with a universal abstraction for messages and data, whether the programmer thinks of data in terms of XML documents or Java objects. For programmers, SDOs eliminate the complexity of the underlying data access technology, such as, JDBC, RMI/IIOP, JAX-RPC, and JMS, and message transport technology such as, java.io.Serializable, DOM Objects, SOAP, and JMS.

Securing data sources

Java EE connector security

The Java 2 Platform, Enterprise Edition (Java EE) connector architecture defines a standard architecture for connecting J2EE to heterogeneous enterprise information systems (EIS). Examples of EIS include Enterprise Resource Planning (ERP), mainframe transaction processing (TP) and database systems.

The connector architecture enables an EIS vendor to provide a standard resource adapter for its EIS. A resource adapter is a system-level software driver that is used by a Java application to connect to an EIS. The resource adapter plugs into an application server and provides connectivity between the EIS, the application server, and the enterprise application. Accessing information in EIS typically requires access control to prevent unauthorized accesses. J2EE applications must authenticate to the EIS to open a connection.

The J2EE connector security architecture is designed to extend the end-to-end security model for J2EE-based applications to include integration with EIS environments. An application server and an EIS collaborate to ensure the correct authentication of a resource principal, which establishes a connection to an underlying EIS. The connector architecture identifies the following mechanisms as the commonly supported authentication mechanisms, although other mechanisms can be defined:

- BasicPassword: Basic user-password-based authentication mechanism that is specific to an EIS

Applications define whether to use application-managed sign-on or container-managed sign-on in the resource-ref elements in the deployment descriptor. Each resource-ref element describes a single connection factory reference binding. The res-auth element in a resource-ref element, whose value is either `Application` or `Container`, indicates whether the enterprise bean code can perform sign-on or whether application server can sign-on to the resource manager using the principal mapping configuration. The resource-ref element is typically defined at application assembly time with an assembly tool. The `resource-ref` can also be defined, or redefined, at deployment time.

© IBM Corporation 2009

Application managed sign-on

To access an EIS system, applications locate a connection factory from the Java Naming and Directory Interface (JNDI) namespace and invoke the getConnection method on that connection factory object. The getConnection method might require a user ID and password argument. A J2EE application can pass in a user ID and password to the getConnection method, which subsequently passes the information to the resource adapter. Specifying a user ID and password in the application code might compromise some security, however.

The user ID and password, if coded into the Java source code, are available to developers and testers in the organization. Also, the user ID and password are visible to users if they decompile the Java class.

The user ID and password cannot be changed without first requiring a code change. Alternatively, application code might retrieve sets of user IDs and passwords from persistent storage or from an external service. This approach requires that IT administrators configure and manage a user ID and password using the application-specific mechanism.

To access this authentication data, the application server supports a component-managed authentication alias to be specified on a resource. This authentication data is common to all references to the resource. Click **Resources** > **Resource Adapters** > **J2C connection factories** > **configuration_name**. Select Use component-managed authentication alias.

When `res-auth=Application`, the authentication data is taken from the following elements, in order:

1. The user ID and password that are passed to the getConnection method
2. The component-managed authentication alias in the connection factory or the data source
3. The custom properties user name and password in the data source

The user name and password properties can be initially defined in the resource adapter archive (RAR) file.

These properties can also be defined in the administrative console or using wsadmin scripting from custom properties.

Container-managed sign-on

The user ID and password for the target enterprise information systems (EIS) can be supplied by the application server. The product provides container-managed sign-on functionality. The application server locates the proper authentication data for the target EIS to enable the client to establish a connection. Application code does not have to provide a user ID and password in the getConnection call when it is configured to use container-managed sign-on, and authentication data does not have to be common to all references to a resource. The uses a Java Authentication and Authorization Service (JAAS) pluggable authentication mechanism to use a pre-configured JAAS login configuration, and LoginModule to map a client security identity and credentials on the running thread to a pre-configured user ID and password.

The product supports a default many-to-one credential mapping LoginModule module that maps any client identity on the running thread to a preconfigured user ID and password for a specified target EIS. The default mapping module is a special purpose JAAS LoginModule module that returns a PasswordCredential credential that is specified by the configured Java 2 connector (J2C) authentication data entry. The default mapping LoginModule module performs a table lookup, but does not perform actual authentication. The user ID and password are stored together with an alias in the J2C authentication data list.

The J2C authentication data list is located on the Global security panel from Java Authentication and **Authorization Service** > **J2C Authentication data**. The default principal and credential mapping function is defined by the DefaultPrincipalMapping application JAAS login configuration.

J2C authentication data that is modified using the administrative console takes effect when the modification is saved into the repository, and Test Connection is performed. Also, J2C authentication data that is modified using wsadmin scripting takes effect when any application is started or restarted for a given the application server server process. J2C authentication data modification takes effect by invoking the SecurityAdmin MBean method, updateAuthDataCfg. Set the HashMap parameter to null to enable the Securityadmin MBean to refresh the J2C authentication data using the latest values in the repository.

Do not modify the DefaultPrincipalMapping login configuration because the product includes performance enhancements to this frequently used default mapping configuration. The product does not support modifying the DefaultPrincipalMapping configuration, changing the default LoginModule module, or stacking a custom LoginModule module in the configuration.

On the z/OS® platform, if a user ID and password are not present or defaulted by an alias, the WebSphere Application Server runtime searches for a System Authorization Facility (SAF) user ID in the subject.

For most systems, the default method with a many-to-one mapping is sufficient. However, the product does support custom principal and credential mapping configurations. Custom mapping modules can be added to the application logins JAAS configuration by creating a new JAAS login configuration with a unique name. For example, a custom mapping module can provide one-to-one mapping or Kerberos functionality.

Trusted connections also provide a one-to-one mapping while supporting client identity propagation. In addition by utilizing the DB2® trusted context object, trusted connections can take advantage of connection pooling to reduce the performance penalty of closing and reopening connections with a different identity. Using trusted connections also enhances the security of your DB2 database by eliminating the need to assign all privileges to a single user. The connection is established by a user whose credentials are trusted by the DB2 server to open the connection and the same user is also then trusted to assert the identity of the other users accessing the DB2 server from the application. A new mapping configuration called TrustedConnectionMapping has been created to implemented trusted connections.

You also can use the WebSphere Application Server administrative console to bind the resource manager connection factory references to one of the configured resource factories. If the value of the res-auth element is `Container` within the deployment descriptor for your application, you must specify the mapping configuration. To specify the mapping configuration, use the Resource references link under References on the **Applications** > **Application Types** > **Websphere enterprise applications** > **application_name panel**. See the topic, Mapping resource references to references, for additional directions.

J2C mapping modules and mapping properties

Mapping modules are special JAAS login modules that provide principal and credential mapping functionality. You can define and configure custom mapping modules using the administrative console.

You also can define and pass context data to mapping modules by using login options in each JAAS login configuration. In the product, you also can define context data using mapping properties on each connection factory reference binding.

Login options that are defined for each JAAS login configuration are shared among all resources that use the same JAAS login configuration and mapping modules. Mapping properties that are defined for each connection factory reference binding are used exclusively by that resource reference.

Consider a usage scenario where an external mapping service is used.

For example, you might use the Tivoli® Access Manager global sign-on (GSO) service. Use the Tivoli Access Manager GSO to locate authentication data for both backend servers.

You have two EIS servers: DB2 and MQ. The authentication data for DB2 is different from that for MQ, however. Use the login option in a mapping JAAS login configuration to specify the parameters that are required to establish a connection to the Tivoli Access Manager GSO service. Use the mapping properties in a connection factory reference binding to specify which EIS server requires the user ID and password.

For more detailed information about developing a mapping module, see the topic, J2C principal mapping modules.

Note:

- The mapping configuration at the connection factory has moved to the resource manager connection factory reference. The mapping login modules that are developed using WebSphere Application Server Version 5 JAAS callback types can be used by the resource manager connection factory reference, but the mapping login modules cannot take advantage of the custom mapping properties feature.
- Connection factory reference binding supports mapping properties, and passes those properties to mapping login modules by way of a new WSMappingPropertiesCallback callback. In addition, the WSMappingPropertiesCallback callback and the new WSManagedConnectionFactoryCallback callback are defined in the com.ibm.wsspi package. Use the new mapping login modules with the new callback types.

Secure message delivery with inbound SecurityContext

Security information can be supplied by an EIS resource adapter to the application server using security inflow context. The security inflow context mechanism enables the work manager to execute the actions of a Work instance under an established identity. Those actions include the delivery of messages to Java EE message endpoints that are processed as message-driven beans (MDB) under an identity that is configured in a security domain of the application server.

Attention: Security inflow context is only supported for JCA 1.6-compliant resource adapters. Currently, the product does not provide a built-in resource adapter that supports a security inflow context. Using inbound SecurityContext to secure message delivery requires a resource adapter that supports Java EE messaging and security inflow context.

Secure message delivery to a message endpoint requires that global security is enabled in the global security configuration. It also requires that application security is enabled on the application server that hosts the application providing the message endpoint MDB. For more information about global security, see the topic, “Global security settings”.

The security policy of the application deployment descriptor must be configured with roles that are associated with user identities in the application realm, which is the user registry of the security domain that scopes to the application. This security configuration enables EJB security and authorizes specific user identities in the application realm to access MDB methods. For more information about security overview, see the topic, “Security”.

Secure message delivery also requires the resource adapter to define implementations for both the WorkContextProvider and SecurityContext interfaces. To deliver a secure message, a resource adapter first submits a work instance that provides a SecurityContext implementation, which the work manager uses to establish the execution subject for that work instance.

While establishing the execution subject, a SecurityContext can provide implementations of Java Authentication Service Provider Interface for Containers (JASPIC) callbacks that the work manager uses to determine caller and group identities (CallerPrincipalCallback, GroupPrincipalCallback), and to authenticate the caller identity and password (PasswordValidationCallback). If the caller identity is in the application realm, then the work manager asserts that identity by constructing a WSSubject instance containing the caller principal, any group principals, and all private credentials.

Alternatively, the SecurityContext can provide an execution subject that is an instance of WSSubject instance created by another login or authentication module. The work manager accepts this WSSubject instance only when its caller principal is in the application realm or within a trusted realm. For more information about realms, see the topic, “Configuring inbound trusted realms for multiple security domains”.

The work manager rejects a Work instance whenever it cannot establish a WSSubject instance. Otherwise, it dispatches the instance on a managed thread under the WSSubject instance that was either asserted or accepted. If the SecurityContext provides no caller identity, the Work instance dispatches under a WSSubject instance containing the unauthenticated caller principal.

When dispatched, a Work instance might attempt to deliver messages to the MDB of the secured application. All messages are delivered under the WSSubject instance established for the Work instance. The EJB security collaborator affords access to the onMessage method of the MDB whenever the caller principal of the WSSubject instance is associated with a role that is declared in the application deployment descriptor. Otherwise, the collaborator denies access and the message fails to deliver. During delivery, the MDB can use the EJB Context methods, isCallerInRole and getCallerPrincipal, to make additional access decisions, and the MDB might access other entities in the security domain for which the caller principal has authorization.

Enabling trusted context for DB2 databases

Enable trusted context in your applications to improve how the application server interacts with DB2 database servers. Use trusted connections to preserve the identity records of clients that are connecting to a DB2 database through your applications; trusted connections can provide a more secure environment by granting access based on the identity of those users.

Before you begin

Ensure that the following prerequisites are met before enabling trusted connections:

- You are using a database server that is running DB2 Database for Linux, UNIX, and Windows Version 9.5 or later or DB2 Database Version 9.1 or later for z/OS. See the list of list of supported software for the application server for more support information.
- You do not need to be connected to the database to configure trusted context in the application server.
- Trusted context is enabled for the DB2 database.
- Global security is enabled. See the topic, Setting up, enabling and migrating security, for more information on configuring security.

About this task

With trusted connections you can:

- Access the DB2 database with the caller identity, obviating the need to create a new connection for every user.
- Preserve the identity of the end-user when the application server is interacting with the database.
- Strengthen database security by avoiding granting all of the privileges to a single user.
- Improve performance, as compared to the existing model of using the resetConnection() method to take advantage of identity propagation.

Note: Non-trusted connections cannot be used as trusted connections. If the connection pool contains only non-trusted connections and a request comes in for a trusted connection, a new request will be sent to the database for the trusted connection.

Procedure

Enable trusted context for your applications.

- Enable trusted context when you are installing a new application.
  1. Perform a typical installation for the application until you reach **Step 7: Map resource references to resources** in the installation wizard.
  2. In **Step 7: Map resource references to resources**, select **Use trusted connections (one-to-one mapping)** in the **Specify authentication method** section.
  3. Select an authentication alias from the list that matches an alias that is already defined in the DB2 data source. If you do not have an alias defined that is suitable, continue with the installation, and enable trusted context after the application is installed.

     Note: You can specify a default user (UNAUTHENTICATED) to be used if no client identity is available, but that default ID (UNAUTHENTICATED) must also exist in the DB2 database. If the com.ibm.mapping.unauthenticatedUser is set to null or an empty string, then the application server will use the default user (UNAUTHENTICATED). For more information, see the information about setting the security properties for trusted connections.
  4. Select a data source from the table that has trusted context enabled.
  5. Click **Apply**.
  6. Edit the properties of the custom login configuration. Read the topic, Setting the security properties for trusted connections.

     Note: Ensure that all of the authentication values are set to `none` for the trusted connections to work. For example, if you used a trusted connection to connect to DB2, the **Test connection** button will not work and the operation will fail:

     ```
     The test connection operation failed for data source jdbcTestDB on server server1
     at node wasvm04Node02 with the following exception: java.sql.SQLException:
     [jcc][t4][10205][11234][3.59.81] Null userid is not supported. ERRORCODE=-4461,
     SQLSTATE=42815 DSRA0010E: SQL State = 42815, Error Code = -4,461.
     View JVM logs for further details.
     ```
  7. Finish the installation wizard.
- Enable trusted context on an application that is already installed.

  Note: Remove the propagateClientIdentityUsingTrustedContext custom property for the DB2 data source, if it is present. If the propagateClientIdentityUsingTrustedContext is enabled, the application server will issue the following warning at run time:

```
IDENTITY_PROPAGATION_PROP_WARNING=DSRA7029W: The propagateClientIdentityUsingTrustedContext
 custom property for the Datasource is no longer used, value will be ignored.
```

  The application server will determine at run time if the request is using trusted context, and the application server will enable trusted context based on that information. Therefore, the same data source in the application server can be used for both trusted and non-trusted access.
  1. Click **Websphere enterprise applications** > *application_name*.
  2. Click **Resource references** from the **Resources** heading.
  3. Select **Use trusted connections (one-to-one mapping)** in the **Specify authentication method** section.
  4. Select an authentication alias from the list that matches an alias that is already defined in the DB2 data source. If you do not have an alias defined that is suitable, define a new alias.
     a. Click **JDBC** > **Data sources** > *data_source_name*.
     b. Click **JAAS - J2C authentication data** from the **Related Items** heading.
     c. Click **New**.
     d. Define the properties for the alias in **General properties**.

e. Click **OK**.

Note: You can specify a default user (UNAUTHENTICATED) to be used if no client identity is available, but that default ID (UNAUTHENTICATED) must also exist in the DB2 database. If the com.ibm.mapping.unauthenticatedUser is set to null or an empty string, then the application server will use the default user (UNAUTHENTICATED). For more information, see the information about setting the security properties for trusted connections.

5. Select a data source from the table that has trusted context enabled.
6. Click **Apply**.
7. Edit the properties of the custom login configuration. Read the topic, Setting the security properties for trusted connections.

What to do next

Be aware of the following error conditions that can occur if trusted context is not configured properly:

- The application server will issue a warning if you use the TrustedConnectionMapping login configuration and the database server does not support trusted context. The application server will then return a normal, non-trusted connection. If you are using a DB2 database for the database server, and it doesn't support trusted connections, then the DB2 database server will throw an exception.
- The application server will throw the following exception if you use the TrustedConnectionMapping login configuration and ThreadIdentity is specified:

```
IDENTITY_PROPAGATION_CONFLICT2_ERROR=DSRA7028E: You cannot use the TrustedConnectionMapping
 login configuration when the ThreadIdentity property is enabled.
```

- The application server will throw the following exception if you use the TrustedConnectionMapping login configuration and reauthentication is specified:

```
IDENTITY_PROPAGATION_CONFLICT1_ERROR=DSRA7025E: The reauthentication custom property for
 the Datasource cannot be enabled when you are using the TrustedConnectionMapping login configuration.
```

Setting the security properties for trusted connections

Trusted connections are a solution that can pass the requesting user identity to DB2 and also take full advantage of the connection pooling. Utilizing the DB2 trusted context object, the trusted connection is used to separate the identity used to establish the connection from the identity that accessed the DB2 server services. The connection is established by a user whose credentials are authorized by the DB2 server to open the connection and trusted by the DB2 server to assert the identity of the requesting users when accessing the DB2 server from the application.

Before you begin

To use the trusted connection functionality, you must be running at database server with DB2 Database for Linux, UNIX, and Windows Version 9.5 or later or DB2 Database Version 9.1 or later for z/OS. Trusted connections can be used if the application server is installed on iSeries systems, as long as a supported version of DB2 is installed on a platform other than iSeries systems, and the DB2 universal driver is used. See the list of list of supported software for the application server for more support information. An existing J2EE connector (J2C) data alias must exist for passing user credentials to the DB2 server when establishing a connection, meaning container authorization must be used.

Read about “Enabling trusted context for DB2 databases” on page 9 for steps to configure the application server to use trusted connections.

About this task

Trusted connections support client identity propagation while taking advantage of connection pooling to reduce the performance penalty of closing and reopening connections with a different identity. When you select **Use trusted connection (one-to-one mapping)** for the connection mapping, five custom properties

are created. Review these properties to ensure that the default values of these properties correspond with your intended settings.

Procedure

1. Click **Enterprise applications** > ***application_name*** > **Resource references** > **Resources** panel in the administrative console.
2. Select the correct enterprise bean, and click **Mapping Properties** to view the properties that are set by default when you configured the trusted connection.
3. Confirm that the default values assigned to these properties are correct for your environment.

Table 1. Security Properties. This table lists the security property values:

Property	Default Value	Information
com.ibm.mapping.authDataAlias	none	The value that is assigned for this property is the value that you selected from the menu list.
com.ibm.mapping.propagateSecAttrs	false	A false value for this property specifies that the security attributes are not propagated. You can change this value to true to add the RunAs subject as an opaque token in the IdentityPrincipal object.
com.ibm.mapping.targetRealmName	null	If this value is not specified or null, the security run time process will use the current user realm name. This process assumes that the Enterprise Information System (EIS) is using the current user realm. In this context, a realm is a logical representation of the user repository. If the application server and DB2 server are using different user repositories, the value of this property should be set to the realm name of the DB2 server. This enables a principal or credential mapping to be set at the target EIS.
com.ibm.mapping.unauthenticatedUser	UNAUTHENTICATED	This property is a user identity that is used by the EIS to indicate a user identity that is unauthenticated. This is defined at com.ibm.ISecurityUtilityImpl.SecConstants.java public final static String UnauthenticatedString = "UNAUTHENTICATED"
com.ibm.mapping.useCallerIdentityproperty	false	A false value for this property specifies the Run As identity is asserted in the IdentityPrincipal object. Change the value of this property to true if you want to assert the caller identity in the IdentityPrincipal object instead of the Run As identity.

4. Click **OK** to confirm all the current values.
5. Click **OK** and **Save** on the Resource references panel to save your changes to the master configuration.

Results

After the completion of these steps and a restart of the application server, trusted connections will be used with the chosen mapping properties to connect with the DB2 database server.

Trusted connections with DB2

Trusted connections allow for the application server to use DB2 Trusted Context objects to establish connections with a user whose credentials are trusted by the DB2 server to open the connection. By establishing a Trusted Context, this user is then trusted to assert other user identities on the DB2 server without the expense of reauthentication. This also enhances the security of your DB2 database by eliminating the need to assign all privileges to a single user. Implementing trusted connections results in client identity propagation while leveraging connection pooling to eliminate the performance penalty of closing and reopening connections with a different identity.

Restriction: To use the trusted connection functionality you must be using DB2 Database for Linux, UNIX, and Windows Version 9.5 or later or DB2 Database Version 9.1 or later for z/OS. You can use trusted connections if version 7.0 is installed on an iSeries system as long as a supported version of DB2 is installed on a platform other than an iSeries system and the DB2 universal driver is being used. See the list of supported software for the application server for more support information.

To reduce the significant expense of establishing new connections, the connection manager maintains a connection pool in which each connection is tracked by the credential originally used to open the connection. When an application needs a connection, the connection manager uses the credential object to match a free connection from the connection pool. If no free connection is available and the maximum number of connections has not been reached, the connection pool manager opens a new connection using that credential object. This connection mapping is the default connection mapping used by the application server and is known as a many-to-one credential mapping because the connection is opened using the credential object in the subject, which is usually not the same as the RunAs identity. This simple mapping supports easy connection pooling, but the caller identity is never propagated to database server.

To propagate the caller identify to the database server, you can plug in a Java Authentication and Authorization Service (JAAS) login module. Using this method, you would map the application server user credential to the user credential suitable for the database server security realm. This approach maintains the caller identity, but does not use connection pooling.

Trusted connections are used instead of the default mapping or a JAAS mapping to connect to the data source. Trusted connections support client identity propagation and can also use connection pooling to reduce the performance penalty of closing and reopening connections with a different identity. Trusted connections use the DB2 trusted context object.

The DB2 trusted context is an object that the database administrator defines and that contains a system authorization ID and a set of trust attributes. The trust attributes identify characteristics of a connection that are required for the connection to be considered trusted. The relationship between a database connection and a trusted context is established when the connection to the DB2 server is created. After a trusted context is defined, and an initial trusted connection to the DB2 database server is made, the application server can use that database connection from a different user without a full reauthentication. This is because an authentication token is required with the user identity. The database authenticates the user and then verifies the user authorization to access the database before allowing any database requests to be processed on behalf of that user.

Restriction: For a z/OS server, the user identity must be the Resource Access Control Facility (RACF®) ID.

Using the trusted connection provides the needed plug-in points to support adding your own secure implementation of the DB2 trusted context. Trusted connections separate the identity used to establish the connection from the identity that accesses the back-end server services. The connection is established by a user whose credentials are trusted by the DB2 server to open the connection. The same user is also then trusted to assert the identity of the other users. This assertion also helps strengthen database security by eliminating the need to grant all privileges to a single user.

When the application requests a connection to the database, the connection manager can find any idle trusted connection and assert the user identity to the backend server. All the operations performed on the backend server are from the asserted user identity. The use of an identity mapping may still be needed if the back-end server uses a different user repository than that of the application server.

A new mapping configuration called TrustedConnectionMapping implements trusted connections. The TrustedConnectionMapping configuration maps the RunAs subject to a resource subject that contains the following elements:

- A resource principal object that this resource subject represents

- A PasswordCredential object in the private credential set
- An IdentityPrincipal object in the principal set

The principal object represents the RunAs identity. The PasswordCredential object contains a user ID and password to be used by the resource adapter to establish the trusted connection. The IdentityPrincipal object by default contains the RunAs identity, but can be changed to use the identify of the caller. The IdentityPrincipal object also contains an original user identity that represents the user who sent the request initially, an optional realm name that indicates the set of registries where the user identity is defined and an optional security token, which is a serialized security context of the user.

Enabling trusted context with authentication for DB2 databases

Enable trusted context in your applications to improve how the application server interacts with DB2 database servers. Use trusted connections to preserve the identity records of clients that are connecting to a DB2 database through your applications; trusted connections can provide a more secure environment by granting access based on the identity of those users. DB2 provides an option for trusted connections in which a password is required when switching the user identity. You can configure the application server to use trusted connections with authentication, and plug-in your own code to take advantage of trusted context with authentication.

Before you begin

Refer to the topic on enabling trusted context for DB2 databases to ensure that trusted connections are properly configured for the application server.

About this task

If the WITH AUTHENTICATION option is specified when the trusted context is created, the database requires that you provide an authentication token with the end user's identity. The database authenticates the end user and verifies the end user's authorization to access the database before the database allows any requests to be processed.

The end user's identity must be the RACF ID.

Procedure

1. Set useTrustedContextWithAuthentication custom property to true for the DB2 data source.
   a. Click **JDBC** > **Data sources**.
   b. Click the name of the data source that you want to configure.
   c. Click **Custom properties** from the **Additional Properties** heading.
   d. Click **New**.
   e. Complete the required fields. Use the following information:

Name	Value
useTrustedContextWithAuthentication	true

   If the useTrustedContextWithAuthentication custom property is not set to true, the application server will provide an implementation of reusing DB2 trusted connections without authentication at run time. In this case you are not required to provide anything to use the trusted context feature.
2. Use the login configuration for TrustedConnectionMapping, as described in the topic on enabling trusted context for DB2 databases.
3. Extend the DataStoreHelper class, and provide the implementation for the getPasswordForUseWithTrustedContextWithAuthentication method as described in the topic on developing a custom DataStoreHelper class. At run time, the application server will call this method to return the password that the application server is required to use to switch the trusted context identity when you have enabled trusted context with authentication. The password that is returned by this

method will be sent to the database when the application server switches trusted context identities, and the password will not be stored by the application server.

This application server only calls this method if the following is true:

- You set the useTrustedContextWithAuthentication data source custom property to true.
- You use the TrustedConnectionMapping login configuration.

The following is an example of the getPasswordForUseWithTrustedContextWithAuthentication method:

```
public String getPasswordForUseWithTrustedContextWithAuthentication(String identityname, String realm)
              throws SQLException
   {
      return customersOwnUtility().getPassword(identityname)   // customers use their own
                                                               // implementation to get the password
   }
```

Note: You cannot enable the useTrustedContextWithAuthentication custom property for the data source without overwriting the getPasswordForUseWithTrustedContextWithAuthentication method in the DataStoreHelper class to get the password for switching the identity for trusted connections. If you do not provide implementation for the getPasswordForUseWithTrustedContextWithAuthentication method, the application server will throw an exception with the following message at run time:

```
TRUSTED_WITH_AUTHENTICATION_IMPLEMENTATION_ERROR=DSRA7033E: You cannot enable the
useTrustedContextWithAuthentication custom property for the data source without
overwriting the getPasswordForUseWithTrustedContextWithAuthentication DataStoreHelper.
TRUSTED_WITH_AUTHENTICATION_IMPLEMENTATION_ERROR.explanation=The
useTrustedContextWithAuthentication custom property is enabled, but the implementation
code for the DataStoreHelper method that will return the password that the application
server will use to switch the identity is not provided.
TRUSTED_WITH_AUTHENTICATION_IMPLEMENTATION_ERROR.useraction=Overwrite the
getPasswordForUseWithTrustedContextWithAuthentication DataStoreHelper method and
provide the implementation code that will return the password, or set the
useTrustedContextWithAuthentication custom property for the data source to false.
```

Configuring the application server and DB2 to authenticate with Kerberos

The Kerberos authentication mechanism may be used when both WebSphere Application Server and the DB2 server are configured for Kerberos authentication. Kerberos authentication can provide single sign on (SSO) end-to-end interoperable solutions and preserves the original requester identity.

Before you begin

In the application server, you can configure a DB2 data source, the application server, and your application so that the DB2 data source and the application server interoperate using delegated Kerberos credentials in an end-to-end manner for database access by the application. In order to take advantage of DB2 Kerberos authentication using delegated credentials from the application server, referred to in this topic as *option 1*, you need to configure both DB2 and the application server to use Kerberos as the authentication mechanism. See the topic, “Kerberos (KRB5) authentication mechanism support for security” to learn how to set up Kerberos as the authentication mechanism in this version of the application server.

The XARecovery and TestConnection facilities of the application server are not able to supply delegated Kerberos credentials to the data source. There might also be situations where the application server security component is unable to supply delegated Kerberos credentials for a given connection request. To account for these cases you can configure a DB2 connection using Kerberos authentication referred to in this topic as *option 2*. For this option, a user ID and password must be supplied to the JDBC driver that the driver uses to obtain its own Kerberos credentials. To use this option, you must configure a J2C authentication data alias on the application server which defines the user ID and password that the DB2

JDBC driver will use to request a Kerberos Ticket Granting Ticket (TGT). The TGT is used for Kerberos authentication to a DB2 server. To the application server, this looks much like the typical user ID and password authentication.

You must use a DB2 JDBC driver that supports Kerberos authentication and is operating in type 4 mode. The supported JDBC drivers are:

- IBM Data Server Driver for JDBC and SQLJ (identified in the application server as DB2 using IBM JCC Driver)
- IBM DB2 JDBC Universal Driver Architecture (identified in the application server as DB2 Universal JDBC Driver Provider)

About this task

Use the following steps to configure the application server and DB2 to authenticate with Kerberos:

Procedure

1. Configure the DB2 Server for Kerberos authentication. Refer to DB2 Kerberos security documentation in the DB2 information center, for example, the topic “Kerberos authentication details”. Another helpful reference is “DB2 UDB Security, Part 6” that is located on the IBM developerWorks website. Verify that DB2 Kerberos authentication is working.
2. Configure the application server to use Kerberos security. See the topic “Configuring Kerberos as the authentication mechanism using the administrative console”. Verify that application server Kerberos authentication is working.
3. Configure the DB2 data source in the application server to use Kerberos authentication. There are two steps to complete this task, you need to configure the resource adapter in the application server to pass Kerberos credentials and password credentials to the JDBC driver and secondly, you need to configure the JDBC driver to use Kerberos authentication when connecting to the DB2 server. For more information on completing these steps, see the topic “Configuring a data source using the administrative console”.

Table 2. Custom properties and values. When configuring the DB2 data source, you must pay particular attention to the security settings and the custom properties.

Name	Value
kerberosServerPrincipal **Note:** This property is optional except when connecting to a DB2 server that is running on a z/OS platform (as of DB2 for LUW, v8 FP11).	user@REALM or service_name/hostname@REALM
SecurityMechanism **Note:** A value for this property of 11, indicates that the JDBC driver needs to use Kerberos authentication when connecting to the DB2 server.	11

   a. For option 2, you should configure the “Mapping-configuration alias” to “DefaultPrincipalMapping”, or another login configuration which does not generate GSSCredentials, and set the “Container-managed authentication alias” to reference an alias to use for Kerberos login by the JDBC driver. The testConnection facility also uses this alias if no component-managed authentication alias is configured.
   b. For option 1, delegated Kerberos credentials, you should configure the “Mapping-configuration alias” to “KerberosMapping”. This will indicate that the resource adapter in the application server should provide delegated credentials to the DB2 JDBC driver. The testConnection facility and XA transaction recovery facility are not able to supply delegated Kerberos credentials, but can revert to option 2 authentication. If you do not need those features, you can select *none* for each of the authentication aliases. If testConnection is used and a valid authentication alias is configured, an informational message, DSRA82211, is logged. This message indicates that testConnection is not

able to offer Kerberos credentials. If no alias is configured, then testConnection fails with a Kerberos invalid credentials error reported by the JDBC driver.

Important: If KerberosMapping is configured, but the security component is unable to provide Kerberos credentials for a particular connection request, the resource adapter can be configured to revert to connection authentication using Default Principle Mapping. To configure this fallback, select an alias from the container-managed authentication alias list. To disable this fallback, select *none* from the container-managed authentication alias list.

4. To enable Kerberos mapping (option 1), you also must specify container-managed authentication. To specify container-managed authentication, the application must use a resource reference to lookup the data source The resource reference must specify KerberosMapping as the login configuration. If a login configuration is specified for a resource reference, then for application access through that resource reference, the specified login configuration takes precedence over the mapping-configuration alias value specified on the data source. A container-managed authentication alias can also be specified on the resource reference.

Connection thread identity

The application server for z/OS allows you to assign a thread identifier as an owner of a connection, when you first obtain the connection. The thread identity function only applies to Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) resource adapters and Relational Resource Adapter (RRA) wrappered Java Database Connectivity (JDBC) providers that support the use of thread identity for connection ownership.

In this article the term thread identity refers to the Java EE Identity (such as the RunAs Identity), as opposed to the OS thread identity. Refer to the topic, Synchronizing a Java thread identity and an operating system thread identity, and the topic, Understanding Connection Manager RunAs Identity Enabled and operating system security, for more information.

The following table lists the JCA resource adapter and JDBC provider processes that support thread identity and thread security. It also provides the level of thread identity support:

Table 3. JCA resource adapter and JDBC provider support for thread identity and thread security. Read the next section for definitions of thread identity support.

Connectors	Thread identity support	OS thread security
IMS™ Connector - local `ConnectionFactory` configuration	ALLOWED	Not supported
IMS Connector - remote `ConnectionFactory` configuration	NOTALLOWED	Not supported
CTG CICSECIConnector - local `ConnectionFactory` configuration	ALLOWED	Not supported
CTG CICSECIConnector - remote `ConnectionFactory` configuration	NOTALLOWED	Not supported
IMS JDBC Connector - local `ConnectionFactory` configuration (By default, IMS JDBC only supports this type of configuration.)	REQUIRED	True
RRA DB2 for z/OS local JDBC provider - data sources configured to the local DB2	ALLOWED	True
RRA DB2 Universal JDBC Driver Provider using Type 2 connectivity	ALLOWED	True

Table 3. JCA resource adapter and JDBC provider support for thread identity and thread security (continued). Read the next section for definitions of thread identity support.

Connectors	Thread identity support	OS thread security
RRA DB2 Universal JDBC Driver Provider using Type 4 connectivity	NOTALLOWED	Not supported
WebSphere MQ JMS Provider: Connection Factory (TransportType = BINDINGS)	ALLOWED	True
WebSphere MQ JMS Provider - Connection Factory (TransportType = CLIENT)	NOTALLOWED	Not supported
WebSphere JMS Provider (such as Integral JMS Provider): Connection Factory	NOTALLOWED	Not supported

WebSphere Application Server for z/OS allows resource adapters and JDBC providers to define the level of thread identity support for the defined connection factories or data sources. The level of support can be:
- `ALLOWED`, which indicates thread identity for connection ownership is allowed for this configuration.
- `NOTALLOWED`, which indicates thread identity for connection ownership is not allowed for this configuration.
- `REQUIRED`, which indicates thread identity for connection ownership is required.

The thread identity function is only available in those server configurations where JCA connectors or JDBC providers access local z/OS resources through callable (not TCP/IP) interfaces. So, for example, CICS® and IMS provide thread identity support only if the target CICS or IMS is configured on the same system as the z/OS WebSphere Application Server.

To use thread identity when getting connections to a connection factory or JDBC data source for your application, you must specify `resauth=Container` for the connection factory or JDBC data source. Use the Eclipse assembly tool or WebSphere Studio Application Developer Integration Edition (WSADIE) to indicate the `resauth=Container` setting.

When the level of thread identity support provided by the connector configuration is `ALLOWED`, if you want to use thread identity for the connections, you cannot specify a Container-managed alias when you define the connection factory or JDBC data source. If you specify a Container-managed alias, the userid defined by the alias is assigned as the owning id for the connections obtained by the application.

When the JDBC provider supports thread identity, the thread identity function is only used when data sources configured for that provider are used by Version 2.0 EJB modules and Version 2.3 servlets.

WebSphere Application Server for z/OS also allows supported resource adapters and JDBC providers to enable *OS thread security* in conjunction with thread identity support. You can use OS thread security when:
- The server configuration supports both thread identity and thread security.
- The Connection Manager RunAs Identity Enabled property is enabled.

  You can configure the server to allow Connection Manager RunAs Identity Enabled support. To enable this option, click **Security** > **Global security** > **z/OS security options** in the administrative console. On the z/OS security options panel, select the **Enable the connection manager RunAs thread identity** option, and click **Apply**.
- The z/OS security product permits synchronization of the Connection management thread identity through the BBO.SYNC FACILITY class or BBO.SYNC SURROGATE class

If these conditions are met, the system creates an access control environment element (ACEE) for the user associated with the thread.

Users of previous versions of WebSphere Application Server for z/OS will note that the instructions for enabling OS Thread Security have changed. Previously, OS Thread Security was enabled via a checkbox named Enable Synch to Thread. This checkbox still exists, but it no longer is associated with any Connection Management functionality. Users who wish to enable OS Thread Security must now use the checkbox named Connection Manager RunAs Identity Enabled

Using thread identity support

The thread identity function allows you to assign a thread identifier as an owner of a connection when you first obtain the connection. This function only applies to Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) resource adapters and Relational Resource Adapter (RRA) wrappered Java Database Connectivity (JDBC) providers that support the use of thread identity for connection ownership.

About this task

In this article the term thread identity refers to the Java EE Identity (such as the RunAs Identity), as opposed to the OS thread identity. Refer to the topics Synchronizing a Java thread identity and an operating system thread identity, and Understanding Connection Manager RunAs Identity Enabled and operating system security, for more information.

Perform the following steps to enable the thread identity function for the connection factories or JDBC provider data sources created with the supported JCA resource adapters and JDBC providers:

Procedure

1. Define **`resauth=Container`** for the application resource. See the topic Connection thread identity for details.
2. Ensure the JCA resource adapters or JDBC providers support the thread identity function.

   Review the supported resource adapters and data source providers, and the level of support: REQUIRED, ALLOWED, and NOTALLOWED. See the topic Connection thread identity for a table of the JCA resource adapter processes and the JDBC provider processes that support thread identity and thread security.

   If the adapter or provider is not listed, then thread identity support is NOTALLOWED, by default.
3. Set the **`Container-managed authentication alias`** to NULL, if you configure the connector locally.

   When the connector is configured locally, the resource adapter determines the level of thread identity support as **`ALLOWED`**. If thread identity support is allowed and you specify **`Container-managed authentication alias`** as NULL, the connector uses the current thread identity as the owner for each connection that is created.

   When the resource adapter or JDBC provider determines that the level of thread identity support is **`REQUIRED`**, any specification for the **`Container-managed authentication alias`** is ignored. Thread identity support in this case always applies.
4. Determine connector behavior when Java 2 security is a factor. See the article Security states with thread identity support for more information.

   If you want the thread identity associated with a connection to be the thread identity, then you must enable Java 2 security. In the case of JDBC providers that support the thread identity function and require the thread to be pushed to the z/OS thread of execution, you must set the server **`Connection Manager RunAs Identity Enabled`** property to **`true`**.

   Note: With Bean Managed Persistence (BMP) beans, if you obtain a connection under the `ejbLoad()` or `ejbStore()` functions during pre-invoke or post-invoke method processing, your thread identity support does not become the **RunAs** identity because the container during this processing is running under server identity. See the topic Delegations for more information. With BMP beans, instead of using thread identity, specify a Container-managed alias to associate the user with the connection.

Security states with thread identity support

Different Java Platform, Enterprise Edition Connector Architecture (JCA) resource adapters and Java Database Connectivity (JDBC) drivers provide different support for authenticating threads that transact with application server resources.

In this article the term thread identity refers to the Java Platform, Enterprise Edition (Java EE) Identity, such as the RunAs Identity, as opposed to the OS thread identity. Refer to the topic, Synchronizing a Java thread identity and an operating system thread identity and the topic, Understanding Connection Manager RunAs Identity Enabled and operating system security, for more information.

The combinations of Java 2 security, server configurations, connector configurations, and container-managed alias support determine the processing that results when you use the thread identity function. Thread identity support is only available with specific JCA resource adapters and JDBC providers. See the article Connection thread identity for a table of resource adapter processes and JDBC provider processes that support thread identity. If your resource adapter or JDBC provider is in the supported list, use the following tables to determine the processing that occurs, based on the settings of the specified properties:

Table 4. Security state. Check your security state, and go to table 2 or 3.

Global security enabled?	
Yes	No
Go to table 2.	Go to table 3.

Table 5. Global security enabled. When your global security is enabled, use the following table to determine the processing that occurs.

<table>
<tr><th colspan="8">Container-managed alias specified?</th></tr>
<tr><td colspan="4">No</td><td colspan="4">Yes</td></tr>
<tr><td colspan="4">Connector Allows or Requires Thread Identity?</td><td colspan="4">Connector Requires Thread Identity?</td></tr>
<tr><td>No</td><td colspan="3">Yes</td><td>No</td><td colspan="3">Yes</td></tr>
<tr><td rowspan="5">Processing is dependent on connector: may throw exception may default to connector user/ password custom properties</td><td colspan="3">Connector requires OS thread security?</td><td rowspan="5">Use specified alias</td><td colspan="3">Connector requires OS thread security?</td></tr>
<tr><td>No</td><td colspan="2">Yes</td><td>No</td><td colspan="2">Yes</td></tr>
<tr><td rowspan="3">Use identity associated with current thread</td><td colspan="2">Server Sync-To-Thread enabled?</td><td rowspan="3">Use identity associated with current thread</td><td colspan="2">Server Sync-To-Thread enabled?</td></tr>
<tr><td>No</td><td>Yes</td><td>No</td><td>Yes</td></tr>
<tr><td>Use Server identity</td><td>Use identity associated with current thread</td><td>Use server identity</td><td>Use identity associated with current thread</td></tr>
</table>

Table 6. Global Security is not enabled. When your global security is not enabled, use the following table to determine the processing that occurs.

<table>
<tr><th colspan="4">Container-managed alias specified?</th></tr>
<tr><td colspan="2">No</td><td colspan="2">Yes</td></tr>
<tr><td colspan="2">Connector ALLOWS or REQUIRES thread identity o be used when getting a connection</td><td colspan="2">Connector REQUIRES thread identity to be used when getting a connection?</td></tr>
<tr><td>No</td><td>Yes</td><td>No</td><td>Yes</td></tr>
</table>

Table 6. Global Security is not enabled (continued). When your global security is not enabled, use the following table to determine the processing that occurs.

Container-managed alias specified?			
Processing is dependent on connector: • May throw exception • May default to connector user/password custom properties	User server identity	Use specified alias	Use server identity

Securing optimized local adapters

Security considerations using optimized local adapters with IMS

This topic reviews considerations for security using optimized local adapters with IMS.

Optimized local adapters APIs can be used in the following IMS-dependent region environments:

- MPR (MPPs)
- Fast Path (IFPs)
- Batch Message Processing (BMPs)
- Batch DL/I

The registration process requires that the user ID on the current thread, or TCB in the dependent region, be authorized, or at least have READ access, to the System Authorization Authority (SAF) CBIND class for the target WebSphere Application Server server. Registration is required before you can send any other requests to WebSphere Application Server.

There are several ways that the user identity is associated with the current IMS task and its TCB. For BMPs, the job user ID is the identity that requires access to the CBIND class. For IFPs and MPPs, the user identity on the TCB can be set another way. If the SECURITY macro for the IMS environment specifies SECLVL=(TRANAUTH,SIGNON), the user ID that is provided at sign-on is required to be in the local SAF database and SAF authentication occurs. In addition, transaction access is checked with SAF.

Running with these options, and using the “Build Security Environment”, exit DFSBSEX0 passes back a return code 4 to IMS. Then, IMS ensures that the TCB that the transaction it is dispatched under is synchronized with the SAF ID that was authenticated.

The user ID of the application user requires READ access to the WebSphere Application Server CBIND SAF class for a successful optimized local adapters Register API call. IMS transactions that are initiated from callers using the Open Transaction Manager Access (OTMA) protocol, use the OTMASE parameter to determine if the current thread/TCB security context is updated. Setting the OTMASE parameter to `OTMASE=FULL`, indicates that the identity passed in by the OTMA client call is the identity on the thread of the MPP or IFP. In this scenario, the client ID requires READ access to the CBIND class.

When transaction work passes from IMS to WebSphere Application Server for z/OS, the user ID is propagated into the WebSphere Application Server EJB container and asserted.

When using the optimized local adapters to call existing unchanged IMS transactions over OTMA, the identity of the current WebSphere Application Server client can be propagated to IMS transactions implemented and asserted in Message Processing (MPR) and Fast Path (IFP) dependent regions. To do this, ensure that the WebSphere server is configured to run with the SyncToOS Thread option enabled. To read more about activating the SyncToOS Thread option, see the topic, z/OS security options. Once SyncToOS Thread is enabled, ensure that the OTMASE parameter for the target IMS environment is set to

F, FULL. With these options configured this way, the identity of the user in the WebSphere Application Server environment is propagated to an IMS MPP or IFP and asserted. This does not apply to Batch Message Processing (BMP) dependent regions.

gotcha: You must configure a BBO.SYNC profile if you are using SAF. Refer to the topic *System Authorization Facility classes and profiles* for a description of how to configure a BBO.SYNC profile.

Securing optimized local adapters for inbound support

Use this task to set up security for your optimized local adapters connections that perform inbound calls.

Before you begin

Run the WebSphere Application Server for z/OS servers with global security and activate the Sync-to-OS Thread option if you intend to use the optimized local adapter APIs with those servers. To read about global security, see the topic, Enabling security. To read more about activating the Sync-to-OS Thread option, see the topic, z/OS security options.

Local access to WebSphere Application Server for z/OS servers is protected by the System Authorization Facility (SAF) CBIND class. This class is defined during profile creation and is used to protect WebSphere Application Server for z/OS servers when Internet Inter-ORB Protocol (IIOP) local client connection requests are made, and optimized local adapters requests. Before running any application that uses the Register API, be sure to grant READ access for the user ID for the job, UNIX System Services (USS) process, or Customer Information Control System (CICS) region to the CBIND class for the target server. This is setup with the BBOCBRAK job. For more information about the CBIND class, read the topic, Using CBIND to control access to clusters.

All inbound requests to WebSphere Application Server run under the authority of the current user on thread. This identity is automatically propagated and is asserted in the Enterprise JavaBeans (EJB) container and this identity is that which the application starts under. Inbound requests that drive into a target enterprise bean arrive in the same manner as method invocations do for local IIOP requests and the security options for RunAs work in the same way as local IIOP requests

When transaction work passes between CICS and WebSphere Application Server for z/OS, either inbound or outbound, you must take into account some special security considerations. For example, you must determine if the authentication for inbound to WebSphere Application Server work should run with the authority of the specific CICS application or the overall CICS region authority. There are similar concerns when WebSphere Application Server sends outbound work to a CICS application; you must determine if CICS should honor the originating application authority or its own CICS current security profile.

Attention: You must make sure that the client applications are authenticated in order for CICS to process the request.

For passing requests in to WebSphere Application Server from CICS, you can indicate that you want to use the current CICS application identity by setting a flag for this with the Register API call.

About this task

The following steps include the tasks that you must complete to secure the optimized local adapters for an inbound call:

Procedure

Configure the security settings. The security identity propagation type is specified in registration flags when the optimized local adapters connection request is made in the call to BBOA1REG. You can select either the CICS region or application security profile.

Attention: In order for this to function properly, you need to have enabled CICS application level security with the SEC=YES CICS startup option.
Also, the CICS application user can only be propagated and asserted on the WebSphere Application Server thread when the ola_cicsuser_identity_propagate environment variable is set to 1. This gives control of this behavior to be managed by the WebSphere Application Server system programmer. When this option is set to 0 (the default), and the Register API calls CICS applications that select application level propagation, an error occurs. For more information about this environment variable see the topic, Optimized local adapters environment variables.
Set the environment variable to permit the CICS application-level identities to be used for authentication when the registration request is made. You can set the variable in the administrative console as follows:

1. Click **Environment** > **WebSphere Variables**.
2. Under **Scope**, select Cell from the Show scope selection drop-down list. If the ola_cicsuser_identity_propagate environment variable displays in the resources list, you do not have to add it again. You can continue with step c. If you have not added the variable to the resource list, you must Click **Add**. The ola_cicsuser_identity_propagate environment variable must be added to the display list the first time you do this task. Each time after the initial addition, you are able to select ola_cicsuser_identity_propagate from the display list after you set the scope.
3. Click ola_cicsuser_identity_propagate A window displays the General Properties where you can configure the variable.
4. Set the WebSphere Application Server environment variable to `1`. If you set the environment variable to `0` (zero) or leave it undefined, the CICS application level security is not honored in an inbound call to WebSphere Application Server.
5. Click **Apply** and **OK**.

Results

You have set up security for the optimized local adapters inbound connections.

What to do next

For more information about using security with IMS, see the topic, Security considerations when using optimized local adapters with IMS.

Securing optimized local adapters for outbound support

Use this task when you want to set up security for your optimized local adapters that perform outbound calls.

Before you begin

Run the WebSphere Application Server for z/OS servers with global security and activate the Sync-to-OS Thread option if you intend to use the optimized local adapter APIs with those servers. To read about global security, see the topic, Enabling security. To read more about activating the Sync-to-OS Thread option, see the topic, z/OS security options.

Alternatively, the system administrator can provide a username and password on the optimized local adapters connection factory, or the application developer can provide a username and password on the ConnectionSpec object, which is used to obtain a connection from the optimized local adapters connection factory. A login is performed using this username and password combination, and the MVS™ user ID

associated with the username is used when making optimized local adapters requests from this connection. If there is no MVS user ID associated with this username, then an MVS user ID is not used when making optimized local adapters requests from this connection.

Local access to WebSphere Application Server for z/OS servers is protected by the System Authorization Facility (SAF) CBIND class. This class is defined during profile creation and is used to protect WebSphere Application Server for z/OS servers when Internet Inter-ORB Protocol (IIOP) local client connection requests are made, and optimized local adapters requests. Before running any application that uses the Register API, be sure to grant READ access for the user ID for the job, UNIX System Services (USS) process, or Customer Information Control System (CICS) region to the CBIND class for the target server. this is set up with the BBOCBRAK job. For more information about the CBIND class, read the topic, Using CBIND to control access to clusters.

For calling from WebSphere Application Server to an application using either the optimized local adapters Host Service and Receive Request APIs, the identity on the thread that the API was called on is used. For environments other than CICS, there is no attempt by the optimized local adapters to assert the WebSphere Application Server application identity. This includes Information Management System (IMS) dependent regions. For these, transactions start under the ID of the user that started the transaction. This includes IMS dependent regions. For these regions, transactions start under the user ID that started the transaction.

When transaction work passes between CICS and WebSphere Application Server for z/OS, either inbound or outbound, you must take into account some special security considerations. For example, you need determine if the authentication for inbound to WebSphere Application Server work should run with the authority of the specific CICS application or the overall CICS region authority. There are similar concerns when WebSphere Application Server sends outbound work to a CICS application; you need to determine if CICS should honor the originating application authority or its own CICS current security profile.

Attention: Ensure that the client applications are authenticated in order for CICS to process the request.

For receiving requests in CICS and processing them with the optimized local adapter CICS Link server (BBO$ task), you can indicate when you start the Link server that you want to have Link server assert the propagated WebSphere Application Server thread-level identity to the CICS thread where the target program starts. This is done with a parameter on the optimized local adapters BBOC CICS transaction.

About this task

The following steps include the tasks that you must complete to secure the optimized local adapters for an outbound call:

Procedure

Configure the security settings. When using the optimized local adapters Host Service or Receive Request APIs in an application that is running under CICS, the authority of the CICS application that called these APIs is used. When using the optimized local adapters CICS Link server, you can indicate that you want the Link server task, BBO$, to assert the WebSphere Application Server identity before calling the target program as follows:

1. On the optimized local adapters BBOC CICS transaction that you are using to start the Link server (with BBOC START_SRVR), pass the SEC=Y parameter. When this is specified, the optimized local adapters Link server task, BBO$, starts the link task, BBO#, with the identity that was propagated from calling the WebSphere Application Server thread.
2. Ensure that the CICS region is running with security enabled and EXEC CICS START checking enabled. Security is enabled at start up with the parameter SEC=YES. The EXEC CICS START checking is enabled at start up with the parameter XUSER=YES.

3. Create a SAF surrogate class that grants the identity that the optimized local adapters Link server is running with the authority to issue EXEC CICS START TRANSACTION API and pass the USERID that was propagated to CICS from WebSphere Application Server. The following is a sample that shows a surrogate class defined for user ID USER1 that allows user ID OLASERVE to issue EXEC CICS START TRANS(BBO#) USERID(USER1) and process optimized local adapters CICS Link transactions that run with the identity of USER1.

```
RDEFINE SURROGAT USER1.DFHSTART UACC(NONE) OWNER(USER1)
PERMIT USER1.DFHSTART CLASS(SURROGAT) ID(USER1)
PERMIT USER1.DFHSTART CLASS(SURROGAT) ID(OLASERVE)
SETROPTS RACLIST(SURROGAT) REFRESH
```

Results

You have set up security for the optimized local adapters connections.

What to do next

For more information about using security with IMS, see the topic, Security considerations when using optimized local adapters with IMS.

Chapter 3. Securing EJB applications

This page provides a starting point for finding information about enterprise beans.

Based on the Enterprise JavaBeans (EJB) specification, enterprise beans are Java components that typically implement the business logic of Java 2 Platform, Enterprise Edition (J2EE) applications as well as access data.

Securing Enterprise JavaBeans applications

Securing enterprise bean applications

You can protect enterprise bean methods by assigning security roles to them. Before you assign security roles, you need to know which Enterprise JavaBeans (EJB) methods need protecting and how to protect them.

About this task

You can assign a set of EJB methods to a set of roles. When an EJB method is secured by associating a set of roles, grant at least one role in that set so that you can access that method. To exclude a set of EJB methods from access, mark the set **excluded**. You can give everyone access to a set of enterprise beans methods by clearing those methods. You can run enterprise beans as a different identity, using the runAs identity, before invoking other enterprise beans.

Note: This procedure might not match the steps that are required when using your assembly tool, or match the version of the assembly tool that you are using. You should follow the instructions for the tool and version that you are using. For more information about using assembly tools see the assembly tool information center.

To secure enterprise bean applications, follow these steps:

Procedure

1. In an assembly tool, import your Enterprise JavaBeans (EJB) Java Archive (JAR) file or an application archive (EAR) file that contains one or more web modules.

   See the information about importing an EJB JAR file or importing an enterprise application EAR file in the Rational Application Developer documentation.
2. In the Project Explorer, click **EJB Projects** directory and click the name of your application.
3. Right-click the deployment descriptor and click **Open with > Deployment Descriptor Editor**. If you selected an enterprise bean `.jar` file, an EJB deployment descriptor editor opens. If you select an application `.ear` file, an application deployment descriptor editor opens. To see online information about the editor, press **F1** and click the editor name.
4. Create security roles. You can create security roles at the application level or at the EJB module level. If you create a security role at the EJB module level, the role displays in the application level. If a security role is created at the application level, the role does not display in all the EJB modules. You can copy and paste one or more EJB module security roles that you create at application level:
   - Create a role at an EJB module level. In an EJB deployment descriptor editor, click the **Assembly** tab. Under Security Roles, click **Add**. In the Add Security Role wizard, name and describe the security role and click **Finish**.
   - Create a role at the application level. In an application deployment descriptor editor, select the **Security** tab. Under the list of security roles, click **Add**. In the Add Security Role wizard, name and describe the security role; then click **Finish**.
5. Create method permissions. Method permissions map one or more methods to a set of roles. An enterprise bean has four types of methods: home methods, remote methods, LocalHome methods and

© IBM Corporation 2009

local methods. You can add permissions to enterprise beans on the method level. You cannot add a method permission to an enterprise bean unless you already have one or more security roles defined. For Version 2.0 EJB projects, an unselected option specifies that the selected methods from the selected beans do not require authorization to run. To add a method permission to an enterprise bean:

a. On the **Assembly** tab of an EJB deployment descriptor editor, under **Method Permissions**, click **Add**. The Add Method Permission wizard is opened.
b. Select a security role from the list of roles found and click **Next**.
c. Select one or more enterprise beans from the list of beans found. You can click **Select All** or **Deselect All** to select or clear all of the enterprise beans in the list. Click **Next**.
d. Select the methods that you want to bind to your security role. The Method elements page lists all the methods that are associated with the enterprise beans. You can click **Apply to All** or **Deselect All** to quickly select or clear multiple methods. The selection affects the default (*) method for each bean only. Creating a method permission for the exact method signature overrides the default (*) method permission setting. The default (*) method represents all the methods within the bean. There are default (*) methods for each interface as well. By not selecting all of the individual methods in the tree, you can set other permissions on the remaining methods.
e. Click **Finish**.

After the method permission is created, you can see the new method permission in the tree. Expand the tree to see the bean and the methods that are defined in the method permission.

6. Exclude user access to methods. Users cannot access excluded methods. Any method in the enterprise beans that is not assigned to a role or that is not excluded, is cleared during the application installation by the deployer.
   a. On the **Assembly** tab of an EJB deployment descriptor editor, under **Excludes List**, click **Add**. The Exclude List wizard is opened.
   b. Select one or more enterprise beans from the list of beans found and click **Next**.
   c. Select one or more of the method elements for the security identity and click **Finish**.
7. Map the security-role-ref and role-name to the role-link. When developing enterprise beans, you can create the security-role-ref element. The security-role-ref element contains only the role-name field. The role-name field determines if the caller is in a specified role(isCallerInRole()) role and contains the name of the role that is referenced in the code. Because you create security roles during the assembly stage, the developer uses a *logical role name* in the **role-name** field and provides enough information in the **Description** field for the assembler to map the actual role (role-link). The security-role-ref element is located at the EJB level. Enterprise beans can have zero or more security-role-ref elements.
   a. On the **Reference** tab of an EJB deployment descriptor editor, under the list of references, click **Add**. The Add Reference wizard is opened.
   b. Select **Security role reference** and click **Next**.
   c. Name the security role reference, select a security role to link the reference to, describe the security role reference, and click **Finish**.
   d. Map every role-name that is used during development to the role (role-link) using the previous steps.
8. Specify the RunAs identity for enterprise bean components. The RunAs identity of the enterprise bean is used to invoke the next enterprise beans in the chain of EJB invocations. When the next enterprise beans are invoked, the RunAsIdentity identity passes to the next enterprise beans for performing an authorization check on the next enterprise bean. If the RunAs identity is not specified, the client identity is propagated to the next enterprise bean. The RunAs identity can represent each of the enterprise beans or can represent each method in the enterprise beans.
   a. On the **Access** tab of an EJB deployment descriptor editor, next to the **Security Identity (Bean Level)** field, click **Add**. The Add Security Identity wizard is opened.
   b. Select the appropriate run as mode, describe the security identity, and click **Next**. Select the **Use identity of caller** mode to instruct the security service to not make changes to the credential settings for the principal. Select the **Use identity assigned to specific role (below)** mode to use

a principal that is assigned to the specified security role for running the bean methods. This association is part of the application binding in which the role is associated with the user ID and password of a user who is granted that role. If you select the **Use identity assigned to specific role (below)** mode , you must specify a role name and role description.

c. Select one or more enterprise beans from the list of beans found and click **Next**. If Next is unavailable, click **Finish**.

d. Optional: On the Method elements page, select one or more of the method elements for the security identity and click **Finish**.

9. Close the deployment descriptor editor and, when prompted, click **Yes** to save the changes.

Results

After securing an EJB application, the resulting `.jar` file contains security information in its deployment descriptor. The security information of the EJB modules is stored in the `ejb-jar.xml` file.

What to do next

After securing an EJB application using an assembly tool, you can install the EJB application using the administrative console. During the installation of a secured EJB application, follow the steps in the topic, Deploying secured applications, to complete the task of securing the EJB application.

Chapter 4. Securing Messaging resources

This page provides a starting point for finding information about the use of asynchronous messaging resources for enterprise applications with WebSphere Application Server.

WebSphere Application Server supports asynchronous messaging based on the Java Message Service (JMS) and the Java EE Connector Architecture (JCA) specifications, which provide a common way for Java programs (clients and Java EE applications) to create, send, receive, and read asynchronous requests, as messages.

JMS support enables applications to exchange messages asynchronously with other JMS clients by using JMS destinations (queues or topics). Some messaging providers also allow WebSphere Application Server applications to use JMS support to exchange messages asynchronously with non-JMS applications; for example, WebSphere Application Server applications often need to exchange messages with WebSphere MQ applications. Applications can explicitly poll for messages from JMS destinations, or they can use message-driven beans to automatically retrieve messages from JMS destinations without explicitly polling for messages.

WebSphere Application Server supports the following messaging providers:

- The WebSphere Application Server default messaging provider (which uses service integration as the provider).
- The WebSphere MQ messaging provider (which uses your WebSphere MQ system as the provider).
- Third-party messaging providers that implement either a JCA Version 1.5 resource adapter or the ASF component of the JMS Version 1.0.2 specification.

Securing messaging

The steps to take to secure asynchronous messaging.

About this task

Security for messaging is enabled only when WebSphere Application Server administrative security is enabled. In this case:

- JMS connections made to a messaging provider are authenticated.
- Access to JMS resources owned by a messaging provider is controlled by access authorizations.
- Requests to create new connections to a messaging provider must include a user ID and password for authentication.
- The user ID and password do not have to be provided by the application.

Standard Java EE Connector Architecture (JCA) authentication is used for a request to create a new connection to a messaging provider. If authentication is successful, the JMS connection is created; if authentication fails, the connection request is ended.

Notes:

- User IDs that are longer than 12 characters cannot be used for authentication with a WebSphere MQ network. For example, the default Windows NT user ID “Administrator” is not valid for use in this context because it contains 13 characters.
- Users that exploit the connection thread identity support do not have to provide a user ID and password for authentication.
- In addition to the authorization needed for creating a connection to a messaging provider, you also typically need authorization to access specific JMS resources associated with that provider.

© IBM Corporation 2009

For example, if you are using the WebSphere MQ messaging provider to connect to a WebSphere MQ network, you might also need permission from the WebSphere MQ network to write to a given queue.

- To enable the WebSphere MQ messaging provider to connect in bindings transport mode to WebSphere MQ, you set the**Transport type** parameter on the WebSphere MQ queue connection factory to `BINDINGS`, and you configure the WebSphere MQ messaging provider with native libraries information.

To secure your asynchronous messaging, complete one or more of the following steps:

Procedure

- Enable security.
- Use JCA authentication to create a new connection to the messaging provider.

  If the resource authentication (**res-auth**) property is set to `Application`, set the **Component-managed Authentication Alias** property on the connection factory. If the application that tries to create a connection to the messaging provider specifies a user ID and password, those values are then used to authenticate the creation request. Otherwise, the values defined by the **Component-managed Authentication Alias** property are used. If you do not set the **Component-managed Authentication Alias** property on the connection factory, a runtime JMS exception message is generated when an attempt is made to connect to the messaging provider.

  If the **res-auth** property is set to `Container`, set the **Container-managed Authentication Alias** property on the connection factory, and specify the user ID and password within this alias. If you are using bindings transport mode, then you can use the connection thread identity support instead of specifying a container-managed alias.
- Authorize access to messages stored by the default messaging provider.

  Access to these messages is controlled by authorization to access the service integration bus destinations on which the messages are stored. For information about authorizing permissions for individual bus destinations, see “Administering destination roles” on page 62.
- Configure security for message-driven beans that use listener ports

  Complete this step if you are working with a message-driven bean and are configuring a message-driven bean listener under the Message Listener Service.

Configuring security for message-driven beans that use activation specifications

Use this task to configure resource security and security permissions for message-driven beans.

About this task

Messages handled by message-driven beans have no client credentials associated with them. The messages are anonymous.

To call secure enterprise beans from a message-driven bean, the message-driven bean needs to be configured with a RunAs Identity deployment descriptor. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component.

For more information about EJB security, see EJB component security. For more information about configuring security for your application, see Assembling secured applications.

Connections used by message-driven beans can benefit from the added security of using JCA container-managed authentication. To enable the use of JCA container authentication aliases and mapping, define an authentication alias on the activation specification that the message-driven bean is configured with. If defined, the message-driven bean uses the authentication alias for its JMSConnection security credentials instead of any application-managed alias.

To set the authentication alias, you can use the administrative console to complete the following steps. This task description assumes that you have already created an activation specification. If you want to create a new activation specification, see the related tasks.

Procedure

- For a message-driven bean listening on a JMS destination of the default messaging provider, set the authentication alias on an activation specification.
  1. To display a list of existing JMS activation specifications, navigate to the following administrative console collection panel: **Resources** > **JMS->Activation specifications**
  2. If you have already created a JMS activation specification, click its name in the list displayed. Otherwise, click **New** to create a new JMS activation specification.
  3. Set the **Authentication alias** property.
  4. Click **OK**
  5. Save your changes to the master configuration.
- For a message-driven bean listening on a destination (or endpoint) of third-party JCA 1.5-enabled provider, set the authentication alias on an activation specification.
  1. To display the activation specification settings, click **Resources** > **Resource Adapters** > **J2C activation specifications** > ***activation_specification_name***
  2. Set the **Authentication alias** property.
  3. Click **OK**
  4. Save your changes to the master configuration.

Configuring security for message-driven beans that use listener ports

For non-Java EE Connector Architecture (JCA) messaging providers, the association between connection factories, destinations, and message-driven beans is provided by listener ports. In this case, you can configure resource security and security permissions for message-driven beans by setting the container-managed alias. The MDB listener's security information is established when the MDB listener's JMS Connection is created.

Before you begin

A listener port allows a deployed message-driven bean associated with the port to retrieve messages from the associated destination. For more information about listener ports, see Message-driven beans - listener port components.

Note: For WebSphere Application Server Version 7 and later, listener ports are stabilized. For more information, read the article on stabilized features. You should plan to migrate your WebSphere MQ message-driven bean deployment configurations from using listener ports to using activation specifications. However, you should not begin this migration until you are sure the application does not have to work on application servers earlier than WebSphere Application Server Version 7. For example, if you have an application server cluster with some members at Version 6.1 and some at a later version, you should not migrate applications on that cluster to use activation specifications until after you migrate all the application servers in the cluster to the later version.

About this task

In most respects, the security for an MDB is identical to security for any other enterprise bean. For instance, access to JDBC resources and JCA resources (for example CICS, IMS) is handled in the same way as for an entity or session bean. Access to other JMS resources is also handled in the same way as for other enterprise beans.

To secure an MDB which has been deployed on a listener port, you configure authentication and authorization for the server to connect to a JMS provider and a destination so that a message can be retrieved from the destination for processing by the `onMessage()` method of the MDB.

With some MDBs, the `onMessage()` method attempts to access additional JMS resources after the initial JMS connection has been made. In this case, security is handled identically to JMS calls made by an entity or session EJB.

The security information for an MDB which has been deployed onto a listener port is required when the initial JMS connection is created. When an MDB is deployed on a listener port, the security information for the MDB is determined by the values specified for the connection factory which the listener port is using. The user ID that is used by the listener port to create the JMS connection, is determined by the type of authentication alias which has been specified on the queue connection factory:

1. If a container-managed alias has been defined for the connection factory, the user ID associated with the container-managed alias is used in the connection creation call, for example createQueueConnection(userid,password).
2. If a component-managed alias has been defined for this connection factory, the user ID associated with the component-managed alias is used for the connection creation call.
3. If neither alias is specified and the connection factory is defined in bindings mode (that is, `TransportType = "BINDINGS"`), the server identity is used. The server identity translates more specifically into the servant identity in the servants, and the controller identity in the controller. Therefore, for a listening-in controller, the controller identity is relevant and the servant identity is relevant. For related information about listening-in controllers, see Message listener service on z/OS.

Note: The authentication aliases referred to here are the authentication aliases associated with the connection factory defined by the administrator. No application resource reference is associated with the MDB or the listener port, therefore no authentication alias must be set at that level.

To set the container-managed alias (if you elect that option), use the administrative console to complete the following steps:

Procedure

1. Display the listener port settings, by clicking **Servers** > **Server Types** > **WebSphere application servers** > ***application_server*** > **[Communications] Messaging** > **Message listener service** > **[Additional properties] Listener ports** > ***listener_port***
2. Get the name of the JMS connection factory, by looking at the connection factory JNDI name property.
3. Display the JMS connection factory properties. For example, to display the properties of a queue connection factory, click **Resources** > **JMS->Queue connection factories->*queue_connection_factory***.
4. Set the “Container-managed authentication alias” property.
5. Click **OK**

What to do next

Invoking other EJBs

Messages arriving at a listener port have no client credentials associated with them. The messages are anonymous. To call secure enterprise beans from a message-driven bean, the message-driven bean must be configured with a RunAs Identity deployment descriptor. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component.

For more information about EJB security, see “Securing enterprise bean applications” on page 27. For more information about configuring security for your application, see “Securing applications during assembly and deployment” on page 118.

Chapter 5. Securing Mail, URLs, and other Java EE resources

This page provides a starting point for finding information about resources that are used by applications that are deployed on a Java Enterprise Edition (Java EE)-compliant application server. They include:

- JavaMail support for applications to send Internet mail
- URLs, for describing logical locations
- Resource environment entries, for mapping logical names to physical names
- Java DataBase Connectivity (JDBC) resources and other technology for data access (discussed elsewhere)
- Java Message Service (JMS) resources and other messaging system support (discussed elsewhere)

Securing applications that use the JavaMail API

JavaMail API security permissions best practices

In many of its activities, the JavaMail API needs to access certain configuration files. The JavaMail and JavaBeans Activation Framework binary packages themselves already contain the necessary configuration files. However, the JavaMail API allows the user to define user-specific and installation-specific configuration files to meet special requirements.

The two locations where you can place these configuration files are the `<user.home>` and `<java.home>/lib` directories. For example, if the JavaMail API needs to access a file named mailcap when it sends a message, the API:

1. Tries to access `<user.home>/mailcap`.
2. If the first attempt fails due to a lack of security permission or a nonexistent file, the API searches in `<java.home>/lib/mailcap`.
3. If the second attempt also fails, the API searches in the `META-INF/mailcap` location in the class path. This location actually leads to the configuration files contained in the mail-impl.jar and activation-impl.jar files.

Application Server uses JavaMail API configuration files that are contained in the mail-impl.jar and activation-impl.jar files, and there are no mail configuration files in `<user.home>` and `<java.home>/lib` directories. To ensure proper functioning of the JavaMail API, Application Server grants *file read* permission for both the mail-impl.jar and activation-impl.jar files to all of the installed applications.

JavaMail code attempts to access configuration files at `<user.home>` and `<java.home>/lib`, which can cause an access control exception to be thrown, since the default configuration does not grant file read permission for those two locations by default. This activity does not affect the proper functioning of the JavaMail API, but you might see a large amount of mail-related security exceptions reported in the system log, and these errors could overshadow harmful errors for which you are looking. This is a sample of the security message, SECJ0314W:

```
[02/31/08 12:55:38:188 PDT] 00000058 SecurityManag W   SECJ0314W: Current Java 2 Security policy reported a
potential violation of Java 2 Security Permission.
Please refer to Problem Determination Guide for further information.

Permission:

      D:\o063919\java\jre\lib\javamail.providers : access denied (java.io.FilePermission
D:\o063919\java\jre\lib\javamail.providers read)


Code:

     com.ibm.ws.mail.SessionFactory  in  {file:/D:/o063919/lib/runtime.jar}


Stack Trace:
```

```
java.security.AccessControlException: access denied (java.io.FilePermission D:\o063919\java\jre\lib\javamail.providers read)
 at java.security.AccessControlContext.checkPermission(AccessControlContext.java(Compiled Code))
 at java.security.AccessController.checkPermission(AccessController.java(Compiled Code))
 at java.lang.SecurityManager.checkPermission(SecurityManager.java(Compiled Code))
 at com.ibm.ws.security.core.SecurityManager.checkPermission(SecurityManager.java(Compiled Code))
 at java.lang.SecurityManager.checkRead(SecurityManager.java(Compiled Code))
 at java.io.FileInputStream.<init>(FileInputStream.java(Compiled Code))
 at java.io.FileInputStream.<init>(FileInputStream.java:89)
 at javax.mail.Session.loadFile(Session.java:1004)
 at javax.mail.Session.loadProviders(Session.java:861)
 at javax.mail.Session.<init>(Session.java:191)
 at javax.mail.Session.getInstance(Session.java:213)
 at com.ibm.ws.mail.SessionFactory.getObjectInstance(SessionFactory.java:67)
 at javax.naming.spi.NamingManager.getObjectInstance(NamingManager.java:314)
 at com.ibm.ws.naming.util.Helpers.processSerializedObjectForLookupExt(Helpers.java:894)
 at com.ibm.ws.naming.util.Helpers.processSerializedObjectForLookup(Helpers.java:701)
 at com.ibm.ws.naming.jndicos.CNContextImpl.processResolveResults(CNContextImpl.java:1937)
 at com.ibm.ws.naming.jndicos.CNContextImpl.doLookup(CNContextImpl.java:1792)
 at com.ibm.ws.naming.jndicos.CNContextImpl.doLookup(CNContextImpl.java:1707)
 at com.ibm.ws.naming.jndicos.CNContextImpl.lookupExt(CNContextImpl.java:1412)
 at com.ibm.ws.naming.jndicos.CNContextImpl.lookup(CNContextImpl.java:1290)
 at com.ibm.ws.naming.util.WsnInitCtx.lookup(WsnInitCtx.java:145)
 at javax.naming.InitialContext.lookup(InitialContext.java:361)
 at emailservice.com.onlinebank.bpel.EmailService20060907T224337EntityAbstractBase$JSE_6.
execute(EmailService20060907T224337EntityAbstractBase.java:32)
 at com.ibm.bpe.framework.ProcessBase6.executeJavaSnippet(ProcessBase6.java:256)
 at emailservice.com.onlinebank.bpel.EmailService20060907T224337EntityBase.invokeSnippet
(EmailService20060907T224337EntityBase.java:40)
```

Note: If this situation is a problem, consider adding more read access permissions for more locations. This should eliminate most, if not all, JavaMail-related harmless security exceptions from the log file.

The permissions required by JavaMail are as follows:

```
grant codeBase "file:${application}" {
  // Allow access to default configuration files
  permission java.io.FilePermission "${java.home}${/}jre${/}lib${/}javamail.address.map", "read";
  permission java.io.FilePermission "${java.home}${/}jre${/}lib${/}javamail.providers", "read";
  permission java.io.FilePermission "${java.home}${/}jre${/}lib${/}mailcap", "read";
  permission java.io.FilePermission "${java.home}${/}lib${/}javamail.address.map", "read";
  permission java.io.FilePermission "${java.home}${/}lib${/}javamail.providers", "read";
  permission java.io.FilePermission "${java.home}${/}lib${/}mailcap", "read";
  permission java.io.FilePermission "${user.home}${/}.mailcap", "read";
  permission java.io.FilePermission "${was.install.root}${/}lib${/}activation-impl.jar", "read";
  permission java.io.FilePermission "${was.install.root}${/}lib${/}mail-impl.jar", "read";
  permission java.io.FilePermission "${was.install.root}${/}plugins${/}com.ibm.ws.prereq.javamail.jar", "read";
  // If using an isolated mail provider,
  // add additional file read permissions for each jar defined
  // for the isolated mail provider
  // permission java.io.FilePermission "path${/}mail.jar, "read";

  // Allow connection to mail server using SMTP
  permission java.net.SocketPermission "*:25", "connect,resolve";
  // Allow connection to mail server using SMTPS
  permission java.net.SocketPermission "*:465", "connect,resolve";

  // Allow connection to mail server using IMAP
  permission java.net.SocketPermission "*:143", "connect,resolve";
  // Allow connection to mail server using IMAPS
  permission java.net.SocketPermission "*:993", "connect,resolve";

  // Allow connection to mail server using POP3
  permission java.net.SocketPermission "*:110", "connect,resolve";
  // Allow connection to mail server using POP3S
  permission java.net.SocketPermission "*:995", "connect,resolve";

  // Allow System.getProperties() to be used
  // permission java.util.PropertyPermission "*", "read,write";
  // Otherwise use the following to allow system properties to be read
  permission java.util.PropertyPermission "*", "read";
};
```

Chapter 6. Securing OSGi applications

Securing OSGi applications is very similar to securing enterprise applications. For most security frameworks, no additional steps are required. For Java 2 security, there is some optional extra configuration that is specific to OSGi Applications.

About this task

For most security frameworks that are supported by WebSphere Application Server, configuring security for OSGi applications requires no additional steps to those that are required for enterprise applications. For example: If you enable security, and you add a secure asset, you must specify a target server that is in the global security domain. This requirement is the same whether the asset is an enterprise application or an OSGi application.

For application security with OSGi applications, you can modify the security role to user or group mapping when you add the asset to the business-level application.

For Java 2 security in enterprise applications, you set permissions at the application level. For OSGi applications, you can also set Java 2 security permissions at the bundle level. To support this finer-grained security, there are extra configuration steps that you can complete when you create an OSGi application, when you migrate an enterprise application to an OSGi application, and when you add an enterprise bundle archive (EBA) asset to a business-level application.

Procedure

- Use application security with OSGi applications.

  Application security controls which users may access which parts of the application. For more information, see Application security.

  - Modify the security role to user or group mapping.

    You can modify this mapping when you add the asset to the business-level application as a composition unit. For more information, see Adding an EBA asset to a composition unit by using the administrative console, Adding an EBA asset to a composition unit by using wsadmin commands, and Security role to user or group mapping [Settings].
  - Use application security with web application bundles (WABs).

    You secure WABs in the same way that you secure web applications in Java EE. Application security enforces any security constraints that are defined in the `web.xml` file for a WAB. When a web client tries to access a protected resource, the client is prompted for authentication.
  - Configure bean security in the Blueprint XML file.

    You can configure bean security in the Blueprint XML file of your OSGi applications, so that the methods of the bean can be accessed only by users that are assigned a specified role. You can configure bean-level security, so that a single role is associated with all the methods of the bean, or you can configure method-level security, where different roles are associated with specific methods.
  - Use application security with EJB bundles.

    You secure enterprise beans in EJB bundles in the same way that you secure enterprise beans in Java EE. Application security enforces any bean method security settings that are defined in the `ejb-jar.xml` file for an EJB bundle.
- Use Java 2 security with OSGi applications.

  Java 2 security controls access to protected system resources from the application.

  - Learn about using Java 2 security with OSGi applications.

    Using Java 2 security in OSGi applications is very similar to using Java 2 security in enterprise applications. For more information about Java 2 security in enterprise applications, see Java 2

© Copyright IBM Corp. 2012

security. For an overview of the main differences when you use Java 2 security in an OSGi application, see Java 2 security and OSGi Applications. This topic describes the following differences:

- The format and locations of the `permissions.perm` files in an OSGi application.
- The relationship between application-level `permissions.perm` files in OSGi applications and `was.policy` files in enterprise applications.
- The default permissions that apply to every OSGi application, in addition to any that are provided through a `permissions.perm` file.

– Configure Java 2 security for your OSGi application.

1. Create `permissions.perm` files. For more information, see Java 2 security and OSGi Applications.
2. Check the security permissions. The security permissions are displayed when you import the OSGi application as an asset. For more information, see Deploying an OSGi application as a business-level application.

– Migrate Java 2 security settings as part of migrating an enterprise application to an OSGi application.

When you convert an application from Java EE to OSGi, any existing `was.policy` file is converted into a `permissions.perm` file to be used with the OSGi permissions framework, and all permissions are promoted to the application level. If you need finer granularity, you can modify the file after conversion. For more information, see Java 2 security and OSGi Applications, and Converting an enterprise application to an OSGi application.

Chapter 7. Securing Portlet applications

This page provides a starting point for finding information about portlet applications, which are special reusable Java servlets that appear as defined regions on portal pages. Portlets provide access to many different applications, services, and web content.

Portlet URL security

Portlet URL security

WebSphere Application Server enables direct access to portlet Uniform Resource Locators (URLs), just like servlets. This section describes security considerations when accessing portlets using URLs.

For security purposes, portlets are treated similar to servlets. Most portlet security uses the underlying servlet security mechanism. However, portlet security information resides in the `portlet.xml` file, while the servlet and JavaServer Pages files reside in the `web.xml` file. Also, when you make access decisions for portlets, the security information, if any, in the `web.xml` file is combined with the security information in the `portlet.xml` file.

Portlet security must support both programmatic security, that is `isUserInRole`, and declarative security. The programmatic security is exactly the same as for servlets. However, for portlets, the `isUserInRole` method uses the information from the security-role-ref element in `portlet.xml`. The other two methods used by programmatic security, `getRemoteUser` and `getUserPrincipal`, behave the same way as they do when accessing a servlet. Both of these methods return the authenticated user information accessing the portlet.

The declarative security aspect of the portlets is defined by the security-constraint information in the `portlet.xml` file. This is similar to the security-constraint information used for the servlets in the `web.xml` file with the following differences:

- The auth-constraint element, which lists the names of the roles that can access the resources, does not exist in the `portlet.xml` file. The `portlet.xml` file contains only the user-data-constraint element, which indicates what type of transport layer security (HTTP or HTTPS) is required to access the portlet.
- The security-constraint information in the `portlet.xml` file contains the portlet-collection element, while the `web.xml` file contains the web-resource-collection element. The portlet-collection element contains only a list of simple portlet names, while the web-resource-collection contains the url-patterns as well as the HTTP methods that need protection.

The portlet container does not deal with the user authentication directly. For example, it does not prompt you to collect the credential information. The portlet container must, instead, use the underlying servlet container for the user authentication mechanism. As a result, there is no auth-constraint element in the security-constraint information in the `portlet.xml` file.

In WebSphere Application Server, when a portlet is accessed using a URL, the user authentication is processed based on the security-constraint information for that portlet in the `web.xml` file. This implies that to authenticate a user for a portlet, the web.xml file must contain the security-constraint information for that portlet with the relevant auth-constraints contained in it. If a corresponding auth-constraint for the portlet does not exist in the web.xml file, it indicates that the portlet is not required to have authentication. In this case, unauthenticated access is permitted just like a URL pattern for a servlet that does not contain any auth-constraints in the `web.xml` file. An auth-constraint for a portlet can be specified directly by using the portlet name in the url-pattern element, or indirectly by a url-pattern that implies the portlet.

Attention: You cannot have a servlet or JSP with the same name as a portlet for WebSphere Application Server security to work with portlet.

The following examples demonstrate how the security-constraint information contained in the portlet.xml and web.xml files in a portlet application are used to make security decisions for portlets. The security-role-ref element, which is used for isUserInRole calls, is not discussed here because it is used the same way for servlets.

In the examples later in this section (unless otherwise noted), there are four portlets (MyPortlet1, MyPortlet2, MyPortlet3, MyPortlet4) defined in portlet.xml. The portlets are secured by combining the information, if any, in the web.xml file when they are accessed directly through URLs.

All of the examples show the contents of the web.xml and portlet.xml files. Use the correct tools when creating these deployment descriptor files as you normally would when assembling a portlet application.

Example 1: The web.xml file does not contain any security-constraint data

In the following example, the security-constraint information is contained in portlet.xml:

```
<security-constraint>
   <display-name>Secure Portlets</display-name>
   <portlet-collection>
   <portlet-name>MyPortlet1</portlet-name>
   <portlet-name>MyPortlet3</portlet-name>
   </portlet-collection>
   <user-data-constraint>
   <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

In this example, when you access anything under MyPortlet1 and MyPortlet3, and these portlets are accessed using the unsecured HTTP protocol, you are redirected through the secure HTTPS protocol. The transport-guarantee is set to use secure connections. For MyPortlet2 and MyPortlet4, unsecured (HTTP) access is permitted because the transport-guarantee is not set. There is no corresponding security-constraint information for all four portlets in the web.xml file. Therefore, all of the portlets can be accessed without any user authentication and role authorization. The only security involved in this instance is the transport-layer security using Secure Sockets Layer (SSL) for MyPortlet1 and MyPortlet3.

.

Table 7. Security constraints that are applicable to the individual portlets. The following table lists the security constraints that are applicable to the individual portlets.

URL	Transport Protection	User Authentication	Role Based Authorization
/MyPortlet1/*	HTTPS	None	None
/MyPortlet2/*	None	None	None
/MyPortlet3/*	HTTPS	None	None
/MyPortlet4/*	None	None	None

Example 2: The web.xml file contains portlet specific security-constraint data

In the following example, the security-constraint information that corresponds to the portlet is contained in web.xml. The portlet.xml file is the same as that shown in the previous example.

```
<security-constraint id="SecurityConstraint_1">
    <web-resource-collection id="WebResourceCollection_1">
      <web-resource-name>Protected Area</web-resource-name>
      <url-pattern>/MyPortlet1/*</url-pattern>
      <url-pattern>/MyPortlet2/*</url-pattern>
      </web-resource-collection>
    <auth-constraint id="AuthConstraint_1">
      <role-name>Employee</role-name>
    </auth-constraint>
</security-constraint>
```

The security-constraint information contained in the web.xml file in this example indicates that the user authentication must be performed when accessing anything under the MyPortlet1 and MyPortlet2 portlets.

When you attempt to access these portlets directly using URLs, and there is no authentication information available, you are prompted to enter their credentials. After you are authenticated, the authorization check is performed to see if you are listed in the Employee role. The user/group to role mapping is assigned during the portlet application deployment. In the `web.xml` file previously listed, note the following:

- Because the `web.xml` file uses url-pattern, the portlet names have been modified slightly. MyPortlet1 is now /MyPortlet1/*, which indicates that everything under the MyPortlet1 URL is protected. This matches the information in the `portlet.xml` file because the security runtime code converts the portlet-name element in the `portlet.xml` file to url-pattern (for example, MyPortlet1 to /MyPortlet1/*), even for the transport-guarantee.
- The http-method element in the `web.xml` file is not used in the example because all HTTP methods must be protected.

Table 8. Security constraints that are applicable to the individual portlets. The following table lists the new security constraints that are applicable to the individual portlets.

URL	Transport Protection	User Authentication	Role Based Authorization
MyPortlet1/*	HTTPS	Yes	Yes (Employee)
MyPortlet2/*	None	Yes	Yes (Employee)
MyPortlet3/*	HTTPS	None	None
MyPortlet4/*	None	None	None

Example 3: The `web.xml` file contains generic security-constraint data implying all portlets.

In the following example, the security-constraint information is contained in the `web.xml` file that corresponds to the portlet. The `portlet.xml` file is the same as that shown in the first example.

```
<security-constraint id="SecurityConstraint_1">
    <web-resource-collection id="WebResourceCollection_1">
      <web-resource-name>Protected Area</web-resource-name>
      <url-pattern>/*</url-pattern>
      </web-resource-collection>
    <auth-constraint id="AuthConstraint_1">
      <role-name>Manager</role-name>
    </auth-constraint>
    </security-constraint>
```

In this example, /* implies that all resources that do not contain their own explicit security-constraints should be protected by the Manager role as per the URL pattern matching rules. Because the `portlet.xml` file contains explicit security-constraint information for MyPortlet1 and MyPortlet3, these two portlets are not protected by the Manager role, only by the HTTPS transport. Because the `portlet.xml` file cannot contain the auth-constraint information, any portlets that contain security-contraints in it are rendered unprotected when an implying URL (`/*` for example) is listed in the `web.xml` file because of the URL matching rules.

In the previous case, both MyPortlet1 and MyPortlet3 can be accessed without user authentication. However, because MyPortlet2 and MyPortlet4 do not have security-constraints in the `portlet.xml` file, the /* pattern is used to match these portlets and are protected by the Manager role, which requires user authentication.

Table 9. Security constraints that are applicable to the individual portlets. The following table lists the new security constraints that are applicable to the individual portlets with this setup.

URL	Transport Protection	User Authentication	Role Based Authorization
MyPortlet1/*	HTTPS	None	None
MyPortlet2/*	None	Yes	Yes (Manager)
MyPortlet3/*	HTTPS	None	None
MyPortlet4/*	None	Yes	Yes (Manager)

If in the previous example, if you must also protect a portlet contained in the `portlet.xml` file (for example, MyPortlet1), the `web.xml` file should contain an explicit security-constraint entry in addition to `/*` as shown in the following example:

```
<security-constraint id="SecurityConstraint_1">
         <web-resource-collection id="WebResourceCollection_1">
            <web-resource-name>Protected Area</web-resource-name>
            <url-pattern>/*</url-pattern>
            </web-resource-collection>
         <auth-constraint id="AuthConstraint_1">
            <role-name>Manager</role-name>
          </auth-constraint>
      </security-constraint>
      <security-constraint id="SecurityConstraint_2">
         <web-resource-collection id="WebResourceCollection_2">
            <web-resource-name>Protection for MyPortlet1</web-resource-name>
            <url-pattern>/MyPortlet1/*</url-pattern>
            </web-resource-collection>
         <auth-constraint id="AuthConstraint_1">
            <role-name>Manager</role-name>
          </auth-constraint>
      </security-constraint>
```

In this case, MyPortlet1 is protected by the Manager role and requires authentication. The data-constraint of `CONFIDENTIAL` is also applied to it because the information in the `web.xml` file and the `portlet.xml` file are combined. Because MyPortlet3 is not explicitly listed in the `web.xml` file, it is still not protected by the Manager role and does not require user authentication.

Table 10. Security constraints that are applicable to the individual portlets. The following table shows the effect of this change.

URL	Transport Protection	User Authentication	Role Based Authorization
MyPortlet1/*	HTTPS	Yes	Yes (Manager)
MyPortlet2/*	None	Yes	Yes (Manager)
MyPortlet3/*	HTTPS	None	None
MyPortlet4/*	None	Yes	Yes (Manager)

Chapter 8. Securing Service integration

This page provides a starting point for finding information about service integration.

Service integration provides asynchronous messaging services. In asynchronous messaging, producing applications do not send messages directly to consuming applications. Instead, they send messages to destinations. Consuming applications receive messages from these destinations. A producing application can send a message and then continue processing without waiting until a consuming application receives the message. If necessary, the destination stores the message until the consuming application is ready to receive it.

Securing service integration

Messaging security protects a service integration bus from unauthorized access. When administrative security is enabled for the application server, by default messaging security is also enabled for the bus. You can also manually administer messaging security for the bus.

Before you begin

Review the security requirements for the bus. For guidance, see Service integration security planning.

About this task

Providing administrative security is also enabled, messaging security enforces a security policy that prevents unauthorized client applications from connecting to the bus, and accessing bus resources. There might be circumstances when you do not require messaging security, for example on a development system. In this case, you can disable messaging security.

You can customize the security configuration for the bus by using the administrative console, or `wsadmin` scripting commands. The security configuration controls the following aspects of bus security:

- Authorizing groups of users in the user registry to undertake selected operations on bus destinations.
- The transport policies that maintain the integrity of messages in transit on the bus.
- The use of global, and multiple custom security domains.
- The integrity of links between messaging engines, foreign buses and databases.

Procedure

- “Securing buses” on page 44
- “Disabling bus security” on page 53
- “Enabling client SSL authentication” on page 53
- “Adding unique names to the bus authorization policy” on page 55
- “Administering authorization permissions” on page 56
- “Administering permitted transports for a bus” on page 82
- “Securing messages between messaging buses” on page 85
- “Securing access to a foreign bus” on page 86
- “Securing links between messaging engines” on page 86
- “Controlling which foreign buses can link to your bus” on page 87
- “Securing database access” on page 87
- “Securing mediations” on page 88

© Copyright IBM Corp. 2012

Securing buses

Securing a service integration bus provides the bus with an authorization policy to prevent unauthorized users from gaining access. If a bus is configured to use multiple security domains, the bus also has a security domain and user realm to further enforce its authorization policy.

Before you begin

- If administrative security is not enabled for the cell that hosts the bus, you must enable it. The tasks below use an administrative console wizard that detects if administrative security is not enabled, and takes you through the steps to enable it. You must supply the type of user repository used by the server, and the administrative security username and password.
- If the bus contains a bus member at WebSphere Application Server Version 6, you must provide an inter-engine authentication alias to establish trust between bus members, and to enable the bus to operate securely. The administrative console wizard detects whether an inter-engine authentication alias is required, and prompts you to supply one. If you want to specify a new inter-engine authentication alias, you must provide a user name and password.

About this task

When you secure a bus, consider the following points:

- If you are securing a bus that contains only Version 7.0 or later bus members, you can use a non-global security domain for the bus. If the bus has a WebSphere Application Server Version 6 bus member, or might have a Version 6 bus member in the future, you must assign the bus to the global security domain.
- If you want to assign the bus to a custom domain, you can select an existing security domain, or create a new one.
- If you assign the bus to a custom domain, you must specify a user realm. You can select an existing user realm, or use the global user realm.

What to do next

- The bus is secured after you restart all the servers that are members of the bus, or (for a bus that has bootstrap members) servers for which the SIB service is enabled.
- Use the administrative console to control access to the bus by administering users and groups in the bus connector role.

Adding a secured bus

In this task you add a new service integration bus that is secured by default. The security settings for the bus are stored in a security domain. When you add a new bus, you can assign it to the default global security domain, the cell-level domain, or specify a custom domain that contains a set of settings that are unique to the bus, or shared with another resource.

Before you begin

- Plan the security requirements for the bus. For more information about security planning, see Service integration security planning. For more information about security domains, see Messaging security and multiple security domains.
- Stop all servers that have the SIB Service enabled. This ensures that the bus security configuration is applied consistently when the servers are restarted. For more information, see Stopping an application server.

About this task

This task uses an administrative console security wizard to add a new bus. If the wizard detects that administrative security is disabled, it prompts you to configure a user repository, and enable administrative security.

By default, connecting clients are required to use SSL protected transports to ensure data confidentiality and integrity. If you do not want clients to use SSL protected transports, you can specify that you do not require this option.

The type of security domain you can specify for the bus depends on the versions of the bus members you intend to add to the bus:

- You must specify the global domain if you want to add one or more WebSphere Application Server Version 6 bus members.
- You can specify the global, cell-level, or custom domain if you want to add WebSphere Application Server Version 7.0 or later bus members only.

Procedure

1. In the navigation pane, click **Service integration -> Buses**. A list of buses is displayed.
2. Click **New**.
3. Type a name for the new bus. You must choose bus names that are compatible with the WebSphere MQ queue manager naming restrictions. You cannot change a bus name after the bus is created, which means that you can only interoperate with WebSphere MQ in the future if you use compatible names. See the topic about WebSphere MQ naming restrictions in the related links.
4. Ensure that the **Bus security** check box is selected.
5. Click **Next**. The Bus Security Configuration wizard is started.
6. Read the Introduction panel, and click **Next**.
7. If the wizard detects that administrative security is disabled, follow the prompts to select, and configure the appropriate user repository.
8. Click **Next**. A summary of the administrative security settings for the bus is displayed.
9. Review the summary, and click **Finish**. Administrative security for the cell is now enabled.
10. If you do not want clients to use SSL protected transports, clear the check box **Require clients use SSL protected transports** .
11. Select a security domain for the bus.
12. If you have selected to use a custom security domain, follow the prompts to specify a user realm.
13. Review the summary of your choices, and click **Finish**.
14. Save your changes to the master configuration.

Results

You have created a new bus secured with your chosen security settings.

What to do next

- You must propagate the bus security configuration to all the affected nodes, and restart the servers. For more information, see Synchronizing nodes using the wsadmin scripting tool and Starting an application server.
- You can add bus members to the bus.
- Groups of users in the user repository require explicit authority to access the bus. For more information, see “Administering authorization permissions” on page 56.

Securing an existing bus by using multiple security domains

You can configure an existing bus to use a cell-level or custom security domain. Using non-global security domains provides the scope to use multiple security domains. The bus can inherit security settings from the cell, or have a unique security configuration.

Before you begin

- Review the information in Service integration security planning and Messaging security and multiple security domains.
- The bus you want to secure must exist in the administrative console. If you want to create a new bus, see “Adding a secured bus” on page 44.
- Ensure that all the bus members are at WebSphere Application Server Version 7.0 or later; use of non-global security domains is not supported for earlier versions of WebSphere Application Server. If the bus has a WebSphere Application Server Version 6 bus member, see “Securing an existing bus by using the global security domain” on page 47. For more information about using security domains, see Service integration security planning and Messaging security and multiple security domains.
- Ensure that there are no indoubt transactions on the messaging engine because incomplete transactions cannot be recovered after the bus is secured. For more information, see Resolving indoubt transactions.
- Stop all servers on which the SIB Service enabled. This ensures that the bus security configuration is applied consistently when the servers are restarted. For more information, see Stopping an application server.

About this task

This task uses the administrative console Bus Security Configuration wizard to secure an existing bus. If the wizard detects that administrative security for the cell is disabled, it prompts you to enable it. You must specify the type of user repository, the administrative security username and password. By default, connecting clients are required to use SSL protected transports to ensure data confidentiality and integrity. You can choose not to use this option. You can specify that the bus uses the cell-level or a custom security domain. If you choose a custom security domain, you must also specify a user realm.

Procedure

1. In the navigation pane, click **Service integration -> Buses -> *security_value***. The general properties for the selected bus are displayed.
2. Click **Configure Bus Security** to start the Bus Security Configuration wizard.
3. Read the Introduction panel, and click **Next**.
4. If administrative security is disabled, follow the instructions to configure the appropriate user repository, and click **Next**.
5. Review the summary of your choices:
   a. Optional: If you want to make changes, click **Previous** to return to an earlier panel, and make the changes you require.
   b. Click **Finish** when you are ready to confirm your choices.

   Administrative security for the cell is now enabled.
6. If you do not want clients to use SSL protected transports, clear the check box **Require clients use SSL protected transports** .
7. Select the cell-level or custom security domain for the bus.
8. Optional: To create a new custom security domain:
   a. Use the name suggested for the security domain, or type a new one.
   b. Optional: Provide a description of the security domain.
   c. Select a user realm for the domain. You can use the user realm configured in the global security domain, or follow the steps to configure a new user realm.
9. Click **Next**.
10. Review the summary of your choices:
    a. Optional: If you want to make changes, click **Previous** to return to an earlier panel, and make the changes you require.

b. Click **Finish** to confirm your choices.
11. Save your changes to the master configuration.

Results

You have specified that the selected bus uses a cell-level or custom security domain. The security settings configured for the bus are displayed in the updated Bus Security Settings panel. The bus is secured after you restart all the servers that are members of the bus, or (for a bus that has bootstrap members) servers for which the SIB service is enabled.

What to do next

You must propagate the bus security configuration to all the affected nodes, and restart the servers. For more information, see Synchronizing nodes using the wsadmin scripting tool and Starting an application server.

Securing an existing bus by using the global security domain

Use this task to secure an existing service integration bus by using the global security domain.

Before you begin

- Review the information in Service integration security planning.
- The bus you want to secure must exist in the administrative console. If you want to create a new bus, see “Adding a secured bus” on page 44.
- If administrative security is not enabled for the cell that hosts the bus, the wizard prompts you to enable it. You need to know the type of user repository, and the administrative security username and password.
- If the service bus contains a bus member at WebSphere Application Server Version 6, the wizard prompts you to select an existing authentication alias, or specify a new one. If you want to specify a new authentication alias, you must provide a username and password.
- Ensure that there are no indoubt transactions on the messaging engine because incomplete transactions cannot be recovered after the bus is secured. For more information, see Resolving indoubt transactions.
- Stop all servers on which the SIB Service enabled. This ensures that the bus security configuration is applied consistently when the servers are restarted. For more information, see Stopping an application server.

About this task

Use this task if you want to secure a bus that exists already in the administrative console, and you want to use the default global security domain. For example, a bus that has a bus member at WebSphere Application Server Version 6. A mixed-version bus cannot use non-global security domains.

This task uses an administrative console wizard to guide you through the steps to secure a bus. The following steps are conditional, depending on the bus environment:

- If administrative security is not enabled for the cell that hosts the bus, the wizard prompts you to enable administrative security.
- If the bus has a bus member at WebSphere Application Server Version 6, the wizard prompts you for an authentication alias to establish trust between bus members, and to enable the bus to operate securely.

Use the administrative console to secure a selected bus by using the global security domain as follows:

Procedure

1. In the navigation pane, click **Service integration -> Buses -> *security_value***. The general properties for the selected bus are displayed.

2. Click **Configure Bus Security** to start the Bus Security Configuration wizard.
3. Read the Introduction panel, and click **Next**. The next step is conditional, depending on whether administrative security is enabled or disabled:
   - If administrative security is disabled, complete all the following steps.
   - If administrative security is already enabled, continue from step 7.
4. Select the appropriate user repository, and click **Next**.
5. Depending on the type of user registry you selected, do one of the following:
   - For a federated repository, specify a username and password for administrative security, and click **Next**.
   - For all other types of repository, follow the wizard prompts, and click **Next**.
6. Review the summary of your choices:
   a. Optional: If you want to make changes, click **Previous** to return to an earlier panel, and make the changes you require.
   b. Click **Finish** when you are ready to confirm your choices.

   Administrative security for the cell is now enabled.
7. If you do not want clients to use SSL protected transports, clear the check box **Require clients use SSL protected transports** . By default, clients are required to use SSL protected transports to ensure data confidentiality and integrity.
8. Select the global security domain option, and click **Next**.
9. If at least one bus member is at Version 6, you must specify an authentication alias. Specify either an existing authentication alias, or create a new one:
   - Select **Specify existing authentication alias**, and select the alias name from the drop-down list.
   - Select **Create a new authentication alias**, type a unique alias name and password.
10. Review the summary of your choices:
    a. Optional: If you want to make changes, click **Previous** to return to an earlier panel, and make the changes you require.
    b. Click **Finish** to confirm your choices.
11. Save your changes to the master configuration.

Results

You have secured the bus using the global security domain. The new security settings for the bus are displayed in the updated Bus Security Settings panel. The bus is secured after you restart all the servers that are members of the bus, or (for a bus that has bootstrap members) servers for which the SIB service is enabled.

What to do next

- You must propagate the bus security configuration to all the affected nodes, and restart the servers. For more information, see Synchronizing nodes using the wsadmin scripting tool and Starting an application server.
- Groups of users in the user repository require explicit authority to access the bus. For more information, see “Administering authorization permissions” on page 56.

Migrating an existing secure bus to multiple domain security

Use this task to migrate a secured service integration bus from the global security domain to a cell-level or custom security domain.

Before you begin

- Review the information in Service integration security planning and Messaging security and multiple security domains.

- All the bus members must be at WebSphere Application Server Version 7.0 or later; use of multiple domain security is not supported for earlier versions of the product.
- Ensure that there are no indoubt transactions on the messaging engine because incomplete transactions cannot be recovered after the bus is secured. For more information, see Resolving indoubt transactions.
- Stop all servers on which the SIB Service enabled. This ensures that the bus security configuration is applied consistently when the servers are restarted. For more information, see Stopping an application server.

About this task

The security settings for a bus are held in a security domain. There are three types of security domain:
- The global security domain which a bus uses by default.
- A cell level security domain which the bus might inherit from the administrative cell.
- A custom domain which might contain security settings that are unique to the bus.

You can use the administrative console to change the type of security domain that the bus uses. Note that the link **Configure Security Domain** only becomes active if you select and apply the option to use a selected security domain. In this case, you must also specify a user realm. You can either use the existing global security settings, or customize a user realm specifically for the domain.

Procedure

1. In the navigation pane, click **Service integration -> Buses -> *security_value***. The security settings panel for the selected bus are displayed.
2. Select either **Inherit the cell level security domain** or **Use the selected domain**, depending on the type of security domain you want to use for the bus.
3. Click **Apply**.
4. Complete the following steps if you want to create a custom security domain:
   a. Click the link **Configure Security Domain**. The security domain configuration panel for the selected bus is displayed.
   b. Use the name suggested for the security domain, or type a new one.
   c. Optional: Type a description of the security domain.
   d. Select the type of user realm for the domain. You can either use the global security settings, or configure a new one.
5. Click **Next**.
6. Review the summary of your choices:
   a. Optional: If you want to make changes, click **Previous** to return to an earlier panel, and make the changes you require.
   b. Click **Finish** to confirm your choices.
7. Save your changes to the master configuration.

Results

You have migrated your existing bus from the global domain to a non-global security domain. The new security settings for the bus are displayed in the updated Bus Security Settings panel.

What to do next

You must propagate the bus security configuration to all the affected nodes, and restart the servers. For more information, see Synchronizing nodes using the wsadmin scripting tool and Starting an application server.

Configuring bus security by using an administrative console panel

Use the administrative console to configure the security properties for an existing service integration bus.

Before you begin

- Review the information in Service integration security planning and Messaging security and multiple security domains.
- The bus must exist in the administrative console. If you want to create a new bus, see Adding buses.
- Ensure that there are no indoubt transactions on the messaging engine because incomplete transactions cannot be recovered after the bus is secured. For more information, see Resolving indoubt transactions.
- Stop all servers on which the SIB Service enabled. This ensures that the bus security configuration is applied consistently when the servers are restarted. For more information, see Stopping an application server.
-

About this task

This task uses the Bus Security administrative console panel. You can start the Bus security wizard from the panel, or specify individual security properties directly in the panel. The bus security properties are effective only when administrative security for the cell is enabled. If the wizard detects that administrative security is disabled, it prompts you to enable it.

The security properties available to a particular bus depend on the versions of the bus members:

- If the bus has a WebSphere Application Server Version 6 bus member, you must specify the global security domain. You must also specify an inter-engine authentication alias to prevent unauthenticated messaging engines from establishing a connection with the bus.
- If the bus contains Version 7.0 or later bus members only, you can specify any type of security domain. You do not need to specify an inter-engine or mediation authentication alias.

If you want to run mediations across multiple security domains, you can specify a single server identity for the bus, rather than specify a mediation authentication alias for each domain. You can use a server identity to run mediations on the global domain.

Procedure

1. In the navigation pane, click **Service integration -> Buses -> *security_value***. *security_value* is either `Enabled` or `Disabled`, depending on the security status of the bus.
2. Optional: Click **Launch Bus Security Wizard** to start the wizard, or specify the following properties directly:

   Enable bus security
   Bus security is enabled by default. Clear this check box if you want to disable security for the selected bus. Note that the check box is read-only if administrative security is disabled.

   Inter-engine authentication alias
   The name of the authentication alias used to authorize communication between messaging engines on the bus. Specify an inter-engine authentication alias if the bus has a Version 6 bus member, bus security is enabled, and you want to prevent unauthorized messaging engines from establishing a connection with the bus.

   Permitted transports
   Specify one of the following transports for the bus:

   - Any messaging transport chain defined to any bus member.
   - Only messaging transport chains that are protected by an SSL chain.
   - Only the transports specified in the list of permitted transports.

If you want to add and remove permitted transports, click **Service integration -> Buses -> *security_value* -> [Additional Properties] Permitted transports**.

Use the Server ID when running mediations
Check this option if you want to run mediations by using the server identity, instead of by using a mediation authentication alias.

Mediations are deployed as applications, and run in the domain used by the application server, not the bus domain. If you want to run a mediation on multiple servers in different domains, you must ensure that the user identity in the mediation authentication alias exists in the configuration for each domain. Alternatively, you can choose to use the server identity option. You can use this option when multiple domains are not in use.

Mediations authentication alias
The name of the authentication alias used to authorize mediations to access the bus. If the bus has a WebSphere Application ServerVersion 6.0.x bus member, you must specify a mediations authentication alias. If you specify a mediations authentication alias for a bus that contains WebSphere Application ServerVersion 7.0 or later bus members only, it is ignored.

Bus security domain
Specify one of the following security domains for the bus:

Global domain
You must specify the global domain if the bus contains a Version 6 bus member, or you do not want the bus to use multiple domains.

Cell level domain
Specify the cell-level security domain if the bus has Version 7.0 or later bus members only, and you want the bus to share security settings with the administrative cell.

Custom domain
Specify a custom security domain if the bus has Version 7.0 or later bus members only, and you want the bus to use a security domain that is used by another resource, or you want to create a new security configuration for this bus.

3. Save your changes to the master configuration.

Results

You have configured security properties for the selected bus.

What to do next

You can use the administrative console to control access to the bus.

Configuring the bus to access secured mediations

Use this task to ensure that the service integration bus is authorized to access secured mediations.

Before you begin

The mediation is secured by using a Java Platform, Enterprise Edition (Java EE) Connector Architecture authentication alias. For information about creating a Java EE authentication alias, see Managing Java 2 Connector Architecture authentication data entries for JAAS.

About this task

To configure the bus to access a secured mediation, you must add the mediation authentication alias for the secured mediation to the properties for the bus:

- If the bus has a Version 6 bus member, you must provide the principal and its associated password.

- If the bus has WebSphere Application Server Version 7.0 or later bus members only, you need only provide the principal.

Procedure

1. Log into the navigation pane.
2. Click **Service integration -> Buses -> *security_value***. The bus security configuration panel is displayed.
3. In the **Mediations authentication alias** field, select the principal for the mediation, and its associated password if required.
4. Click **OK**.
5. Save your changes to the master configuration.

Results

The selected bus is configured to access secured mediations.

What to do next

You can assign security roles to your mediation handlers to protect them from use by unauthorized users. For more information, see "Deploying secured applications" on page 120.

Configuring a bus to run mediations in a multiple security domain environment

Use this task to configure a secured bus so that it can run mediations successfully on bus members in different security domains.

Before you begin

The secured bus must be configured to use a non-global security domain. For more information about securing buses by using multiple security domains, refer to "Securing buses" on page 44.

About this task

If your bus topology has bus members in different security domains, you can configure the bus to allow mediations to run under the server identity. This means that a mediation can run on any server in any domain. You do not have to add a dedicated user ID for each mediation to the user repository, or maintain a mediation authentication alias.

Use the administrative console to configure a secured bus to run mediations successfully as follows:

Procedure

1. In the navigation pane, click **Service integration -> Buses -> *security_value***. The security settings for the selected bus are displayed.
2. Check the option **Use the Server ID when running mediations**.
3. Click **Apply**.
4. Save your changes to the master configuration.

Results

You have configured the bus to run mediations successfully across servers in multiple security domains.

What to do next

You can use the administrative console to control access to the bus by administering users and groups in the bus connector role.

Disabling bus security

If you do not require messaging security, you can choose to disable messaging security. Any new buses added after messaging is disabled are not secured.

Before you begin

- Ensure that there are no indoubt transactions on the messaging engine because incomplete transactions cannot be recovered after the bus security is disabled. For more information, see Resolving indoubt transactions.
- Stop all servers on which the SIB Service enabled before you disable bus security. This ensures that the bus security configuration is applied consistently when the servers are restarted. For more information, see Stopping an application server.

About this task

Messaging security is enabled by default, providing administrative security for the cell is enabled. However, there might be circumstances when you do not require messaging security, for example on a development system. In this case, you can use the administrative console to disable messaging security. If you want to re-enable bus security, see “Securing buses” on page 44.

Procedure

1. In the navigation pane, click **Service integration -> Buses**. A list of buses is displayed.
2. Find the bus for which you want to disable security, and click **Enabled** in the security column. The security settings for the selected bus are displayed.
3. Clear the check box **Enable bus security**.
4. Click **Apply**.
5. Save your changes to the master configuration.

Results

You have disabled security for the selected bus.

What to do next

You must propagate the bus security configuration to all the affected nodes, and restart the servers. For more information, see Synchronizing nodes using the wsadmin scripting tool and Starting an application server.

Enabling client SSL authentication

You can configure a service integration bus to allow connecting client JMS applications to authenticate by using Secure Sockets Layer (SSL) certificates.

About this task

This is the parent task for the steps required to establish client SSL authentication for connections between messaging engines and JMS applications running in a client container. You must configure the bus to allow client SSL authentication, and configure the JMS client application to undertake client SSL authentication.

Procedure

- “Configuring a bus to allow client SSL authentication” on page 54.
- “Configuring JMS client applications to perform client SSL authentication” on page 54.

Configuring a bus to allow client SSL authentication

You can configure a service integration bus to enable connecting client JMS applications to authenticate by using Secure Sockets Layer (SSL) certificates.

Before you begin

You must ensure that the following tasks have been completed:

- Administrative security is enabled. For more information, see Enabling security.
- A stand-alone Lightweight Directory Access Protocol (LDAP) user registry has been configured for storing user and group IDs. To access the user registry, you must know a valid user ID that has the administrative role, and password, the server host and port of the registry server, and the base distinguished name (DN). For more information, see Configuring Lightweight Directory Access Protocol user registries.
- Bus security is enabled. For more information, see “Disabling bus security” on page 53.
- JMS client applications have been configured to authenticate by using client SSL certificates.

About this task

If you want to allow connecting JMS application clients to authenticate to the bus by using client SSL certificates, define an SSL configuration. There are two parts to this task. First you use the administrative console to map SSL certificates to entries in the LDAP user registry. Secondly, you create a unique SSL configuration for each endpoint address for which you want to use client SSL authentication. Do not use the default SSL configuration for the bus.

Procedure

1. Use the administrative console to define certificate filters to map an SSL certificate to an entry in the LDAP server. For more information, see Creating a Secure Sockets Layer configuration. The client SSL certificate is mapped to a user ID in the user registry.
2. Create a separate SSL configuration file for each endpoint address for server, bus member or cluster on the bus, and select that client authentication is required. For more information, see Creating a Secure Sockets Layer configuration

Results

The bus is configured to allow client SSL authentication.

What to do next

Connecting JMS client applications can now authenticate to the bus using client SSL certificates.

Configuring JMS client applications to perform client SSL authentication

You can configure JMS client applications to authenticate to the bus by using client Secure Sockets Layer (SSL) authentication.

Before you begin

- You have already obtained a Secure Sockets Layer (SSL) certificate for the JMS client application.
- The JMS client application is already configured to use SSL. For more information, see ssl.client.props client configuration file

About this task

This task has two objectives. First, you install the SSL certificate for the client application in the key store for the application client. Secondly, you modify the sib.client.ssl.properties file to use client SSL

authentication. You use the Key Management (iKeyman) utility to work with SSL certificates. The iKeyman user interface is Java-based and uses the Java support that is installed with IBM HTTP Server.

Take the following steps to configure a JMS client application to use client SSL authentication:

Procedure

1. Start the iKeyman user interface. Refer to the *iKeyman User Guide* available from IBM developer kits for more information about using iKeyman.
2. When prompted, select the key store for the JMS client application.
3. When prompted for the type off certificate to work with, select the option **Personal certificates**. A list of personal certificates is displayed.
4. Select that you want to import a certificate to the selected key store.
5. When prompted, type the location and name for the certificate. You can provide an alias for the certificate. The certificate is installed into the keystore of the client application.
6. Close the iKeyman user interface.
7. Open a text editor to work with the `sib.client.ssl.properties` properties file. This file is located in the `profile_root/properties` directory of the application server installation, where `profile_root` is the directory in which profile-specific information is stored.
8. Set the value for the property `com.ibm.ssl.client.clientAuthentication` to *True*.
9. Set the value for the property `com.ibm.ssl.client.keyStoreClientAlias` to the alias name for the certificate in the client key store.
10. Save the `sib.client.ssl.properties` properties file.

Results

You have now configured a JMS client application to use client SSL authentication.

Adding unique names to the bus authorization policy

How to update the authorization policy for the service integration bus with unique name entries.

About this task

You should carry out this task if you are migrating from WebSphere Application Server Version 6 to WebSphere Application Server Version 7.0 or later. In this task, you manually run the **`populateUniqueNames`** command to query the user repository for a selected bus for unique names, and add them to the authorization policy. If you do not manually run this command, the messaging engine performs the query, and adds the missing unique names to the authorizations policy, which adversely affects the start up time.

When you migrate from a Version 6 node to a Version 7.0 or later node, the authorization policy only contains the user and group security names; it does not contain the names in the user registry that uniquely define each user and group. If an LDAP user registry is in use, the unique name is the distinguished name (DN). By default, only missing unique names are added to the authorization policy. If you set the **`-force`** parameter, all unique name entries added to the authorization policy

Procedure

1. Run a scripting command.
2. At the **`wsadmin`** command prompt, type the **`populateUniquenames`** command. The following example syntax queries the user repository for the unique names that match the security names for a bus called Bus 1, and adds the missing unique names to the authorization policy .
   ```
   AdminTask.populateUniquenames('[-bus Bus1]')
   ```
3. Save your changes to the master configuration repository. The following example presents the syntax:
   ```
   AdminConfig.save()
   ```

Results

The authorization policy for the bus is updated with the missing unique names.

Example

The following example updates all the unique name entries in the authorization policy for a bus called Bus 1.

```
AdminTask.populateUniqueNames(AdminTask.populateUniquenames('[-bus Bus1 -force TRUE]')
```

What to do next

Use the administrative console to administer bus security authorizations.

Administering authorization permissions

Service integration messaging security uses role-based authorization. When a user is assigned to a role, the user is granted all of the permissions that the role contains. By administering authorization permissions, you can control user access to a bus and its resources when messaging security is enabled.

Before you begin

For guidance on security authorization for a service integration bus, refer to Service integration security planning.

About this task

When a bus is created, a set of default authorization roles is created. Default roles provide authenticated users who have the bus connector role with full access to all local destinations on the bus. By default, only members of the Server group have the bus connector role. If a specific user needs to connect to the bus, you must explicitly add that user to the bus connector role.

You can make changes to authorization permissions when messaging security is enabled or disabled. Any changes that you make when security is disabled do not have any effect until security is enabled, as described in "Disabling bus security" on page 53.

LDAP Registry Tip: When you specify the group authorization permissions, the group distinguished name (DN) must be used. If you specify a common name (CN) for the group name, users in that group do not have the specified authorities. For more details see Standalone Lightweight Directory Access Protocol registries.

When security is enabled, by default users cannot connect to a foreign bus. If a specific user needs to connect to a foreign bus, you must explicitly add that user to the foreign bus access list.

Procedure

- "Administering the bus connector role" on page 57
- "Administering default roles" on page 59
- "Administering destination roles" on page 62
- "Administering foreign bus roles" on page 67
- "Administering temporary destination prefix roles" on page 70
- "Administering topic space root roles" on page 74
- "Administering topic roles" on page 77
- "Removing access roles from unknown users and groups" on page 81

Administering the bus connector role

Adding a group of users to the bus connector role for a local bus grants the members of the group permission to access local bus destinations. Use the administrative console to list, add and remove groups of users in the bus connector role.

Before you begin

The users and groups you want to work with must exist in the user repository.

About this task

Service integration bus security uses role-based authorization. By administering groups of users in the bus connector role, you can control access to the local service integration bus and its resources when messaging security is enabled.

Procedure

- "Listing users and groups in the bus connector role"
- "Adding users and groups in the bus connector role"
- "Removing users and groups from the bus connector role" on page 59

Listing users and groups in the bus connector role:

Service integration bus security uses role-based authorization. By listing the users and groups in the bus connector role, you can find out which users and group members are authorized to connect to a selected secured local bus, and its resources.

Before you begin

Ensure that security is enabled for the bus. For more information, refer to "Securing buses" on page 44.

About this task

In this task you use the administrative console to list the users and groups in the bus connector role for a selected local bus. By default, the list is empty for a new bus.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Users and groups in the bus connector role**.

Results

A list of the users and groups in the bus connector role for the selected bus is displayed. The list is empty for a new bus.

What to do next

You can add and delete users and groups in the bus connector role for the selected bus.

Adding users and groups in the bus connector role:

Service integration bus security uses role-based authorization. By adding users and groups to the bus connector role for a secured bus, you can control which users and group members have access to the bus and its resources.

Before you begin

- Ensure that security is enabled for the bus. For more information, refer to "Securing buses" on page 44.
- The users and groups that you want to add to the bus connector role must exist already in the user repository.

About this task

Adding users and groups to the bus connector role enables them to connect to the bus to carry out messaging operations. You can add a user directly to the bus connector role, or indirectly by adding a group to which the user belongs. You can also add special groups of users. There are three special groups:

Server
: The server identity is a WebSphere Application Server . You cannot specify the Server group for a JMS message-driven bean (MDB).

All Authenticated
: This group comprises all user identities that authenticate successfully to the bus.

Everyone
: The user identities in this group are anonymous, and connect to the bus without security authentication.

Tips:

- If the user registry is an LDAP registry, you must use the group distinguished name (DN) when you specify a group name to add to a bus connector role. Using the common name (CN) causes problems in security authorization. For more information, refer to Service integration bus security: Troubleshooting tips and Standalone Lightweight Directory Access Protocol registries.
- If you attempt to add a user or a group that is already a member of the bus connector role, a warning message is displayed.

In this task you use an administrative console wizard to add groups and users to the bus connector role for a selected local bus.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Users and groups in the bus connector role**. A list of the users and groups already in the bus connector role for the selected bus is displayed. By default, the list is empty for a new bus.
3. Click **New** to start the Security Resource Wizard.
4. Choose whether you want to add groups or users:
   - If you want to add a special group, select **The built-in special groups** option.
   - If you want to add other groups or users in the user repository, select the appropriate option, and complete the following mandatory fields:

     Search pattern
     : Specify a string to match against user IDs or group names in the user repository. Only user IDs or group names that match the search pattern are retrieved, subject to the maximum number of search results. You can specify wildcard characters.

     Maximum number of search results to display
     : Specify the maximum number of user IDs or group names to display.
5. Click **Next** to display a list of groups or users.
6. Select the names of the groups or users you want to add to the bus connector role, and click **Next**.
7. Click **Finish** to confirm you choices.

8. Save your changes to the master configuration.

Results

The selected users and groups are added to the bus connector role for the selected bus.

Removing users and groups from the bus connector role:

Service integration bus security uses role-based authorization. By removing selected users and groups from the bus connector role for a selected secured bus you prevent those users and group members from connecting to the bus.

About this task

The users and groups that you remove from the bus connector role for a bus can no longer undertake messaging operations on the bus. Note that removing a user from the bus connector role does not prevent that user from connecting to the bus if they are also a member of a group that is in the bus connector role.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Users and groups in the bus connector role**. A list of the users and groups in the bus connector role for the selected bus is displayed.
3. Select the check box next to the name of the user or group that you want to remove, and click **Delete**.
4. Save your changes to the master configuration.

Results

The selected users and groups are removed from the bus connector role.

Administering default roles

Service integration bus security uses role-based authorization. By adding a user or a group to the default roles for a secured bus, you can control which users and group members have access to the bus and its resources in the default roles when messaging security is enabled.

About this task

The default roles are sender, receiver, browser, and creator. These roles apply to bus destinations that do not have a destination role, or have been configured to inherit the default roles. By default, all destinations inherit the default roles. Access in the default roles exists in addition to any specific access roles that have been configured for a destination.

Procedure

- "Listing users and groups in default roles"
- "Adding users and groups to default roles" on page 60
- "Removing users and groups from default roles" on page 61

Listing users and groups in default roles:

Service integration bus security uses role-based authorization. By listing the users and groups in the default roles for a selected secured bus, you can find out which users and group members are authorized to perform messaging operations on a local bus destinations that is allowed to inherit default roles.

About this task

In this task you use the administrative console to list users and groups in the default access roles for a selected secured bus. The default role types are sender, receiver, browser and creator.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage default access roles**. The Default access roles panel is displayed. The information for the default access is displayed in a collapsed section.
3. Expand the Default access header.

Results

A list of users and groups in default roles for the selected bus is displayed.

What to do next

You can also add and remove users and groups in default roles.

Adding users and groups to default roles:

Service integration bus security uses role-based authorization. By adding selected users and groups to the default roles for all the local bus destinations on a secured bus, you provide those users and group members with access to the local bus destinations that are allowed to inherit default roles.

Before you begin

If a bus destination is not allowed to inherit the default roles, you must first add the user or group to the role that grants authorization permission for the specific local destination. For more information, see “Adding users and groups to destination roles” on page 63.

About this task

The default roles are sender, receiver, creator and browser. In this task you use an administrative console wizard, the Security wizard, to add selected users or groups to the default roles. The Security wizard requests information to enable it to retrieve selected users or groups from the potentially very large number of users and groups in the user repository.

Procedure

1. Log onto the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage default access roles**. The Default access roles panel is displayed.
3. Expand the Default access header to list the users and groups that have been assigned to default access roles.
4. Click **Add** to start the Security wizard. The wizard takes you through the following steps to add selected users or groups to default access roles:
   a. Search for the users or groups that you want to add to default access roles:

      Users or Groups
      Select either **Users** or **Groups** to specify whether you want to grant access roles to users or groups.

      Search pattern
      This field is mandatory. Specify a search string that is matched against user IDs or group

names in the user repository. Only user IDs or group names that match the search pattern are retrieved, subject to the maximum number of search results. Wildcard characters are allowed.

Maximum number of search results to display
: This field is mandatory. Specify the maximum number of user IDs or group names you want the administrative console to display.

b. Click **Next**. The wizard displays the users or groups in the user repository that match the information that you provided in the previous step.
c. Select the check boxes next to the user IDs or group names that you want to add to the default access roles, and click **Next**. A list of user IDs or group names that you can add to the default access roles is displayed. Note that some users or groups might already be assigned to default access roles.
d. Select the role types that you want to assign to a user or group. For example, to assign a group to the sender role, click the sender icon for the appropriate group name. The icon changes from ☐ to ☑ to show that you have added the user or group to the access role for the resource.
e. Complete the previous step for each user or group that you want to add to access roles, and then click **Next**. A summary of your role type assignments is displayed.
f. Optional: Click **Previous** to review and change your assignments, if required.
g. Click **Finish** to confirm your assignments. The Default access roles panel is redisplayed and shows the new role type assignments.

5. Save your changes to the master configuration.

Results

The selected users and groups are added to selected default roles for the selected bus.

Removing users and groups from default roles:

Service integration bus security uses role-based authorization. By removing selected users and groups from the default roles for a selected secured bus, you prevent those users and group members from accessing the bus by using the default roles.

About this task

When you remove users and groups from the default roles for a bus, they can no longer access local bus destinations that inherit default access permissions. Note that removing a user from the default roles does not prevent that user from accessing the bus if they are also a member of a group that has been assigned to the default roles for that bus.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage default access roles**. The Default access roles panel is displayed.
3. Expand the Default access header to list the users and groups that have been assigned to default access roles for the selected bus.
4. Select the users and groups that you want to remove from the default access roles for this bus, and click **Remove**.
5. Save your changes to the master configuration.

Results

The selected users and groups are removed from the default roles for the selected bus. The Default access roles panel displays the changes to the default access role assignments.

What to do next

You can complete other security administration tasks by using the administrative console.

Administering destination roles

Service integration bus security uses role-based authorization. When messaging security is enabled, users and groups must have authority to undertake messaging operations, at a bus destination. By administering destination roles, you can control which users and groups can undertake operations at a bus destination, and the types of operations that they can perform.

About this task

You use the administrative console to administer users and groups in access roles for a destination. The access roles available for a destination depend on the type of destination. The table below lists the roles that you can assign for each destination type:

Table 11. Destination roles. The first column of the table contains the list of destination types. The second column contains the access roles that can be assigned for the destination types.

Destination type	Access roles
queue	sender, receiver, browser, creator
port	sender, receiver, browser, creator
webService	sender, receiver, browser, creator
topicSpace	sender, receiver
foreignDestination	sender
alias	sender, receiver, browser

In addition to controlling which users and groups have access to a specific local or foreign destination, you can also control the inheritance of access roles for a specific local destination. In this case, the default access roles that apply to all the destinations in the local bus namespace are added to any access roles that have been added for a specific destination.

Procedure

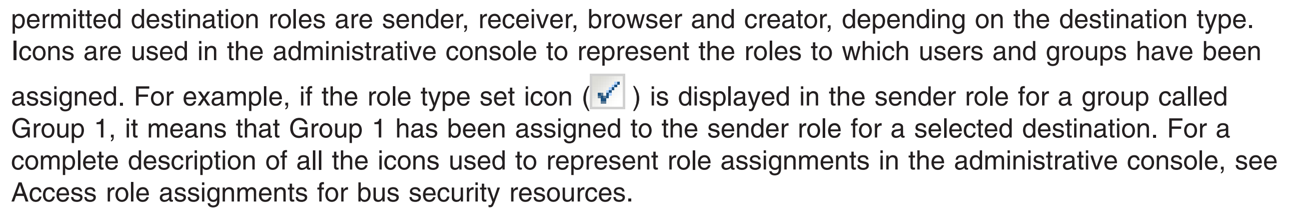
- "Adding users and groups to destination roles" on page 63
- "Removing users and groups from destination roles" on page 65
- "Listing users and groups in destination roles"
- "Restoring default inheritance for a destination" on page 65
- Disabling inheritance from the default resource
- "Overriding inheritance from the default resource for a destination" on page 66

Listing users and groups in destination roles:

Service integration bus security uses role-based authorization. By listing the users and groups in the destination roles for a selected secured bus, you can find out which users and groups are authorized to access the bus, and its resources.

About this task

In this task you use the administrative console to list all the users and groups in destination roles for selected destinations. The list includes users and groups that have references in the service integration role-based configuration; it does not include all the users and groups that exist in the user repository. The

permitted destination roles are sender, receiver, browser and creator, depending on the destination type. Icons are used in the administrative console to represent the roles to which users and groups have been assigned. For example, if the role type set icon () is displayed in the sender role for a group called Group 1, it means that Group 1 has been assigned to the sender role for a selected destination. For a complete description of all the icons used to represent role assignments in the administrative console, see Access role assignments for bus security resources.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage destination access roles**. The Destinations panel lists all the destinations defined for the selected bus.
3. Select one or more destinations to work with:
   - Click the name of a single destination.
   - Select the check boxes next to multiple destinations, and click **Manage Access Roles**.

   The Destination access roles panel is displayed. The information for each selected destination is displayed in a collapsed section.
4. Expand a destination header.

Results

The Destination access roles panel lists the users and groups in access roles for the expanded destination.

What to do next

You can now administer the users and groups in destination roles at this destination.

Adding users and groups to destination roles:

Service integration bus security uses role-based authorization. By adding users and groups to the destination roles for a secured bus, you can control which users and group members can undertake messaging operations at a bus destination.

Before you begin

Ensure that the following conditions are met:
- Security is enabled for the bus. For more information, see “Securing buses” on page 44.
- The users and groups that you want to add to destination roles must exist already in the user repository.

About this task

By adding users or groups to the destination role, you grant the users or groups authority to undertake the operation defined by the role at a selected destination. The destination roles are sender, receiver, browser, and creator, depending on the destination type.

In this task you use the administrative console Security wizard to retrieve selected users or groups from the user repository, and add them to destination roles for selected bus destinations.

Tip: To add a large number of users to destination roles, it is advisable to create a group in the user repository, and add the group to the destination roles.

Procedure

1. Start the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage destination access roles**. A list of the destinations defined for the selected bus is displayed in the Destinations panel.
3. Select one or more destination to work with:
   - Click a single destination name.
   - Select the check boxes next to multiple destination names, and then click **Manage Access Roles**.

   The Destination access roles panel is displayed. The information for each destination you have selected is displayed in a collapsed section.
4. Expand a destination header to list the users and groups that have been assigned to roles for this destination. You can verify that the user or group you want to add does not already have a role at this destination.
5. Click **Add** to start the Security wizard. The wizard takes you through the following steps to add selected users or groups to access roles for the expanded destination:
   a. Search for the users or groups that you want to add to access roles for the expanded destination:

      Users or Groups
      Select either **Users** or **Groups** to specify whether you want to grant access roles to users or groups.

      Search pattern
      This field is mandatory. Specify a search string that is matched against user IDs or group names in the user repository. Only user IDs or group names that match the search pattern are retrieved, subject to the maximum number of search results. Wildcard characters are allowed.

      Maximum number of search results to display
      This field is mandatory. Specify the maximum number of user IDs or group names you want the administrative console to display.
   b. Click **Next**. The wizard displays the users or groups in the user repository that match the information that you provided in the previous step.
   c. Select the check boxes next to the user IDs or group names that you want to add to access roles for the currently expanded destination, and click **Next**. A list of user IDs or group names that you can add to destination roles is displayed. Note that some users or groups might already be assigned to access roles for this destination.
   d. Select the appropriate access role icon for the user ID or group name that you want to add to the role at this destination. For example, select the **Receiver** icon for a user ID or group name that you want to add to the receiver role. The icon changes from ☐ to ☑ to show that you have added the user or group to the access role for the resource.
   e. Repeat the previous step to add more users or groups to access roles for the currently expanded destination, and then click **Next**. A summary of your access role assignments is displayed.
   f. Optional: Click **Previous** to review and change your assignments, if required.
   g. Click **Finish** to confirm your assignments.
6. Repeat steps 4 and 5 for each destination you want to work with.
7. Save your changes to the master configuration.

Results

The selected users and groups are added to the access roles for the currently expanded destination. The new access role assignments are displayed in the Destination access roles panel.

Example

A group called MyGroup receives messages from three queues, Queue 1, Queue 2, and Queue 3. If you want the group MyGroup to produce and consume messages at an additional destination, Queue 4, you add MyGroup to Queue 4, and then add MyGroup to the sender and receiver roles for Queue 4.

What to do next

Use the administrative console to complete other security administrative tasks.

Removing users and groups from destination roles:

Service integration bus security uses role-based authorization. By removing users and groups from the destination roles for a secured bus, you can prevent those users and group members from performing messaging operations on the bus.

About this task

When selected users and groups no longer require access to a destination, you can remove them from all the roles for that destination.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage destination access roles** A list of the destinations defined for the selected bus is displayed in the Destinations panel.
3. Select one or more destinations to work with:
   - Click a single destination name.
   - Select the check boxes next to multiple destination names, and then click **Manage Access Roles**.

   The Destination access roles panel is displayed. The information for each destination you have selected is displayed in a collapsed section.
4. Expand a destination header to list the users and groups that have been assigned to roles at this destination, and verify that the user or group that you want to remove has a role at this destination.
5. Select the users and groups that you want to remove from all role types at this destination, and click **Remove**.
6. Save your changes to the master configuration.

Results

The selected users and groups are removed from all role types at the selected destination. The Manage access roles for users and groups panel displays the updated role type assignments.

Example

The members of three groups, Group A, Group B, and Group C, belong to the sender role and the receiver role for two queue destination, Queue 1 and Queue 2. If Group B is no longer required to send and receive messages on Queue 2, you can use this task to remove Group B from all the role types on Queue 2.

What to do next

Use the administrative console to complete other security administrative tasks.

Restoring default inheritance for a destination:

Service integration bus security uses role-based authorization. By default, all local destinations inherit access roles from the default resource. If default inheritance has been previously overridden, you can restore it for a selected destination.

Before you begin

Default inheritance has been overridden for a selected secured destination. For more information, see "Overriding inheritance from the default resource for a destination."

About this task

If default inheritance has been overridden for a particular destination, you can restore it. In this task, you use the administrative console to restore the role type assignments from the default resource to a selected destination. A destination can only inherit access roles that are allowed for that particular type of destination. For example, a topic space can inherit the sender and receiver roles, but it cannot inherit the browser role. Inherited access roles are added to any existing access roles for the destination.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage destination access roles**. The Destinations panel lists all the destinations defined for the selected bus.
3. Select one or more destinations to work with:
   - Click the name of a single destination.
   - Select the check boxes next to multiple destinations, and click **Manage Access Roles**.

   The Destination access roles panel is displayed. The information for each selected destination is displayed in a collapsed section.
4. Expand a destination to list the users and groups that have been assigned to roles for this destination.
5. Select the **Inherit from default** check box.
6. Click **OK** to save your changes.
7. Save your changes to the master configuration.

Results

The role type assignments for the default resource are inherited by the selected destination. The Destination access roles panel displays the newly inherited default access roles for the destination, and any existing access roles.

Overriding inheritance from the default resource for a destination:

Service integration bus security uses role-based authorization. By default, local destinations can inherit access roles from the default resource. If you do not want users and groups in the default access role to access a particular destination, you can override default inheritance for a selected destination.

About this task

All the destinations in a local bus namespace can inherit default access roles with the following exceptions:

- A destination for which default inheritance is overridden.
- Foreign destinations.
- Alias destinations that have an alias bus name that is not the local bus name.

In this task, you use the administrative console to override default inheritance for a selected destination. This means that the users or groups that belong to the default access role can no longer access the selected destination.

Procedure

1. Log into the administrative console
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage destination access roles**. The Destination panel lists all the destinations defined for the selected bus.
3. Select one or more destinations to work with:
   - Click the name of a single destination.
   - Select the check boxes next to multiple destinations, and click **Manage Access Roles**.

   The Destination access roles panel is displayed. The information for each selected destination is displayed in a collapsed section.
4. Expand a destination to list the users and groups that have been assigned to roles for this destination.
5. Clear the **Inherit from default** check box.
6. Click **OK** to save your changes.
7. Save your changes to the master configuration.

Results

The inherited role type assignments are removed from the selected destination. The Destination access roles panel displays the updated access roles for the destination.

Administering foreign bus roles

Service integration bus security uses role-based authorization. When messaging security is enabled, groups of users require authority to send messages from a local bus destination to a foreign bus. By listing, adding and removing users and groups in foreign bus roles, you can control who can send messages to foreign buses.

Before you begin

These tasks assumes that one or more foreign bus connections have been configured. For more information, see Configuring foreign bus connections.

About this task

The foreign bus connection represents another service integration bus, either in the same cell as the local bus or in a different cell, or it represents a WebSphere MQ queue manager. From the local bus, every other bus is regarded as a foreign bus, even if it is a bus in the same cell. Messages route to a foreign bus either directly through a link between the local bus and the foreign bus, or indirectly through one or more intermediate buses. A member of a group that belongs to the sender role on the local bus and the foreign bus can send messages directly to the foreign bus. The sender role is the only foreign bus role.

Procedure

- "Listing users and groups in foreign bus roles"
- "Adding users and groups to foreign bus roles" on page 68
- "Removing users and groups from foreign bus roles" on page 70

Listing users and groups in foreign bus roles:

Service integration bus security uses role-based authorization. When messaging security is enabled, users and groups require authority to send messages from a secured local bus destination to a secured foreign

bus. By listing all the users and groups in foreign bus roles for a selected foreign bus, you can find out who has authority to send messages from the local bus to the selected foreign bus.

About this task

In this task you use the administrative console to list users and groups in the sender role for selected foreign buses. The list includes users and groups that have references in the service integration role-based configuration. It does not include all the users and groups that exist in the external user repository.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage foreign bus access roles**. A list of all the foreign buses defined for the selected bus is displayed.
3. Select one or more foreign buses:
   - Click the name of a single foreign bus.
   - Select the check boxes next to multiple foreign buses and click **Manage Access Roles**.

   The Foreign bus access roles panel is displayed. The access roles information for each foreign bus is displayed in a collapsible section.
4. Expand the section header for a foreign bus.

Results

A list of all the users and groups that have been assigned to the sender role for the currently expanded foreign bus is displayed.

What to do next

You can add and remove users and groups in the sender role for the selected foreign bus.

Adding users and groups to foreign bus roles:

Service integration bus security uses role-based authorization. When messaging security is enabled, users and groups require authority to send messages from a secured local bus destination to a secured foreign bus. By adding selected users and groups to the sender role for a selected foreign bus, you can control who has authority to send messages to the selected foreign bus.

Before you begin

This task assumes that the following conditions have been met:

- One or more foreign bus connections have been configured for the local bus. For more information, see Configuring foreign bus connections.
- The users and groups that you want to add to foreign bus roles must exist in the user repository.

About this task

By default, when security is enabled, users and groups cannot send messages to a foreign bus. You must add them to the sender role for the foreign bus. In this task you uses an administrative console wizard to select one or more foreign buses, retrieve selected users or groups from the potentially very large number of users and groups in the user repository, and add them to the sender role for the selected foreign buses.

Procedure

1. Start the administrative console.

2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage foreign bus access roles**. A list of the foreign buses defined for the selected bus is displayed in the Foreign buses panel.
3. Select one or more foreign buses to work with:
   - Click a single foreign bus name.
   - Select the check boxes next to multiple foreign bus names, and then click **Manage Access Roles**.

   The Foreign bus access roles panel is displayed. The access roles information for each foreign bus you have selected is displayed in a collapsed section.
4. Expand a foreign bus header to list the users and groups that have been assigned to roles for this foreign bus. You can verify that the user or group you want to add does not already have a role for this foreign bus.
5. Click **Add** to start the Security wizard. The wizard takes you through the following steps to add selected users or groups to the sender role for the selected foreign bus:
   a. Search for the users or groups that you want to add to the sender role for the expanded foreign bus:

      Users or Groups
      Select either **Users** or **Groups** to specify whether you want to grant access roles to users or groups.

      Search pattern
      This field is mandatory. Specify a search string that is matched against user IDs or group names in the user repository. Only user IDs or group names that match the search pattern are retrieved, subject to the maximum number of search results. Wildcard characters are allowed.

      Maximum number of search results to display
      This field is mandatory. Specify the maximum number of user IDs or group names you want the administrative console to display.
   b. Click **Next**. The wizard displays the users or groups in the user repository that match the information that you provided in the previous step.
   c. Select the check boxes next to the user IDs or group names that you want to add to the sender role for the currently expanded foreign bus, and click **Next**. A list of users IDs or group names that you can add to the sender role is displayed. Note that some users or groups might already be assigned to the sender role for this foreign bus.
   d. Select the **Sender** icon for a user ID or group name that you want to add to the sender role. The icon changes from ☐ to ☑ to show that you have added the user or group to the access role for the resource.
   e. Repeat the previous step for each user or group you want to add to the sender role, and then click **Next**. A summary of your role assignments is displayed.
   f. Optional: Click **Previous** to review and change your assignments, if required.
   g. Click **Finish** to confirm your assignments.
6. Save your changes to the master configuration.

Results

The selected users and groups are added to the sender role for the selected foreign bus. The new access roles are displayed in the Foreign bus access roles panel.

What to do next

Use the administrative console to complete other security administrative tasks.

Removing users and groups from foreign bus roles:

Service integration bus security uses role-based authorization. By removing users and groups from the foreign bus roles for a selected secured local bus, you prevent those users and groups from sending messages to the foreign bus.

About this task

In this task you use the administrative console to remove selected users or groups from the sender role for a selected foreign bus.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage foreign bus access roles**. A list of all the foreign buses defined for the selected bus is displayed.
3. Select one or more foreign buses to work with:
   - Click a single foreign bus name.
   - Select the check boxes next to multiple foreign bus names, and then click **Manage Access Roles**.

   The Foreign bus access roles panel is displayed. The access roles information for each selected foreign bus is displayed in a collapsed section.
4. Expand the section header for a foreign bus to display details for all the users and groups that have the sender role for this foreign bus.
5. Select the users and groups that you want to remove from the sender role, and click **Remove**. The selected users and groups are removed from the sender role for this foreign bus.
6. Save your changes to the master configuration.

Results

The selected users and groups are removed from the sender role for the selected foreign bus. The Foreign bus access roles panel displays the changed access role assignments.

What to do next

Use the administrative console to complete other security administrative tasks.

Administering temporary destination prefix roles

Service integration bus security uses role-based authorization. A temporary destination prefix can have two role types: creator and sender. The messaging engine uses the temporary destination prefix at runtime to determine which users and groups have authority to create a temporary destination, and send messages to temporary destinations. By administering temporary destination prefix roles for a bus, you control which users and groups can create and send messages to temporary destinations for a selected bus.

Before you begin

Ensure that security is enabled for the bus. For more information, refer to “Securing buses” on page 44.

About this task

By default, a bus does not contain any temporary destination prefixes. You use the administrative console to add a new temporary destination prefix to the bus, and then assign selected users and groups to the

sender role for the new temporary destination prefix. The creator role is assigned by default. All authenticated users can create temporary destinations by default. You can remove selected users and groups from the sender role for a selected temporary destination prefix, and you can remove a selected temporary destination prefix.

Procedure

- "Listing users and groups in temporary destination prefix roles"
- "Adding users and groups to temporary destination prefix roles"
- "Removing users and groups from temporary destination prefix roles" on page 73
- "Removing a temporary destination prefix" on page 73

Listing users and groups in temporary destination prefix roles:

Service integration bus security uses role-based authorization. Temporary destination prefix roles are used to authorize access to the temporary destinations for a bus. By listing the users and groups in the temporary destination prefix roles for a selected bus, you can find out which users and groups can create temporary destinations, and send messages to temporary destinations for a selected bus.

About this task

In this task you use the administrative console to list users and groups in temporary destination prefix roles for the selected bus. The list includes users and groups that have references in the role-based security configuration for service integration. It does not include all the users and groups that exist in the external user repository.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage temporary destination prefix access roles**. The Temporary destination prefixes panel lists all the temporary destination prefixes defined on the bus.
3. Select one or more temporary destination prefixes to work with:
   - Click the name of a single temporary destination prefix.
   - Select the check boxes next to multiple temporary destination prefixes, and click **Manage Access Roles**.

   The Temporary destination prefix access roles panel is displayed. The access roles information for each temporary destination prefix is displayed in a collapsed section.
4. Expand the header for a selected temporary destination prefix to show its access roles.

Results

The expanded section lists the users, groups and group members that are assigned to access roles for the selected temporary destination prefix.

What to do next

You can now administer the users and groups in the sender role for a selected temporary destination prefix.

Adding users and groups to temporary destination prefix roles:

Service integration bus security uses role-based authorization. The messaging engine uses the temporary destination prefix at runtime to determine whether a client application is authorize to create, or send

messages to a particular temporary destination. By adding users and groups to temporary destination prefix roles for a selected bus, you can control which users and groups can create temporary destinations, and send messages to them.

Before you begin

The users and groups that you want to add to temporary destination prefix roles must already exist in the user repository.

About this task

By default, the bus security configuration does not contain any temporary destination prefixes. In this task, you use the administrative console Security wizard to first add a new temporary destination prefix, and then add users and groups to the sender role for the new temporary destination prefix. Note that the creator role is assigned by default to the creator of the temporary destination; you cannot use the administrative console to add users and groups to the creator role. By default, members of the All Authenticated group have authority in the creator role for temporary destination prefixes.

Procedure

1. Log into the administrative console. The Temporary destination prefixes panel lists all the temporary destination prefixes defined for the selected bus. By default, this list is empty.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage temporary destination prefix access roles**
3. Click **Add** to start the Security wizard:
   a. Define the name of the temporary destination prefix, and identify the users or groups that you want to add to the sender role for the temporary destination prefix:

      Resource
      This field is mandatory. Specify a name for the new temporary destination prefix.

      Users or Groups
      Select either **Users** or **Groups** to specify whether you want to grant access roles to users or groups.

      Search pattern
      This field is mandatory. Specify a search string that is matched against user identities or group names in the user repository. Only user identities or group names that match the search pattern are retrieved, subject to the maximum number of search results. Wild card characters are allowed.

      Maximum number of search results to display
      This field is mandatory. Specify the maximum number of user identities or group names you want the administrative console to display.
   b. Click **Next**. The wizard displays the users or groups in the user repository that match the information that you provided in the previous step.
   c. Select the check boxes for the user identities or group names that you want to assign to the sender role for the temporary destination prefix, and click **Next**. Note that you cannot assign users and groups to the creator role; it is assigned by default.
   d. Select the **Sender** icon for each user identity or group name that you want to add to the sender role. The icon changes from ☐ to ☑ to show that you have added the user or group to the access role for the resource.
   e. Click **Next**. A summary of your role type assignments is displayed.
   f. Optional: Click **Previous** to review and change your role type assignments. Make your changes on the Select role types page, and then click **Next**. Note that you cannot change the name of the temporary destination prefix.

   g. Click **Finish** to confirm your assignments. The role type assignments are saved to the master configuration, and the new assignments are displayed in the Temporary destination prefixes panel.
4. Save your changes to the master configuration.

Results

The selected users, groups, and group members are added to the sender role for the selected temporary destination prefix roles. The Manage access roles panel displays the new access roles.

Removing users and groups from temporary destination prefix roles:

Service integration bus security uses role-based authorization. When security is enabled, a temporary destination prefix role is used to authorize access to temporary destinations. The temporary destination prefix is used at runtime to create temporary destinations on the bus. By removing users and groups from temporary destination prefix roles for a selected bus, you can prevent selected users and groups from sending messages to temporary destinations on the bus.

About this task

In this task you use the administrative console to remove users, groups, and group members from the sender role for selected temporary destination prefixes. Note that you cannot use this task to remove users and groups from the creator role. If you want to remove the creator role from a user or group, refer to "Removing a temporary destination prefix."

Procedure
1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage temporary destination prefix access roles** The Temporary destination prefixes panel lists all the temporary destination prefixes defined for the selected bus.
3. Select one or more temporary destination prefixes to work with:
   - Click the name of a single temporary destination prefixes.
   - Select the check boxes next to multiple temporary destination prefixes, and click Manage Access Roles.

   The Temporary destination prefix access roles panel is displayed. The access roles information for each temporary destination prefix is displayed in a collapsed section.
4. Expand the header for a selected resource to show its role type assignments.
5. Select the users and groups that you want to remove from the sender role for the currently selected temporary destination prefix, and click **Remove**.
6. Save your changes to the master configuration.

Results

The selected users, groups, and group members are removed from the sender role for the selected temporary destination prefix. The Temporary destination prefix access roles panel is updated to show the changes to the access role assignments.

Removing a temporary destination prefix:

A temporary destination prefix is used by the service integration bus security model to determine what operations the creator of the temporary destination can perform. By removing a temporary destination prefix, you remove the creator role from the identity of the user that created the temporary destination.

About this task

A temporary destination prefix can have two authorization role types: creator and sender. In this task you use the administrative console to remove a selected temporary destination prefix from a selected bus, which removes the creator role from the user identity that created the temporary destination.

If you want to remove users and groups from the sender role for a temporary destination prefix, see "Removing users and groups from temporary destination prefix roles" on page 73.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage temporary destination prefix access roles** The Temporary destination prefixes panel lists all the temporary destination prefixes defined for the selected bus.
3. Select the temporary destination prefix you want to remove.
4. Click **Remove**.
5. Save your changes to the master configuration.

Results

The selected temporary destination prefix is removed, and the creator role is removed from the user identity that created the temporary destination.

Administering topic space root roles

Service integration bus security uses role-based authorization. When messaging security is enabled, groups of users require authority to send and receive messages from the topic space root in a publish/subscribe topic hierarchy. By adding and removing users and groups in topic space root roles, you can control access to the topic space root.

About this task

Topic space root (/) is also called the virtual root, and it is the highest level topic in a publish/subscribe topic hierarchy. The hierarchy itself is called the topic space, and it is a type of destination. Note that these tasks apply only to the topic space root; they do not apply to topics or a topic space. For information about administering topic access roles, refer to "Administering topic roles" on page 77, and for information about administering topic space access roles, see "Administering destination roles" on page 62.

You can add and remove users and groups in the sender and receiver roles for the topic space root. The topic space root can also inherit access in the sender and receiver roles from the topic space, providing the topic space is configured to inherit the default destination roles. For more information about topic inheritance, see Topic security.

For the topic space root roles to have an effect, the **Topic Access Check Required** check box must be selected in the topic space configuration. For more information, see Configuring bus destination properties.

Procedure

- "Listing users and groups in topic space root roles"
- "Adding users and groups to topic space root roles" on page 75
- "Removing users and groups from topic space root roles" on page 76

Listing users and groups in topic space root roles:

Service integration bus security uses role-based authorization. When messaging security is enabled, users and groups require authority to send and receive messages from the topic space root in a

publish/subscribe topic hierarchy. By listing the users and groups in topic space root roles, you can find out who has access to the topic space root for a selected topic space.

About this task

Topic space root (/) is the highest level topic in a publish/subscribe topic hierarchy. The hierarchy itself is called the topic space. In this task you use the administrative console wizard to list users and groups in topic space root roles for a selected root topic.

This task applies to the topic space root only; it does not apply to the topics within the topic space, or to the topic space. If you want to list users and groups in topic roles, refer to "Listing users and groups in topic roles" on page 78. If you want to list users and groups in a topic space (which is a type of destination), see "Listing users and groups in destination roles" on page 62.

The users and groups listed in this task have references in the service integration bus security configuration. The list does not include all users and groups that exist in the external user repository.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage topic access roles**

Results

The Topic space root panel lists all the users and groups in topic space root roles for the selected bus.

What to do next

You can administer the role type assignments for the users and groups displayed.

Adding users and groups to topic space root roles:

Service integration bus security uses role-based authorization. When messaging security is enabled, users and groups require authority to send and receive messages from the topic space root in a publish/subscribe topic hierarchy. By adding users and groups to topic space root roles, you control access to the root topic in a selected topic space.

Before you begin

- The users and groups you want to add to topic space root roles must exist in the user repository.
- Topic space root roles are effective only when the **Topic Access Check Required** setting is enabled in the configuration for a topic space. For more information, see Configuring bus destination properties.

About this task

Topic space root (/) is the highest level topic in a publish/subscribe topic hierarchy. The hierarchy itself is called the topic space. Note that this task applies only to the topic space root; it does not apply to adding users and groups to topics or a topic space. For information about adding users and groups to topic access roles, see "Adding users and groups to topic roles" on page 78, and for adding users and groups to topic space access roles, see "Adding users and groups to destination roles" on page 63.

You can add users and groups to the sender and receiver roles for the topic space root. The topic space root can also inherit access in the sender and receiver roles from the topic space, providing the topic space is configured to inherit the default destination roles. For more information about topic inheritance, see Topic security.

By default, a topic space does not contain a root topic. In this task you use an administrative console wizard to add a root topic to an existing topic space, retrieve the users and groups from the user repository that you want to assign to roles on the new root topic, and add them to the root topic.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage topic access roles**. The Topic spaces panel lists the topic spaces defined on the selected bus.
3. Select the name of the topic space where you want to add a new root topic. The Topics panel displays the selected topic space in a collapsible section.
4. Click **Add** to start the Security wizard:
   a. Identify the users or groups that you want to add to the sender and receiver roles for the new root topic:

      Users or Groups
         Select either **Users** or **Groups** to specify whether you want to grant roles to users or groups.

      Search pattern
         This field is mandatory. Specify a search string that is matched against user IDs or group names in the user repository. Only user IDs or group names that match the search pattern are retrieved, subject to the maximum number of search results. You can use wildcard characters in the search string.

      Maximum number of search results to display
         This field is mandatory. Specify the maximum number of user IDs or group names that you want the administrative console to display.
   b. Click **Next**. The wizard displays the new root topic, and lists the users IDs or group names in the user repository that match the information that you provided in the previous step.
   c. Select the check boxes next to the user IDs or group names that you want to assign to roles on the new root topic.
   d. Click **Next**. The wizard displays the topic role types that you can assign for the users or groups you selected in the previous step. Role types might already have been assigned for a specific user or group.
   e. Select the role types for the selected users or groups. For example, to assign a user to the sender role, select the **Sender** icon for the appropriate user ID. The icon changes from ☐ to ☑ to show that you have added the user or group to the access role for the resource.
   f. Click **Next**. A summary of your role type assignments for the root topic is displayed.
   g. Optional: If you want to change your assignments, click **Previous** to return to the Select role types page, change your assignments, and then click **Next**.
   h. Click **Finish** to confirm your assignments. The role type assignments are saved to the master configuration, and the new assignments are displayed in the Topics panel.
5. Save your changes to the master configuration.

Results

The selected users and groups are added to topic space root roles for the new root topic. The **Manage access roles** panel displays the new access role assignments.

Removing users and groups from topic space root roles:

Service integration bus security uses role-based authorization. When messaging security is enabled, users and groups require authority to send and receive messages from the topic space root in a

publish/subscribe topic hierarchy. By removing users and groups from topic space root roles, you prevent them from accessing the root topic in a selected topic space.

About this task

Topic space root (/) is the highest level topic in a publish/subscribe topic hierarchy. The hierarchy itself is called the topic space. Note that this task applies only to the topic space root; it does not apply to removing users and groups from topics or a topic space. For information about removing users and groups from topic access roles, see "Removing users and groups from topic roles" on page 79, and for removing users and groups from topic space roles, see "Removing users and groups from destination roles" on page 65.

In this task you use the administrative console to remove selected users and groups from the sender and receiver roles for the selected root topic.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage topic access roles**. The Topic spaces panel lists the topic spaces defined on the bus.
3. Select the topic space you want to work with. The selected topic space is displayed in the Topics panel. The root topic (/) is displayed by default.
4. Select the topic space root. The Topic access roles panel lists the role type assignments for the topic space root.
5. Select the names of the users, groups and group members that you want to remove from all role types for the selected root topic, and click **Remove**.
6. Save your changes to the master configuration.

Results

The selected users and groups are removed from all roles for the selected root topic. The Topic access roles panel is updated to show the changes to the access roles assignments.

Administering topic roles

Service integration bus security uses role-based authorization. When messaging security is enabled, users and groups require authority to access a topic in a publish/subscribe topic hierarchy. By adding and removing users and groups in topic roles, you can control access to the topic.

About this task

You use the administrative console to list, add and remove users and groups in the sender and receiver roles, and to define topic role inheritance. By default, a child topic inherits its topic roles from its parent topic. You can change the default roles for a particular topic by adding or removing topic roles at the topic level. You can also allow or block inheritance of topic roles at topic level.

You can add access roles for a topic before it exists. Topics are created at runtime only, and exist only for as long as they are active.

Procedure

- "Listing users and groups in topic roles" on page 78
- "Adding users and groups to topic roles" on page 78
- "Removing users and groups from topic roles" on page 79
- "Enabling topic role inheritance" on page 80
- "Disabling topic role inheritance" on page 81

Listing users and groups in topic roles:

Service integration bus security uses role-based authorization. When messaging security is enabled, users and groups require authority to access topics in a publish/subscribe topic hierarchy. By listing the users and groups that are members of topic roles for a selected topic, you can find out who has authority to send messages to and from the topic.

About this task

In this task you use the administrative console to list users and groups that have access roles for a selected topic in a selected topic space. The list includes users, groups and group members that have references in the role-based security configuration for service integration. It does not list all the users and groups that exist in the external user repository.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage topic access roles -> *topic_space_name*** > ***topic_name***. The Topics panel displays the information for the topic in a collapsed section.
3. Expand the section header to display the users and groups that are assigned to role types for the selected topic.

What to do next

You can add and remove users and groups in the topic roles for the selected topic.

Adding users and groups to topic roles:

Service integration bus security uses role-based authorization. When messaging security, and topic level authorization is enabled, users and groups must be authorized to access topics in a publish/subscribe topic hierarchy. By adding users and groups to topic roles, you control access to a topic in a selected topic space.

Before you begin

- The users and groups you want to add to topic space root roles must exist in the user repository.
- Topic roles are effective only when the **Topic Access Check Required** setting is enabled in the configuration for a topic space. For more information, see Configuring bus destination properties.

About this task

Topics are organized into one or more hierarchies within a topic space. If the **Topic Access Check Required** setting is enabled for the topic space, a user must have authorization to access the topic itself. You can add access roles to a topic before it is created at runtime. A topic inherits access roles from its parent unless you explicitly block the inheritance. For more information, see “Enabling topic role inheritance” on page 80.

In this task you use an administrative console wizard to add users or groups to the sender and receiver roles for a selected topic.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage topic access roles -> *topic_space_name*** > ***topic_name***. The Topic space root panel lists the users and groups that are assigned to role types for the selected topic.

3. Click **Add** to start the Security wizard:
   a. Provide the following information to enable the wizard to identify the users or groups that you want to add to role types for the selected topic:

      Resource
         Specify the name of the topic.

      Users or Groups
         Select either **Users** or **Groups** to specify whether you want to grant access roles to users or groups.

      Search pattern
         This field is mandatory. Specify a search string that is matched against user IDs or group names in the user repository. Only user IDs or group names that match the search pattern are retrieved, subject to the maximum number of search results. Wild card characters are allowed.

      Maximum number of search results to display
         This field is mandatory. Specify the maximum number of user IDs or group names you want the administrative console to display.
   b. Click **Next**. The wizard lists the users IDs or group names that match the information that you provided in the previous step.
   c. Select the check boxes for the user IDs or group names that you want to assign to roles for the selected topic.
   d. Click **Next**. The wizard lists the topic role types that you can assign for the users or groups you selected in the previous step. Role types might already have been assigned for a specific user or group.
   e. Select the role types for each of the selected users or groups. For example, to assign a user ID to the sender role, select the **Sender** icon for that user ID. The icon changes from ☐ to ☑ to show that you have added the user or group to the access role for the resource.
   f. Click **Next**. A summary of your role type assignments for the selected topic is displayed.
   g. Optional: If you want to change your assignments, click **Previous** to return to the Select role types step. Make changes to your assignments, and click **Next** to return to the Confirm step.
   h. Click **Finish** to confirm your assignments and save your changes to the master configuration.

Results

The updated role type assignments for the selected users or groups are displayed in the Topic access roles panel.

Removing users and groups from topic roles:

Service integration bus security uses role-based authorization. When messaging security is enabled, and the **Topic access check required** setting is enabled for the topic space, users and groups require authority to access a topic in the topic space. By removing users and groups from all topic roles for a selected topic, you prevent them from accessing the topic.

Before you begin

Topic roles are effective only when the **Topic Access Check Required** setting is enabled in the configuration for a topic space. For more information, see Configuring bus destination properties.

About this task

This task uses the administrative console to remove users and groups from both the sender and receiver roles for a selected topic in a selected topic space.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage topic access roles ->** ***topic_space_name*** > ***topic_name***. The Topic access roles panel is displayed. The information for the topic is displayed in a collapsed section.
3. Expand the section header to display the users and groups that are assigned to role types for the selected topic.
4. Select the users and groups that you want to remove from the sender and receiver roles for the selected topic, and click **Remove**.
5. Save your changes to the master configuration.

Results

The selected users and groups are removed from the sender and receiver roles for the selected topic. The Topic access roles panel is updated to show that the selected users and groups have no topic role type assignments.

Enabling topic role inheritance:

Service integration bus security uses role-based authorization. When messaging security, and topic level security are enabled, and users and groups require access in the sender and receiver roles to access a topic in a publish/subscribe topic hierarchy. By default, topics inherit these roles from the parent topic. If topic role inheritance has been disabled for a particular topic, you can restore it by using the administrative console.

Before you begin

You must ensure that the following conditions are met:

- Messaging security is enabled. For more information, see “Disabling bus security” on page 53.
- Topic level security is enabled for the topic space. Check the setting **Topic Access Check Required?** in the topic space destination configuration. For more information, see Configuring bus destination properties.

About this task

In this task you use the administrative console to restore topic role inheritance for selected topics. A topic can only inherit the sender and receiver roles from the parent topic in the topic hierarchy.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Authorization Policy] Manage topic access roles ->** ***topic_space_name*** > ***topic_name***. The Topic access roles panel lists users and groups that have been assigned role types for the selected topic.
3. Expand the topic name header to display details of the users and groups that have one or more access roles for this topic.
4. Select the **Inherit sender role from parent topic** check box.
5. Select the **Inherit receiver role from parent topic** check box.
6. Click **OK** to save your changes.
7. Save your changes to the master configuration.

Results

The select topic inherits access roles from the parent topic. The Topic access roles panel displays the inherited access roles for the topic.

Disabling topic role inheritance:

Service integration bus security uses role-based authorization. When messaging security, and topic level security are enabled, users and groups require access in the sender and receiver roles to access a topic in a publish/subscribe topic hierarchy. By default, topics inherit these roles from the parent topic. If you do not want topics to inherit topic roles from the parent topic in the topic hierarchy, you can override topic role inheritance by using the administrative console.

Before you begin

About this task

In this task you use the administrative console to prevent a selected topic from inheriting authorization roles from its parent topic.

Procedure

1. Log into the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage topic access roles -> *topic_space_name*** > ***topic_name***. The Topic access roles panel lists users and groups that have been assigned role types for the selected topic.
3. Expand the topic name header to display details of the users and groups that have access one or more access roles for the selected topic.
4. Clear the **Inherit sender role from parent topic** check box.
5. Clear the **Inherit receiver role from parent topic** check box.
6. Click **OK** to save your changes.
7. Save your changes to the master configuration.

Results

The selected topic cannot inherit access roles from its parent topic. The Topic access roles panel displays the changed access roles for the selected topic.

Removing access roles from unknown users and groups

Service integration bus security uses role-based authorization. Users and groups are assigned to access roles for specific bus resources. If a user or a group that has access roles is removed from the user repository, it becomes an unknown user. You can identify the unknown users and groups for a selected bus, and removes their access roles.

About this task

In this task, you use the administrative console to remove access roles from users and groups for unknown users and groups.

Procedure

1. Click **Service integration -> Buses -> *security_value* -> [Authorization Policy] Manage users and groups not known to the user repository** The Unknown users and groups panel lists all the users and groups that have access roles assigned to them, but they are not known in the user registry.
2. Select the check boxes next to multiple user and group names.
3. Click **Remove all roles**. The access role assignments for the selected users and groups are removed.

4. Save your changes to the master configuration.

Results

The access role assignments are removed from the selected users and groups. The Unknown users and groups panel is updated to show the changes to the role type assignments.

Administering permitted transports for a bus

Use these tasks to configure a transport policy for a service integration bus, and to administer the transports chains that remote applications clients can use to connect to a service integration bus.

Procedure

- "Configuring a transport policy for a bus"
- "Listing permitted transports for a bus" on page 83
- "Adding a permitted transport to a bus" on page 83
- "Removing a permitted transport from a bus" on page 84

Configuring a transport policy for a bus

By configuring a transport policy for the bus you can affect the security of messages in transit.

Before you begin

There are no prerequisites for this task.

About this task

The transport policy for a bus controls which transport mechanism remote application clients can use to connect to the bus.

Procedure

1. Log onto the administrative console.
2. Click **Service integration -> Buses -> *security_value***. The configuration panel for the selected bus is displayed
3. Choose one of the following transport policies for the bus:

   Allow the use of all defined transport channel chains
   Select this option to allow the bus to use unsecured ports.

   Restrict the use of defined transport channel chains to those protected by SSL
   Select this option to prevent the use of the InboundBasicMessaging port.

   Restrict the use of defined transport channel chains to the list of permitted transports
   Select this option if you want connecting client applications to use named transport channel chains. This provides the highest level of control over the use of transport channel chains.
4. Click **Apply**.
5. Save your changes to the master configuration.

Results

The transport policy you have configured for the selected bus controls how application clients connect to the bus.

What to do next

You can use the administrative console to add and remove transport chains in the list of permitted transports for the bus.

Listing permitted transports for a bus

Use this task to display a list of the transport chains that are available for a remote client application to use to connect to a selected service integration bus.

Before you begin

The transport policy for the bus must be set to the option **Restrict the use of defined transport channel chains to the list of permitted transports** for this task to have an effect. For more information about how to configure the transport policy for the bus, see "Configuring a transport policy for a bus" on page 82.

About this task

A permitted transport is a transport chain that a remote client application can use to connect to the bus. In this task, you use the administrative console to list all the permitted transports for a selected bus.

Procedure

1. Log onto the administrative console.
2. Click **Service integration -> Buses ->** ***security_value*** **-> [Additional Properties] Permitted transports**.

Results

The Permitted transports panel displays the list of permitted transports for the selected bus.

What to do next

You can use the administrative console to add and remove transport chains in the list of permitted transports to control which transport chains a remote client application can use to connect to the bus.

Adding a permitted transport to a bus

Use this task to make a new transport chain available to a remote client application to use to connect to a service integration bus.

Before you begin

The transport policy for the bus must be set to the option **Restrict the use of defined transport channel chains to the list of permitted transports** for this task to have an effect. For more information about how to configure the transport policy for the bus, see "Configuring a transport policy for a bus" on page 82.

About this task

A permitted transport is a transport chain that a remote client application can use to connect to the bus. If you want to allow remote client applications to connect to the bus by using a new transport chain, you must add the new transport chain to the list of permitted transports for the bus. In this task, you use the administrative console to add a new transport chain to the list of permitted transports for a selected bus.

Procedure

1. Log onto the administrative console.

2. Click **Service integration -> Buses -> *security_value* -> [Additional Properties] Permitted transports**. The Permitted transports panel displays the list of permitted transports for the selected bus.
3. Click **New**.
4. Select the name of the transport chain you want to add in the list box.
5. Click **OK**.
6. Save your changes to the master configuration.

Results

The new transport name is displayed in the list of permitted transports, and its is available for use by a remote client application to connect to the bus.

What to do next

You can use the administrative console to remove transport chains from a bus.

Removing a permitted transport from a bus

Use this task to remove a selected transport chain from the list of permitted transport chains for a selected service integration bus.

Before you begin

The transport policy for the bus must be set to the option **Restrict the use of defined transport channel chains to the list of permitted transports** for this task to have an effect. For more information about how to configure the transport policy for the bus, see "Configuring a transport policy for a bus" on page 82.

About this task

A permitted transport is a transport chain that a remote client application can use to connect to the bus. If you want to prevent a remote client application from using a particular transport chain to connect to the bus, you can remove the transport chain from the list of permitted transports for the bus. In this task, you use the administrative console to remove a transport chain from the list of permitted transports for a selected bus.

Procedure

1. Log onto the administrative console.
2. Click **Service integration -> Buses -> *security_value* -> [Additional Properties] Permitted transports**. The Permitted transports panel displays the list of permitted transports for the selected bus.
3. Select the name of the transport chain you want to remove.
4. Click **Delete**. The Permitted transports panel displays an updated list of permitted transports for the selected bus.
5. Save your changes to the master configuration.

Results

The selected transport chain is removed from the list of permitted transports for the selected bus, and a remote client can no longer use it to connect to the bus.

What to do next

You can use the administrative console to manage the transport policy for the bus.

Securing messages between messaging buses

Use these tasks to administer the access control security associated with sending messages between buses.

Procedure

- "Protecting messages transmitted between buses"
- "Administering access to foreign destinations"

Protecting messages transmitted between buses

Use this task to protect the integrity of the data that is transmitted between secured linked service integration buses.

Before you begin

- Review the information in the topics Secure transport configuration requirements and Configuring transport chains.
- Configure a transport policy for the bus, as described in the topic "Configuring a transport policy for a bus" on page 82.

About this task

In this task, you configure transport chains for the remote bus to ensure that only secured inbound transport chains can contact the messaging engines on the server. The remote bus might be a foreign bus, or an WebSphere MQ link.

Procedure

1. Ensure that the linked buses are secured. For further information, see "Securing buses" on page 44.
2. Configure the Target inbound transport chain property for the foreign bus, or WebSphere MQ link, as appropriate:
   - For connections to a foreign bus, see Connecting service integration buses to use point-to-point messaging or Connecting service integration buses to use publish/subscribe messaging
   - For connections to WebSphere MQ, refer to Creating a new WebSphere MQ link.

Results

You have configured the Target inbound transport chains for the remote service integration bus.

What to do next

Administering access to foreign destinations

Use these tasks to administer the access control security associated with sending messages to foreign bus destinations.

About this task

To define the authentication information used in the access control checks that are performed when a message is sent to a destination in a foreign bus, complete the following steps:

Procedure

- Define the required authorization permissions to allow a sender to access a destination on a foreign bus. These can be defined by using either a foreign bus definition, as described in "Administering foreign bus roles" on page 67, or a foreign bus destination definition, as described in "Administering destination roles" on page 62. These permissions are used when the message is sent, to check that the sender is allowed to access the foreign bus.

- The administrator of the foreign bus must define the required authorization permissions to allow the messages to access the destination on the foreign bus. These permissions are used when the message enters the foreign bus.
- Define the authorization permissions to allow any messages entering your bus from the foreign bus to access the required destinations.

Securing access to a foreign bus

You can secure the link between a local bus and a foreign bus.

Before you begin

Before you can secure the link between a local bus and a foreign bus, there must be foreign bus connection on the local bus, and therefore a link between the buses.

About this task

This task summarizes the significant tasks to secure the link between a local bus and a foreign bus. For more general information about service integration bus security, see “Securing service integration” on page 43.

When you create a foreign bus connection, there are some options to secure the connection during that procedure.

Procedure

1. Enable security on the service integration bus and the foreign bus. See “Securing buses” on page 44.
2. Secure the link between the buses. See “Securing messages between messaging buses” on page 85.
3. Grant access to the local bus for users who will be sending messages to the foreign bus. See “Administering the bus connector role” on page 57.
4. Grant access to the foreign bus for users who will be sending messages to it. See “Administering foreign bus roles” on page 67.
5. Optional: Give users access to foreign or alias destinations that will forward messages to a foreign bus. See “Administering destination roles” on page 62.

Securing links between messaging engines

For a mixed-version bus, when security is enabled, you must define an inter-engine authentication alias so that the messaging engines can establish trust.

Before you begin

Ensure that the user ID that you intend to use for the inter-engine authentication alias meets the following conditions:

- It exists in the user registry.
- It is used only for messaging engine to messaging engine authentication.
- It has not been added to the bus connector access role.

If you have a secure bus where all bus members are at Version 7.0 or later, trust between Version 7.0 or later messaging engines is established by using a Lightweight Third Party Authentication (LTPA) token, and you do not need to perform this task.

About this task

If you have a secure, mixed-version bus, you must define an inter-engine authentication alias to prevent unauthorized messaging engines from establishing a connection. Messaging engines use the inter-engine authentication alias to establish trust in the following scenarios:

- A WebSphere Application Server Version 6 messaging engine initiates a link with a Version 7.0 or later messaging engine.
- A Version 7.0 or later messaging engine initiates a link with a Version 6 messaging engine.

If you add a server or cluster as a bus member, if that action creates a mixed-version bus, you define an inter-engine authentication alias during that task, and you do not need to perform this task.

Procedure

1. In the navigation pane, click **Service integration -> Buses -> *security_value***. The bus security configuration panel for the corresponding bus is displayed.
2. In the **Inter-engine authentication alias** field, select an authentication alias.
3. Click **OK**.
4. Save your changes to the master configuration.

Results

You have selected an inter-engine authentication alias for the bus to use in establishing trust between mixed-version messaging engines.

What to do next

If you require additional security, you can configure the SSL certificate stores to restrict objects that can make an SSL connection, and thereby connect to the bus. For more information see Creating a Secure Sockets Layer configuration.

Controlling which foreign buses can link to your bus

Use this task to control which foreign buses are allowed to link to your bus.

About this task

When messaging security is enabled, it is important that only authorized foreign buses are allowed to link to your bus.

To control which foreign buses can create a link to your bus, complete the following steps:

Procedure

1. Set the authentication alias property on the foreign bus connection to be used for authentication of the foreign bus joining your bus, as described in Connecting service integration buses to use point-to-point messaging or Connecting service integration buses to use publish/subscribe messaging.
2. If you require extra security, configure the SSL certificate stores to restrict who can make an SSL connection, and hence link to the bus. For more information, see Creating a Secure Sockets Layer configuration.

Securing database access

You can protect the data store from access by unauthorized users.

About this task

To secure access to the data store, you must configure a user name and password on the data store for the messaging engine. If you want to apply additional levels of security such as encrypting the contents of the database, use the specific security features provided by your database.

Securing mediations

Use the following tasks to secure mediations at an operations level. For example, a mediation inherits its identity from a the messaging engine, but you might want to specify an alternative identity for the mediation to use.

Configuring an alternative mediation identity for a mediation handler

Use this task to configure an alternative mediation identity for a mediation handler

About this task

By default, a mediation inherits the identity used by the messaging engine. In some cases, you might want to specify an alternative identity for a mediation handler to use. For example, for a single mediation that sends messages to a destination. To do this, you specify a "run-as" identity for the mediation handler at deployment, and map the mediation handler to an identity other than the default mediation identity by using a role name. Follow the steps below to specify an alternative mediation identity:

Procedure

1. Package your mediation handler as an EAR file.
2. Edit the deployment descriptor file to define the roles. For more information, see Configuring programmatic logins for Java Authentication and Authorization Service.
3. Assign users to the role. For more information, see “Mapping users to RunAs roles using an assembly tool” on page 113 and “Securing applications during assembly and deployment” on page 118.
4. Deploy the mediation handler in WebSphere Application Server, and assign users to the RunAs role. For more information, see “Assigning users to RunAs roles” on page 112. You can confirm the mappings of users to roles, add new users and groups, and modify existing information during this step. For more information, see “Deploying secured applications” on page 120.

Example

What to do next

Next, you are ready to authorize mediations to access destinations. For more information, see “Administering authorization permissions” on page 56.

Configuring the bus to access secured mediations

Use this task to ensure that the service integration bus is authorized to access secured mediations.

Before you begin

The mediation is secured by using a Java Platform, Enterprise Edition (Java EE) Connector Architecture authentication alias. For information about creating a Java EE authentication alias, see Managing Java 2 Connector Architecture authentication data entries for JAAS.

About this task

To configure the bus to access a secured mediation, you must add the mediation authentication alias for the secured mediation to the properties for the bus:

- If the bus has a Version 6 bus member, you must provide the principal and its associated password.

- If the bus has WebSphere Application Server Version 7.0 or later bus members only, you need only provide the principal.

Procedure

1. Log into the navigation pane.
2. Click **Service integration -> Buses -> *security_value***. The bus security configuration panel is displayed.
3. In the **Mediations authentication alias** field, select the principal for the mediation, and its associated password if required.
4. Click **OK**.
5. Save your changes to the master configuration.

Results

The selected bus is configured to access secured mediations.

What to do next

You can assign security roles to your mediation handlers to protect them from use by unauthorized users. For more information, see "Deploying secured applications" on page 120.

Configuring a bus to run mediations in a multiple security domain environment

Use this task to configure a secured bus so that it can run mediations successfully on bus members in different security domains.

Before you begin

The secured bus must be configured to use a non-global security domain. For more information about securing buses by using multiple security domains, refer to "Securing buses" on page 44.

About this task

If your bus topology has bus members in different security domains, you can configure the bus to allow mediations to run under the server identity. This means that a mediation can run on any server in any domain. You do not have to add a dedicated user ID for each mediation to the user repository, or maintain a mediation authentication alias.

Use the administrative console to configure a secured bus to run mediations successfully as follows:

Procedure

1. In the navigation pane, click **Service integration -> Buses -> *security_value***. The security settings for the selected bus are displayed.
2. Check the option **Use the Server ID when running mediations**.
3. Click **Apply**.
4. Save your changes to the master configuration.

Results

You have configured the bus to run mediations successfully across servers in multiple security domains.

What to do next

You can use the administrative console to control access to the bus by administering users and groups in the bus connector role.

Auditing the service integration security infrastructure

Security auditing is an important part of the security infrastructure. Security auditing provides a mechanism for auditable events to be tracked and archived while ensuring the integrity of the records.

Before you begin

Before enabling the security auditing subsystem for service integration, you must enable global security in your environment.

About this task

The primary responsibility of the security infrastructure is to prevent unauthorized usage of resources. Security auditing has two primary goals:

- Confirming the effectiveness and integrity of the existing security configuration.
- Identifying areas where improvement to the security configuration might be needed.

Security auditing achieves these goals by providing an infrastructure that you can use to implement your code to capture and store supported security auditable events. All code other than the Java enterprise application code is considered to be trusted. Each time an enterprise application accesses a resource, any internal application server process that has audit points that are added within their code can be recorded as an auditable event. The security auditing subsystem captures the following types of auditable events:

- Authentication
- Authorization
- Principal and Credential Mapping
- Key management
- System management
- Security policy management
- Audit policy management
- Administrative configuration management
- Administrative runtime management
- User registry and identity management
- Password changes
- Delegation

These types of events can be recorded into audit log files. The audit log flies can be signed and encrypted to ensure data integrity. These audit log files can be analyzed to discover breaches over the existing security mechanisms and to discover potential weaknesses in the current security infrastructure. Security event audit records are also useful for providing evidence, accountability, and vulnerability analysis. The security auditing configuration provides four default filters, a default audit service provider, and a default event factory. This following steps describe how to customize your security auditing subsystem. Additional information specific to messaging is included in the step description where appropriate.

Procedure

1. Enabling the security auditing subsystem

   Security auditing is not performed unless the audit security subsystem has been enabled. Global security must be enabled for the security audit subsystem to function, as no security events occur if global security is not also enabled.

   To allow messaging security events to be audited, audit security must be enabled:

   a. For each bus to be audited, click **Service integration -> Buses -> *security_value***, and select the **Enable the auditing service for this bus** check box.

b. For publish/subscribe messaging, also on each topic space on the bus being audited, click **Service integration -> Buses -> *bus_name* -> [Destination resources] Destinations -> *topic_space_name***, and select the **Enable the auditing service for this topic space** check box.

2. Auditor role

   A user with the auditor role is required to enable and configure the security auditing subsystem. It is important to require strict access control for security policy management. The auditor role provides granularity to support separation of the auditing role from the authority of the administrator.

3. Creating security auditing event type filters

   You can configure event type filters to only record a specific subset of auditable event types in your audit logs. Filtering the event types that are recorded makes for simpler analysis of your audit records by ensuring the records to only display what you selected as important for your an environment.

   The audit events that can be configured for messaging are:

   SECURITY_AUTHN
   This event is produced when the identity of a messaging client or messaging engine connecting to a messaging bus is authenticated.

   SECURITY_AUTHZ
   This event is produced when the identity of a messaging client is checked for access authority to a bus or a message queue when sending, directly or by publication, or receiving messages, directly or by subscription.

   SECURITY_AUTHN_TERMINATE
   This event is produced when the connection between a messaging client or messaging engine and a messaging bus is terminated.

   SECURITY_MGMT_CONFIG
   This event is produced when a messaging client changes the contents of a service data object (SDO) repository in an import or remove operation.

   You can create event filters for each permutation of an event and its possible outcomes such as SUCCESS, DENIED, or error conditions of different levels of severity.

   See messaging security events for more information on which messaging security audit events are produced and when they are produced.

4. Configuring audit service providers for security auditing

   The audit service provider is used to format the audit data object that is passed to it before outputting the data to a repository.

5. Configuring audit event factories for security auditing

   The audit event factory gathers the data associated with the auditable events and creates an audit data object. The audit data object is then sent to the audit service provider to be formatted and recorded to the repository.

6. Protecting your security audit data It is important for the recorded audit data to be both secured and with the data integrity ensured. To ensure that access to the data is restricted and secure, you can encrypt and sign your audit data.

7. Configuring security audit subsystem failure notifications

   You can enable notifications to generate alerts when the security auditing subsystem experiences a failure. Notifications can be configured to record an alert in the audit logs or can be configured to send an alert through email to a specified list of recipients.

Results

After successfully completing this task, you audit data is recorded for the selected auditable events that were specified in the configuration.

What to do next

After configuring security auditing, you can analyze your audit data for potential weaknesses in the current security infrastructure and to discover security breaches that might have occurred over the existing security mechanisms.

Chapter 9. Securing Session Initiation Protocol (SIP) applications

This page provides a starting point for finding information about SIP applications, which are Java programs that use at least one Session Initiation Protocol (SIP) servlet written to the JSR 116 specification.

SIP is used to establish, modify, and terminate multimedia IP sessions including IP telephony, presence, and instant messaging.

Securing SIP applications

You can apply digest authentication and Trust Association Interceptor (TAI) for a SIP application by applying Lightweight Directory Access Protocol (LDAP) security to the application.

Before you begin

Before you can apply security, you must first deploy an application that has been developed to support security (with the `web.xml` file configured for security) and roles. The following software must also be installed:

1. Install a supported LDAP server. For a list of supported LDAP servers, see the IBM website for WebSphere Application Server supported hardware, software, and APIs.

   IBM Tivoli Directory Server users can choose IBM Tivoli Directory Server as the directory type for better performance.

   Note: IBM SecureWay Directory Server was renamed to IBM Tivoli Directory Server in WebSphere Application Server version 6.1.
2. Set up and activate Lightweight Third Party Authentication.

About this task

To apply LDAP security to a SIP application, click **Applications** > **Enterprise Applications** > ***applicationName*** and complete the following steps:

Procedure

1. Click **Detail Properties** > **Security role to user/group mapping**.
2. Check **All Authenticated**.
3. Save all changes.
4. Restart the server.

Configuring security for the SIP container

This section provides instructions specific to security for the SIP container.

Before you begin

You must select a proper Lightweight Directory Access Protocol (LDAP) repository and activate security on WebSphere Application Server before SIP digest authentication can be activated. See the following topics in the information center:

- Selecting a registry or repository
- Authenticating users

© IBM Corporation 2009

About this task

To configure security based on the LDAP, you can use digest authentication for your supported LDAP server. See the information center topic on configuring digest authentication for SIP applications.

To define an LDAP connection between WebSphere Application Server and LDAP, use the security wizard. It can also be defined by selecting it from available realms and defining the proper connection properties to connect LDAP.

To set up a trust association interceptor (TAI), you must specify the trust information for any reverse security proxy servers. See the information center topic on trust association interceptor settings.

Configuring digest authentication for SIP

You can configure SIP digest authentication settings to allow the SIP container to authenticate secured applications.

Before you begin

Ensure that clusters and stand-alone servers are created and federated.

About this task

The SIP container supports digest authentication. When this type of authentication is used, the client does not send a clear text password to the server. Instead, SIP authenticates each request using user data from a Lightweight Directory Access Protocol (LDAP) server. A component that uses LDAP for authentication verifies that the response that the client provides equals the response that the component calculates using LDAP data, which authorizes the request.

gotcha: You must select a proper LDAP repository and activate security on WebSphere Application Server before SIP digest authentication can be activated. See the information center topic on configuring security for the SIP container.

Procedure

1. From the administrative console, click **Security** > **Global security** > **Web and SIP security** > **SIP digest authentication**.
2. Specify a value for one or more settings. See the SIP digest authentication settings topic for more details.
3. Click **OK**.
4. Restart the application server.

Results

The container uses digest authentication to authenticate SIP applications.

SIP digest authentication settings

Use this page to configure Session Initiation Protocol (SIP) digest authentication settings; these settings allow the SIP container to authenticate secured applications.

To view this administrative console page, click **Security** > **Global Security** > **Authentication** > **Web and SIP Security** > **SIP digest authentication**.

Enable digest authentication integrity:

Specifies the authentication integrity (auth-int) quality of protection (QOP) for digest authentication. Digest authentication defines two types of QOP: auth and auth-int. By default, basic authentication (auth) is used. If the value is set to `True`, the auth-int QOP is used, which is the highest level of protection.

Information	Value
Data type	Boolean
Default	True

Enable SIP basic authentication:

Specifies the SIP container supports basic authentication. If the value is set to `True`, requests that have the `Authorization` header with basic schema are authenticated by the application server. Otherwise, digest authentication is required.

Information	Value
Data type	Boolean
Default	False

Enable multiple use of nonce:

Specifies whether to enable multiple uses of the same nonce. If you use the same nonce more than once, then less system resources are required, however, your system is not as secure.

Information	Value
Data type	Boolean
Default	False

Limit nonce maximum age:

Specifies whether to enable the nonce maximum age. If you do not disable this parameter, the nonce never expires.

Information	Value
Data type	Boolean
Default	True

Nonce maximum age:

Specifies the amount of time, in milliseconds, for which a nonce is valid. If the value is set to `1`, the amount of time is considered to be infinite.

Information	Value
Data type	Integer
Default	1

LDAP cache clean intervals:

Specifies the amount of time that must expire, in minutes, before the LDAP cache is cleaned.

Information	Value
Data type	Integer
Default	120

LDAP password attribute name:

Specifies the LDAP attribute name that stores the user password .

Information	Value
Data type	String
Default	Empty string

User cache clean intervals:

Specifies the amount of time that must expire, in minutes, before the security subject cache is cleaned.

Information	Value
Data type	Integer
Default	15

Digest password server class:

Specifies the Java class name that implements the PasswordServer interface.

Information	Value
Data type	String
Default	Empty string

Hashed credentials:

Specifies the name of the LDAP field that contains the hashed credentials. If a value is specified for this setting, then this setting overrides the pws_atr_name setting.

LDAP servers automatically provide password support. Unless you enable the LDAP server to use hashed values, the LDAP server stores user passwords and then the request processing component uses these passwords to validate a request. Because this method of authentication exposes user passwords to potential internet theft, you should enable the use of hashed credentials to authenticate a request.

When you enable the use of hashed credentials, the LDAP server stores a hash value for the user, password and realm information. The SIP container then requests this hash value from the LDAP server instead of asking for a user password. This methodology protects the passwords even if the hash data is compromised through internet theft. However, this methodology has the following limitations:

- The LDAP attribute must store a byte value or a string value. Other attribute types are not supported.
- All of your applications must share the same realm, or you must define a different attribute for each realm.
- The hash function might be different than MD5. In this situation, the SIP container sends a algorithm that is different from the calculated value for the attribute. When this situation occurs, user authentication might fail even if the user provided the proper credentials.

To enable the LDAP server to use hashed credentials, you must define the following two settings:

- Hashedcredentials=value, where value is the name of LDAP attribute that stores the hash value for user, password, and realm.
- Hashedrealm=value, where value is the realm, on which the hashed value is calculated.

Information	Value
Data type	String
Default	Empty string

Hashed realm:

Specifies the realm for hashed credentials, if the hashed credentials setting is enabled.

Information	Value
Data type	String
Default	Empty string

Developing a custom trust association interceptor

When you develop Session Initiation Protocol (SIP) applications, you can create a custom trust association interceptor (TAI).

Before you begin

You may want to familiarize yourself with the general TAI information contained in the Trust Associations documentation. Developing a SIP TAI is similar to developing any other custom interceptors used in trust associations. In fact, a custom TAI for a SIP application is actually an extension of the trust association interceptor model. Refer to the Developing a custom interceptor for trust associations section for more details.

About this task

TAI can be invoked by a SIP servlet request or a SIP servlet response. To implement a custom SIP TAI, you need to write your own Java class.

Procedure

1. Write a Java class that extends the `com.ibm.wsspi.security.tai.BaseTrustAssociationInterceptor` class and implements the `com.ibm.websphere.security.tai.SIPTrustAssociationInterceptor` interface. Those classes are defined in the *WASProductDir*`/plugins/com.ibm.ws.sip.container_1.0.0.jar` file, where *WASProductDir* is the fully qualified path name of the directory in which WebSphere Application Server is installed.
2. Declare the following Java methods:

   `public int initialize(Properties properties) throws WebTrustAssociationFailedException;`
   This is invoked before the first message is processed so that the implementation can allocate any resources it needs. For example, it could establish a connection to a database. `WebTrustAssociationFailedException` is defined in the *WASProductDir*`/plugins/com.ibm.ws.runtime_1.0.0.jar` file. The value of the `properties` argument comes from the **Custom Properties** set in this step.

   `public void cleanup();`
   This is invoked when the TAI should free any resources it holds. For example, it could close a connection to a database.

   `public boolean isTargetProtocolInterceptor(SipServletMessage sipMsg) throws WebTrustAssociationFailedException;`
   Your custom TAI should use this method to handle the `sipMsg` message. If the method returns `false`, WebSphere ignores your TAI for `sipMsg`.

   `public TAIResult negotiateValidateandEstablishProtocolTrust (SipServletRequest req, SipServletResponse resp) throws WebTrustAssociationFailedException;`
   This method returns a TAIResult that indicates the status of the message being processed and a user ID or the unique ID for the user who is trying to authenticate. If authentication succeeds, the TAIResult should contain the status HttpServletResponse.SC_OK and a principal. If authentication fails, the TAIResult should contain a return code of HttpServletResponse.SC_UNAUTHORIZED (401), SC_FORBIDDEN (403), or

SC_PROXY_AUTHENTICATION_REQUIRED (407). This only indicates whether or not the container should accept a message for further processing. To challenge an incoming request, the TAI implementation must generate and send its own SipServletResponse containing a challenge. The exception should be thrown for internal TAI errors. Table 12 describes the argument values and resultant actions for the negotiateValidateandEstablishProtocolTrust method.

Table 12. Description of negotiateValidateandEstablishProtocolTrust arguments and actions.

This table provides a description of negotiateValidateandEstablishProtocolTrust arguments and actions

Argument or action	For a SIP request	For a SIP response
Value of `req` argument	The incoming request	Null
Value of `resp` argument	Null	The incoming response
Action for valid response credentials	Return TAIResult.status containing SC_OK and a user ID or unique ID	Return TAIResult.status containing SC_OK and a user ID or unique ID
Action for incorrect response credentials	Return the TAIResult with the 4*xx* status	Return the TAIResult with the 4*xx* status

The sequence of events is as follows:

a. The SIP container maps initial requests to applications by using the rules in each applications deployment descriptor; subsequent messages are mapped based on JSR 116 mechanisms.

b. If any of the applications require security, the SIP container invokes any defined TAI implementations for the message.

c. If the message passes security, the container invokes the corresponding applications.

Your TAI implementation can modify a SIP message, but the modified message will not be usable within the request mapping process, because it finishes before the container invokes the TAI.

The com.ibm.wsspi.security.tai.TAIResult class, defined in the `*WASProductDir*/plugins/com.ibm.ws.runtime_1.0.0.jar` file, has three static methods for creating a TAIResult. The TAIResult `create` methods take an int type as the first parameter. WebSphere Application Server expects the result to be a valid HTTP request return code and is interpreted as follows:

If the value is HttpServletResponse.SC_OK, this response tells WebSphere Application Server that the TAI has completed its negotiation. The response also tells WebSphere Application Server use the information in the TAIResult to create a user identity.

The created TAIResults have the meanings shown in Table 13.

Table 13. Meanings of TAIResults.

This table lists the meanings of TAIResults

TAIResult	Explanation
public static TAIResult create(int status);	Indicates a status to WebSphere Application Server. The status should not be SC_OK because the identity information is provided.
public static TAIResult create(int status, String principal);	Indicates a status to WebSphere Application Server and provides the user ID or the unique ID for this user. WebSphere Application Server creates credentials by querying the user registry.
public static TAIResult create(int status, String principal, Subject subject);	Indicates a status to WebSphere Application Server, the user ID or the unique ID for the user, and a custom Subject. If the Subject contains a Hashtable, the principal is ignored. The contents of the Subject becomes part of the eventual user Subject.

public String getVersion();
: This method returns the version number of the current TAI implementation.

public String getType();
: This method's return value is implementation-dependent.

3. Compile the implementation after you have implemented it. For example: `/opt/WebSphere/AppServer/java/bin/javac -classpath /opt/WebSphere/AppServer/plugins/com.ibm.ws.runtime_1.0.0.jar;/opt/WebSphere/AppServer/dev/JavaEE/j2ee.jar;/opt/WebSphere/AppServer/plugins/com.ibm.ws.sip.container_1.0.0.jar myTAIImpl.java`
   a. For each server within a cluster, copy the class file to a location in the WebSphere class path (preferably the *WASProductDir*/plugin/ directory).
   b. Restart all the servers.
4. Delete the default WebSEAL interceptor in the administrative console and click **New** to add your custom interceptor. Verify that the class name is dot-separated and appears in the class path.
5. Click the **Custom Properties** link to add additional properties that are required to initialize the custom interceptor. These properties are passed to the `initialize(Properties properties)` method of your implementation when it extends the `com.ibm.websphere.security.WebSphereBaseTrustAssociationInterceptor` as described in the previous step.
6. Save and synchronize (if applicable) the configuration.
7. Restart the servers for the custom interceptor to take effect.

Chapter 10. Securing web applications

This page provides a starting point for finding information about web applications, which are comprised of one or more related files that you can manage as a unit, including:

- HTML files
- Servlets can support dynamic web page content, provide database access, serve multiple clients at one time, and filter data.
- Java ServerPages (JSP) files enable the separation of the HTML code from the business logic in web pages.

IBM extensions to the JSP specification make it easy for HTML authors to add the power of Java technology to web pages, without being experts in Java programming. More introduction...

Web application security components and settings

Web component security

A web module consists of servlets, JavaServer Pages (JSP) files, server-side utility classes, static web content, which includes HTML, images, sound files, cascading style sheets (CSS), and client-side classes or applets. You can use development tools such as Rational® Application Developer to develop a web module and enforce security at the method level of each web resource.

You can identify a web resource by its URI pattern. A web resource method can be any HTTP method (GET, POST, DELETE, PUT, for example). You can group a set of URI patterns and a set of HTTP methods together and assign this grouping a set of roles. When a web resource method is secured by associating a set of roles, grant a user at least one role in that set to access that method. You can exclude anyone from accessing a set of web resources by assigning an empty set of roles. A servlet or a JavaServer Pages (JSP) file can run as different identities before invoking another enterprise bean component. All the secured web resources require the user to log in by using a configured login mechanism. Three types of web login authentication mechanisms are available: basic authentication, form-based authentication and client certificate-based authentication.

In WebSphere Application Server Version 6.1, a portlet resource that is part of a web module can also be protected when it is accessed directly through URL. The protection is similar to other web based resources. For more information, see “Portlet URL security” on page 39.

For more detailed information on web security, see the product architectural overview article.

Securing web applications using an assembly tool

You can use three types of web login authentication mechanisms to configure a web application: basic authentication, form-based authentication and client certificate-based authentication. Protect web resources in a web application by assigning security roles to those resources.

About this task

To secure web applications, determine the web resources that need protecting and determine how to protect them.

Note: This procedure might not match the steps that are required when using your assembly tool, or match the version of the assembly tool that you are using. You should follow the instructions for the tool and version that you are using.

The following steps detail securing a web application using an assembly tool:

© Copyright IBM Corp. 2012

Procedure

1. In an assembly tool, import your web application archive (WAR) file or an application archive (EAR) file that contains one or more Web modules.
2. In the Project Explorer folder, locate your web application.
3. Right-click the deployment descriptor and click **Open With > Deployment Descriptor Editor**. The Deployment Descriptor window opens. To see online information about the editor, press F1 and click the editor name. If you select a web application archive (WAR) file, a web deployment descriptor editor opens. If you select an enterprise application (EAR) file, an application deployment descriptor editor opens.
4. Create security roles either at the application level or at the web module level. If a security role is created at the web module level, the role also displays in the application level. If a security role is created at the application level, the role does not display in all of the web modules. You can copy and paste a security role at the application level to one or more Web module security roles.
   - Create a role at a Web-module level. In a web deployment descriptor editor, click the Security tab. Under **Security Roles**, click **Add.**. Enter the security role name, describe the security role, and click **Finish**.
   - Create a role at the application level. In an application deployment descriptor editor, click the Security tab. Under the list of security roles, click **Add**. In the Add Security Role wizard, name and describe the security role and then click **Finish**.
5. Create security constraints. Security constraints are a mapping of one or more web resources to a set of roles.
   a. On the Security tab of a web deployment descriptor editor, click **Security Constraints**. On the Security Constraints tab, you can do the following actions:
      - Add or remove security constraints for specific security roles.
      - Add or remove web resources and their HTTP methods.
      - Define which security roles are authorized to access the web resources.
      - Specify None, Integral, or Confidential constraints on user data.

        None The application does not require transport guarantees.

        Integral
        Data cannot be changed in transit between the client and the server.

        Confidential
        Data content cannot be observed while it is in transit.

        Integral and Confidential usually require the use of SSL. When deploying applications that are available over public networks, specify Confidential for your web applications constraints

   b. Under Security Constraints, click **Add**.

   c. Under Constraint name, specify a display name for the security constraint and click **Next**.

   d. Type a name and description for the web resource collection.

   e. Select one or more HTTP methods. The HTTP method options are: GET, PUT, HEAD, TRACE, POST, DELETE, and OPTIONS.

   f. Beside the Patterns field, click **Add**.

   g. Specify a URL Pattern. For example, type - `/*`, `*.jsp`, `/hello`. Consult the Servlet specification Version 2.4 for instructions on mapping URL patterns to servlets. The security runtime uses the exact match first to map the incoming URL with URL patterns. If the exact match is not present, the security runtime uses the longest match. The wild card (`*.,*.jsp`) URL pattern matching is used last.

   h. Click **Finish**.

   i. Repeat these steps to create multiple security constraints.
6. Map security-role-ref and role-name elements to the role-link element. During the development of a web application, you can create the security-role-ref element. The security-role-ref element contains only the role-name field. The role-name field contains the name of the role that is referenced in the servlet or JavaServer Pages (JSP) code to determine if the caller is in a specified role. Because

security roles are created during the assembly stage, the developer uses a logical role name in the Role-name field and provides enough description in the Description field for the assembler to map the role actual. The Security-role-ref element is at the servlet level. A servlet or JavaServer Pages (JSP) file can have zero or more security-role-ref elements.

a. Go to the References tab of a web deployment descriptor editor. On the References tab, you can add or remove the name of an enterprise bean reference to the deployment descriptor. You can define five types of references on this tab:
   - EJB reference
   - Service reference
   - Resource reference
   - Message destination reference
   - Security role reference
   - Resource environment reference

b. Under the list of Enterprise JavaBeans (EJB) references, click **Add**.

c. Specify a name and a type for the reference in the **Name** and **Ref Type** fields.

d. Select either **Enterprise Beans in the workplace** or **Enterprise Beans not in the workplace**.

e. Optional: If you select **Enterprise Beans not in the workplace**, select the type of enterprise bean in the **Type** field. You can specify either an entity bean or a session bean.

f. Optional: Click **Browse** to specify values for the local home and local interface in the **Local home** and **Local** fields before you click **Next**.

g. Map every role-name that is used during development to the role using the previous steps. Every role name that is used during development maps to the actual role.

7. Specify the RunAs identity for servlets and JSP files. The RunAs identity of a servlet is used to invoke enterprise beans from within the servlet code. When enterprise beans are invoked, the RunAs identity is passed to the enterprise bean for performing an authorization check on the enterprise beans. If the RunAs identity is not specified, the client identity is propagated to the enterprise beans. The RunAs identity is assigned at the servlet level.

   a. On the Servlets tab of a web deployment descriptor editor, under **Servlets and JSP**, click **Add**. The Add Servlet or JSP wizard opens.

   b. Specify the servlet or JavaServer Pages (JSP) file settings, including the name, initialization parameters, and URL mappings and click **Next**.

   c. Specify the class file destination.

   d. Click **Next** to specify additional settings or click **Finish**.

   e. Click **Run As** on the **Servlets** tab, select the security role and describe the role.

   f. Specify a RunAs identity for each servlet and JSP file that is used by your web application.

8. Configure the login mechanism for the web module. This configured login mechanism applies to all the servlets, JavaServer Pages (JSP) files and HTML resources in the web module.

   a. Click the **Pages** tab of a web deployment descriptor editor and click **Login**. Select the required authentication method. Available method values include: Unspecified, Basic, Digest, Form, and Client-Cert.

   b. Specify a realm name.

   c. If you select the Form authentication method, select a login page and an error page web address. For example, you might use `/login.jsp` or `/error.jsp`. The specified login and error pages are present in the `.war` file.

   d. Install the client certificate on the browser web client and place the client certificate in the server trust keyring file, if ClientCert certificate is selected. The public certificate of the clients certificate authority must be placed in the servers RACF keyring. If the registry is a local OS registry, use the RACDCERT MAP or the equivalent System Authorization Facility (SAF) command to enable an MVS identity creation using the client certificate.

9. Close the deployment descriptor editor and, when prompted, click **Yes** to save the changes.

Results

After securing a web application, the resulting web application archive (WAR) file contains security information in its deployment descriptor. The web module security information is stored in the `web.xml` file. When you work in the web deployment descriptor editor, you also can edit other deployment descriptors in the web project, including information on bindings and IBM extensions in the `ibm-web-bnd.xmi` and `ibm-web-ext.xmi` files.

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

- For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.
- For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

What to do next

After using an assembly tool to secure a web application, you can install the web application using the administrative console. During the web application installation, complete the steps in “Deploying secured applications” on page 120 to finish securing the web application.

Security constraints in web applications

Security constraints determine how web content is to be protected.

These properties associate security constraints with one or more web resource collections. A constraint consists of a web resource collection, an authorization constraint and a user data constraint.

- A web resource collection is a set of resources (URL patterns) and HTTP methods on those resources. All requests that contain a request path that matches the URL pattern described in the web resource collection are subject to the constraint. If no HTTP methods are specified, then the security constraint applies to all HTTP methods.
- An authorization constraint is a set of roles that users must be granted in order to access the resources described by the web resource collection. If a user who requests access to a specified Uniform Resource Identifier (URI) is not granted at least one of the roles specified in the authorization constraint, the user is denied access to that resource.
- A user data constraint indicates that the transport layer of the client or server communications process must satisfy the requirement of either guaranteeing content integrity (preventing tampering in transit) or guaranteeing confidentiality (preventing reading while in transit).

Note: This release of WebSphere Application Server supports security constraints that are defined in the Java Servlet 3.0 specification (JSR-315).

However, if you use the HTTP custom method, see the information in the Security custom properties topic regarding the security.allowCustomHTTPMethods custom property, which differs slightly from its usage in the Java Servlet 3.0 specification.

Security settings

Use the administrative console to modify the security settings for all applications.

You can enable security for applications by selecting the **Enable application security** option on the Global security panel.

The default settings are used as a template or starting point for configuring individual applications. The administrator should still explicitly configure security settings for each application.

The following security settings are specified during application assembly:

Security role settings

When using an assembly tool at an application level (Enterprise Archive (EAR) file), security roles are synchronized with the security roles defined for the embedded modules of the application.

If a security role is manually added to the EAR file, it can be automatically removed when the file is saved if an embedded module does not reference the role, or the role is in conflict with an existing role. In this case, remove the manually added role, but then all roles with the same name are removed.

The role is automatically added again when the file is saved if it is still referenced in an embedded module file. If a duplicate role is added in an embedded module file, delete all roles with the same name and manually read the correct role.

Security constraints

Security constraints declare how to protect web content. These properties associate security constraints with one or more web resource collections. A *constraint* consists of a web resource collection, an authorization constraint, and a user data constraint.

Security constraints are set when configuring a web application in an assembly tool.

Security role references in web applications

Web application developers or Enterprise JavaBeans (EJB) providers must use a role-name in the code when using the available programmatic security Java Platform, Enterprise Edition (Java EE) application programming interfaces (APIs) isUserInRole(String roleName) and isCallerInRole(String roleName).

The roles used in the deployed run-time environment might not be known until the web application and EJB components (for example, Web archive (WAR) files and `ejb-jar.xml` files) are assembled into an enterprise archive (EAR) file. Therefore, the role names used in the web application or EJB component code are logical role names which the application assembler maps to the actual run-time environment roles during application assembly. The security role references provide a level of indirection that insulate web application component and EJB developers from having to know the actual roles in the run-time environment.

The definition of the logical roles and the mapping to the actual run-time environment roles are specified in the security-role-ref element of both the web application and the EJB JAR file deployment descriptors, `web.xml` and `ejb-jar.xml` respectively. Use the assembly tools to define the role names and map them to the actual run-time roles in the environment with the role-link element.

The following code sample is an example of a security-role-ref from an EJB `ejb-jar.xml` deployment descriptor.

```
... <enterprise-beans>
... <entity>
<ejb-name>AardvarkPayroll</ejb-name>
<ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
...
<security-role-ref>
<description>
```

This role should be assigned to the employees of the payroll department. Members of this role have access to the payroll record of everyone. The role has been linked to the payroll-department role. This role

should be assigned to the employees of the payroll department. Members of this role have access to all payroll records. The role has been linked to the payroll-department role.

```
</description> <role-name>payroll</role-name>
<role-link>payroll-department</role-link>
</security-role-ref>
 ...
</entity>
 ...
</enterprise-beans>
```

In the previous example, the string `payroll`, which appears in the <role-name> element, is what the EJB provider uses as the argument to the isCallerInRole() API. The <role-link> element is what ties the logical role to the actual role used in the run-time environment.

Note that for enterprise beans, the security-role-ref element must appear in the deployment descriptor even if the logical role name is the same as the actual role name in the environment.

The rules web application components are slightly different. If no security-role-ref element matching a security-role element is declared, the container must default to checking the role-name element argument against the list of security-role elements for the web application. The isUserInRole method references the list to determine whether the caller is mapped to a security role. The developer must be aware that the use of this default mechanism can limit the flexibility in changing role names in the application without having to recompile the servlet making the call.

See the EJB Version 2.0 and Servlet Version 2.3 specification in the Security: Resources for Learning article for complete details on this specification.

Assigning users and groups to roles

This topic describes how to assign users and groups to roles if you are using WebSphere Application Server authorization for Java Platform, Enterprise Edition (Java EE) roles.

Before you begin

gotcha: If you are using System Authorization Facility (SAF) authorization for Java2 EE (J2EE) roles, refer to System Authorization Facility for role-based authorization for more information.

Before you perform this task:

- Secure the web applications and Enterprise JavaBeans (EJB) applications where new roles are created and assigned to web and enterprise bean resources.
- Create all the roles in your application.
- Verify that you have properly configured the user registry that contains the users that you want to assign. It is preferable to have security turned on with the user registry of your choice before beginning this process.
- Make sure that if you change anything in the security configuration you save the configuration and restart the server before the changes become effective. For example, enable security or change the user registry.

About this task

These steps are common for both installing an application and modifying an existing application. If the application contains roles, you see the Security role to user/group mapping link during application installation and also during application management, as a link in the Additional properties section.

Procedure

1. Access the administrative console.

   Type `http://localhost:`*`port_number`*`/ibm/console` in a web browser.

2. Click **Applications > Application Types > WebSphere enterprise applications > *application_name*** .
3. Under Detail properties, click **Security role to user/group mapping**. A list of all the roles that belong to this application is displayed. If the roles already have users, or if one of the special subjects, AllAuthenticatedUsers, AllAuthenticatedInTrustedRealms, or Everyone is assigned, they display here.
4. To assign the special subjects, select either the **Everyone** or the **All Authenticated in Application's Realm** option for the appropriate roles.
5. To assign users or groups, select the role. You can select multiple roles at the same time, if the same users or groups are assigned to all the roles.
6. Click **Look up users** or **Look up groups**.
7. Get the appropriate users and groups from the user registry by completing the Limit and the Search string fields and by clicking **Search**. The Limit field limits the number of users that are obtained and displayed from the user registry. The pattern is a searchable pattern matching one or more users and groups. For example, `user*` lists users like user1, user2. A pattern of asterisk (*) indicates all users or groups.

   Use the limit and the search strings cautiously so as not to overwhelm the user registry. When you use large user registries such as Lightweight Directory Access Protocol (LDAP) where information on thousands of users and groups resides, a search for a large number of users or groups can make the system slow and can make it fail. When more entries exist than requests for entries, a message displays on top of the panel. You can refine your search until you have the required list.

   If the search string you are using has no matches, a NULL error message is displayed. This message is informational and does not necessarily indicate an error, as it is valid to have no entries matching your selected criteria.
8. Select the users and groups to include as members of these roles from the **Available** field and click **>>** to add them to the roles.
9. To remove existing users and groups, select them from the **Selected** field and click **<<**. When removing existing users and groups from roles, use caution if those same roles are used as RunAs roles.

   For example, if the user1 user is assigned to the role1 RunAs role and you try to remove the user1 user from the role1 role, the administrative console validation does not delete the user. A user can only be part of a RunAs role if the user is already in a role either directly or indirectly through a group. In this case, the user1 user is in the role1 role. For more information on the validation checks that are performed between RunAs role mapping and user and group mapping to roles, see “Assigning users to RunAs roles” on page 112.
10. Click **OK**. If any validation problems exist between the role assignments and the RunAs role assignments, the changes are not committed and an error message that indicates the problem displays at the top of the panel. If a problem exists, make sure that the user in the RunAs role is also a member of the regular role. If the regular role contains a group that contains the user in the RunAs role, make sure that the group is assigned to the role using the administrative console. Follow steps 4 and 5. Avoid using any process where the complete name of the group, host name, group name, or distinguished name (DN) is not used.

Results

The user and group information is added to the binding file in the application. This information is used later for authorization purposes.

Note: If you change your realm you must repeat this process with the new realm name.

What to do next

This task is required to assign users and groups to roles, which enables the correct users and groups to access a secured application. If you are installing an application, complete your installation. After the

application is installed and running you can access your resources according to the user and group mapping that you did in this task. If you manage applications and modify the users and groups to role mapping, make sure you save, stop, and restart the application so that the changes become effective. Try accessing the Java EE resources in the application to verify that the changes are effective.

Note: Depending upon how your active user registry is configured, the search results of security user or group role mappings are displayed in different formats. With federated repository, LDAP, file-based and custom registries can be used. WebSphere Application Server can uniquely identify users from various registries by the user names listed in the table.

Adding users and groups to roles using an assembly tool

After creating new roles and assigning them to enterprise bean and web resources, use this task to add users and groups to roles with an assembly tool.

Before you begin

Before you perform this task, you already completed the steps in "Securing web applications using an assembly tool" on page 101 and "Securing enterprise bean applications" on page 27 where you created new roles and assigned those roles to enterprise bean and web resources. Complete these steps during application installation. The environment user registry under which the application is running is not known until deployment.

About this task

If you already know the environment in which the application is running and the user registry that is used, you can use an assembly tool to assign users and groups to roles. Using the administrative console to assign users and groups to roles is recommended.

The following information applies to authorization using WebSphere Application Server bindings. If you create WebSphere Application Server bindings, but specify System Authorization Facility (SAF) authorization, the WebSphere Application Server bindings are ignored. If SAF authorization is to be used, you must create a SAF EJBROLE profile for each Java Platform, Enterprise Edition (Java EE) role in your application, and permit users and groups to that role. Refer to System Authorization Facility for role-based authorization for reference.

Note: This procedure might not match the steps that are required when using your assembly tool, or match the version of the assembly tool that you are using. You should follow the instructions for the tool and version that you are using.

To add users and groups to roles using an assembly tool, follow these steps:

Procedure

1. In the Project Explorer view of an assembly tool, right-click an enterprise application project, or Enterprise Archive (EAR) file, and click **Open With** > **Deployment Descriptor Editor**. An application deployment descriptor editor opens on the EAR file. To access information about the editor, press F1 and click **Application deployment descriptor editor**.
2. Click the **Security** tab and, under the main panel, click **Add**.
3. In the Add Security Role wizard, name and describe the security role. Click **Finish**.
4. Under WebSphere Bindings, select the user or group extension properties for the security role. Available values include: Everyone, All authenticated users, and Users/Groups.
5. If you selected Users/Groups, click **Add** beside the Users or Groups panes. In the wizard that opens, specify a user or group name and click **Finish**. Repeat this step until you added all the users and groups to which the security role applies.
6. Close the application deployment descriptor editor and, when prompted, click **Yes** to save the changes.

Results

The `ibm-application-bnd.xmi` or `ibm-application-bnd.xml` file in the application contains the users and groups-to-roles mapping table, which is the *authorization table*. For Java EE Version 5 applications, the `ibm-application-bnd.xml` file contains the authorization table.

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

- For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.
- For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

What to do next

After securing an application, install the application using the administrative console.

Security role to user or group mapping

Use this page to specify the users and groups that are mapped to the security roles that are used with the enterprise application.

gotcha: If you are using System Authorization Facility (SAF) authorization for Java2 EE (J2EE) roles, refer to System Authorization Facility for role-based authorization for more information.

To view this administrative console page, click **Applications > Application types > WebSphere enterprise applications >***application_name*. Under Detail Properties, click **Security role to user/group mapping**.

Table 14. User and group mapping. User and group mapping.

Button	Resulting action
Map Users	Lists the users that are mapped to the specified role within this application. If trusted realms are configured, a drop-down list of realms to search is displayed. Users from the non-default realm are displayed as user@realm
Map Groups	Lists the groups that are mapped to this specified role within this application. If trusted realms are configured, a drop-down list of realms to search is displayed. Users from the non-default realm are displayed as user@realm

Table 14. User and group mapping (continued). User and group mapping.

Button	Resulting action
Map Special Subjects	This choice appears if multiple realms are being used. It enables you to map any of the following Special Subjects to a selected role: • **All authenticated in application realm**: All authenticated users that are in the applications realm, which specifies whether to map all of the authenticated users to a specified role. When you map all authenticated users to a specified role, all of the valid users in the current registry who have been authenticated can access resources that are protected by this role. This selection also applies to all authenticated users regardless of the realm. • **Everyone**: map everyone to the selected role. When you map everyone to a role, anyone can access the resources that are protected by this role and, essentially, there is no security. • **None**: Do not map anyone to the selected role **Attention:** • If the secured realm cannot be reached, the left list is replaced with 3 text fields (that is, `name, realm, and uid`). You can add the user when the secured realm is not available. It is not possible to map two subjects to the same role in this release of WebSphere Application Server.

Role:

Lists the specific capabilities to a user. Role privileges give users and groups permission to run as specified.

For example, you might map the user Joe to the administrator role, which enables user Joe to perform all of the tasks associated with the administrator role.

The authorization policy is only enforced when global security is enabled.

Mapped users:

Lists the users that are mapped to the specified role within this application.

Special subjects:

Lists which special subjects are mapped to the security role when an application uses multiple realms.

Mapped groups:

Lists the groups that are mapped to this specified role within this application.

Look up users

Use this page to select and to map users, groups and special subjects for security roles.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application types** > **WebSphere enterprise applications** > *application_name*.
2. Under Detail Properties, click **Security role to user/group mapping**.
3. Select the role and click either **Map users...**, **Map groups...** or **Map Special Subjects**.

Note: Once you click **OK** after making any changes, you must also click **OK** on the previous panel for the changes to be accepted.

Different roles can have different security authorizations. Mapping users or groups to a role authorizes those users or groups to access applications defined by the role. Users and groups are associated with roles defined in an application when the application is installed or configured. Use the Search pattern field to display users in the Available list. Click **>>** to add users from the Available list to the Selected list.

Map users...:

Lists the users that are mapped to the specified role within this application.

Map groups...:

Lists the groups that are mapped to this specified role within this application.

Map Special Subjects:

This choice appears if multiple realms are being used. It enables you to map any of the following to selected roles:

- All authenticated users that are in the applications's realm, which specifies whether to map all of the authenticated users to a specified role. When you map all authenticated users to a specified role, all of the valid users in the current registry who have been authenticated can access resources that are protected by this role.
- All authenticated users regardless of the realm.
- Everyone, which specifies whether to map everyone to a specified role. When you map everyone to a role, anyone can access the resources that are protected by this role and, essentially, there is no security.
- All users in the trusted realms.

  If trusted realms are configured, a drop-down list of realms to search is displayed. Users from the non-default realm are displayed as user@realm.

Note: If the secured realm cannot be reached, the left list is replaced with 3 text fields (that is, `name, realm, and uid`). You can add the user when the secured realm is not available.

It is not possible to map two subjects to the same role in this release of WebSphere Application Server.

Limit:

Specifies the maximum number of users or groups that can be returned when assigning users/groups to roles.

A value of 0 implies a return of all users or groups that match the pattern. You can either increase the limit or refine the search pattern to get all the entries.

Information	Value
Data type	Integer
Units	User name
Default	20
Range	0 or more

Search string:

Indicates the search pattern used to search for the entries in a user registry.

The Search string field contains the search pattern that is used to search for the user or group entries. For example, bob* will search all users or groups starting with bob. A limit of zero (0) retrieves all of the entries that match the pattern. Use a limit of zero (0) only when a small number users or groups match that pattern in the user registry. If the user registry contains more entries that match the pattern than requested for, a message shows in the administrative console to indicate that there are more entries in the user registry.

Information	Value
Data type	String
Units	Number of users
Default	20
Range	A-Z with *

Assigning users to RunAs roles

This article explains how to assign users to the RunAs roles for your application.

Before you begin

Complete the following tasks:
- Secure the web applications and the EJB applications where new RunAs roles are created and assigned to web and EJB resources.
- Create all the RunAs roles in your application. The user in the RunAs role can only be entered if that user or a group to which that user belongs is already part of the regular role.
- Assign users and groups to security roles. Refer to “Assigning users and groups to roles” on page 106 for more information.
- Verify that the user registry requirements are met. These requirements are the same as those discussed in “Assigning users and groups to roles” on page 106. For example, if the role1 role is a role that is also used as a RunAs role, then the user1 user can be added to the RunAs role. The administrative console checks this logic when **Apply** or **OK** is clicked. If the check fails, the change is not made and an error message is displayed at the top of the panel.

When a user ID and password is assigned to a RunAs role, validation occurs using the current active user registry that is configured. By default, the local operating system registry is set as the active user registry. Therefore, when an application is installed and security is disabled on the server, the local operating system registry is used to validate the user ID and password that is assigned to the RunAs Role. If the intended registry for the application is not local operative system, the validation fails. Therefore, map RunAs roles to users when the security is enabled on the server. However, if the active user registry and the intended registry after enabling security are the same, you can assign the user to a RunAs role when security is disabled.

If the Everyone or All Authenticated special subjects are assigned to a role, validation does not occur for that role.

Validation is done every time you click **Apply** in this panel or when you click **OK** in the Security role to user/group mapping panel. The check verifies that all the users in all the RunAs roles do exist directly or indirectly through a group in those roles in the Security role to user/group mappings panel. If a role is assigned both a user and a group to which that user belongs, you can delete either the user or the group from the Security role to user/group mapping panel.

If the RunAs role user belongs to a group and if that group is assigned to that role, make sure that the assignment of this group to the role is done through the administrative console and not through an assembly tool or other method. When using the administrative console, the full name of the group is used (for example, `hostname\groupName` in Windows systems and distinguished names (DN) in Lightweight Directory Access Protocol (LDAP)). During the check, all the groups to which the RunAs role user belongs are obtained from the user registry. Because the list of groups that are obtained from the user registry are

the full names of the groups, the check works correctly. If the short name of a group is entered using an assembly tool, for example `group1` instead of `CN=group1, o=myCompany.com`, this check fails.

About this task

These steps are common to both installing an application and modifying an existing application. If the application contains RunAs roles, you see the User RunAs roles link during application installation and also during managing applications as a link in the Additional properties section.

Procedure

1. Click **Applications > Enterprise Applications >** ***application_name***.
2. Under Detail Properties, click **Security role to user/group mapping**. A list of all the RunAs roles that belong to this application display. If the roles already have users assigned, they display here.
3. To assign a user, select the role. You can select multiple roles at the same time if the same user is assigned to all the roles.
4. Enter the user's name and password in the designated fields. The user name entered can be either the short name, which is preferred, or the full name, as seen when getting users and groups from the user registry.
5. Click **Apply**. The user is authenticated using the active user registry. If authentication is successful, a check is made to verify that this user or group is mapped to the role in the Map security roles to users and groups panel. If authentication fails, verify that the user and password are correct and that the active registry configuration is correct.
6. To remove a user from a RunAs role, select the roles and click **Remove**.

Results

The RunAs role user is added to the binding file in the application. This file is used for delegation purposes when accessing Java EE resources. This step is required to assign users to RunAs roles so that during delegation the appropriate user is used to invoke the EJB methods.

What to do next

If you are installing the application, complete installation. After the application is installed and running, you can access your resources according to the RunAS role mapping. Save the configuration.

If you manage applications and modify User RunAs roles, make sure you save, stop, and restart the application so that the changes become effective. Try accessing your Java Platform, Enterprise Edition (Java EE) resources to verify that the new changes are in effect.

Mapping users to RunAs roles using an assembly tool:

RunAs roles are used for delegation. A servlet or enterprise bean component uses the RunAs role to invoke another enterprise bean by impersonating that role.

Before you begin

Before you perform this task:
- Secure the web application and enterprise bean applications, including creating and assigning new roles to enterprise bean and web resources. For more information, see "Securing web applications using an assembly tool" on page 101 and "Securing enterprise bean applications" on page 27.
- Assign users and groups to roles. For more information, see "Adding users and groups to roles using an assembly tool" on page 108. Complete this step during the installation of the application. The environment or user registry under which the application is going to run is not known until deployment. If you already know the environment in which the application is going to run and you know the user registry, then you can use an assembly tool to assign users to RunAs roles.

About this task

Note: This procedure might not match the steps that are required when using your assembly tool, or match the version of the assembly tool that you are using. You should follow the instructions for the tool and version that you are using.

To define RunAs roles when a servlet or an enterprise bean in an application is configured with RunAs settings, perform these steps:

Procedure

1. In the Project Explorer view of an assembly tool, right-click an enterprise application project or Enterprise Archive (EAR) file and click **Open With** > **Deployment Descriptor Editor**. An application deployment descriptor editor opens on the EAR file. To access information about the editor, press F1 and click **Application deployment descriptor editor**.
2. On the Security tab, under Security Role Run As Bindings, click **Add**.
3. Click **Add** under RunAs Bindings.
4. In the Security Role wizard, select one or more roles and click **Finish**.
5. Repeat steps 3 through 5 for all the RunAs roles in the application.
6. Close the application deployment descriptor editor and, when prompted, click **Yes** to save the changes.

Results

The `ibm-application-bnd.xmi` file in the application contains the user to RunAs role mapping table.

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

- For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.
- For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

What to do next

After securing an application, you can install the application using the administrative console. You can change the RunAs role mappings of an installed application. For more information, see “User RunAs collection” on page 117.

Ensure all unprotected 1.x methods have the correct level of protection:

Use this page to verify that the unprotected Enterprise JavaBeans (EJB) Version 1.*x* methods have the correct level of protection before you map users to roles.

This administrative console page is displayed during the application deployment process. To access the administrative console page, click **Application** > **New application** > **New Enterprise Application**. The

page is displayed as **Ensure all unprotected 1.x methods have the correct level of protection** in the application deployment steps. On this administrative console page, you can specify whether users can access specific EJB modules.

EJB module:

Specifies the EJB module name.

URI:

Specifies the Uniform Resource Identifier (URI) that is used to locate the Java archive (JAR) file for the EJB module.

Deny all access:

Select this option to protect this EJB module by making it inaccessible to users regardless of their access permissions.

Information	Value
Default:	Cleared

Ensure all unprotected 2.x methods have the correct level of protection:

Use this page to verify that the unprotected Enterprise JavaBeans (EJB) Version 2.*x* methods have the correct level of protection before you map users to roles.

This administrative console page is displayed during the application deployment process. To access the administrative console page, click **Applications** > **New application** > ***application_name***. The page is displayed as **Ensure all unprotected 2.x methods have the correct level of protection** in the application deployment steps. On this administrative console page, you can specify whether users can access specific EJB modules.

To use this administrative console page, select the **Uncheck**, **Exclude**, or **Role** option, the check box next to the EJB module, and click **Apply**. If you select **Role** option, select the appropriate role for the EJB module before you click **Apply**.

Uncheck:

Select this option if you do not want the application server to verify the access permissions for the EJB module. Everyone can access the EJB module.

Information	Value
Default:	Selected

Exclude:

Select this option to protect this EJB module by making it inaccessible to users regardless of their access permissions.

Information	Value
Default:	Deselected

Role:

Specifies the EJB level of protection based on the security role.

The roles listed in this menu are obtained from the application scope. If the selected role is not in the module, then it is added to the modules or Java archive (JAR) files.

Information	Value
Default:	Deselected

EJB module:

Specifies the name of the module.

If a module name appears in this list, then the module contains unprotected EJB methods.

URI:

Specifies the Uniform Resource Identifier (URI) that is used to locate the Java archive (JAR) file for the EJB module.

Protection type:

Specifies the level of protection that is assigned to a particular module name.

After you select the **Uncheck**, **Exclude**, or **Role** option and click **Apply**, the selected protection option is displayed in this column.

Correct use of the system identity:

Use this page to manage the system identity properties for the Enterprise JavaBeans (EJB) method in your application.

This administrative console page is displayed during the application deployment process. To access the administrative console , click **Application** > **New application** > **New Enterprise Application**. The is displayed as *Correct use of System Identity* in the application deployment steps.

To use this page, complete the following steps:
1. Select an application that supports security and click **Next**.
2. Select **Detailed - Show all installation options and parameters** and click **Next**.
3. Select the **Correct use of system identity** step.

Bean:

A component that implements a business task or business entity and resides in an EJB container. Entity beans, session beans, and message-driven beans are all enterprise beans.

Module:

In Java EE programming, a software unit that consists of one or more components of the same container type and one deployment descriptor of that type. Examples include EJB, Web, and application client modules.

URI:

A Uniform Resource Identifier (URI) is a unique address that is used to identify content on the Web, such as a page of text, a video or sound clip, a still or animated image, or a program.

Method signature:

The combination of a name of a method along with the number and types of the parameters and their order.

Role:

Specifies the RunAs role that is used for this EJB method.

Username:

Specifies the user name that is assigned to the RunAs role for this EJB method.

The user name is used in conjunction with the RunAs role that you select for the Role.

User RunAs collection:

Use this page to map a specified user identity and password to a RunAs role. This panel enables you to specify application-specific privileges for individual users to run specific tasks using another user identity.

To view this administrative console page, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Detail properties, click **User runAs roles**.

The enterprise beans that you install contain predefined RunAs roles. RunAs roles are used by enterprise beans that need to run as a particular role for recognition while interacting with another enterprise bean.

Username:

Specifies a user name for the RunAs role user.

This user already maps to the role specified in the Mapping users and groups to roles panel. You can map the user to its appropriate role by either mapping the user to that role directly or mapping a group that contains the user to that role. After you specify the user name and password for the user and select a RunAs role, click **Apply**.

Note:

The use of the username field is dependent on whether system authorization facility (SAF) delegation is enabled or disabled.
- SAF delegation is enabled. The username field is NOT used.
- SAF delegation is disabled. The username field is used.

Information	Value
Data type:	String

Password:

Specifies the password for the RunAs user.

Note:

The use of the password field is dependent on whether system authorization facility (SAF) delegation is enabled or disabled.
- SAF delegation is enabled. The password field is NOT used.

- SAF delegation is disabled. The password field is used.

Information	Value
Data type:	String

Role:

Maps specific capabilities to a user.

The authorization policy is only enforced when global security is enabled.

Securing applications during assembly and deployment

Several assembly tools exist that are graphical user interfaces for assembling enterprise or Java Platform, Enterprise Edition (Java EE) applications. You can use these tools to assemble an application and secure Enterprise JavaBeans (EJB) and web modules in that application.

About this task

An EJB module consists of one or more beans. You can enforce security at the EJB method level. A web module consists of one or more web resources: an HTML page, a JavaServer Pages (JSP) file, or a servlet. You can also enforce security for each web resource.

Note: For information about the tools that WebSphere Application Server supports, see Assembly tools.

To secure an EJB module, a Java archive (JAR) file, a web module, a web application archive (WAR) file, or an application enterprise archive (EAR) file, you can use an assembly tool. You can create an application, an EJB module, or a web module and secure them using an assembly tool or development tools such as the IBM Rational Application Developer.

Procedure

1. Secure EJB applications using an assembly tool. For more information, see “Securing enterprise bean applications” on page 27.
2. Secure web applications using an assembly tool. For more information, see “Securing web applications using an assembly tool” on page 101.
3. Add users and groups-to-roles while assembling a secured application using an assembly tool. For more information, see “Adding users and groups to roles using an assembly tool” on page 108.
4. Map users to RunAs roles using an assembly tool. For more information, see “Mapping users to RunAs roles using an assembly tool” on page 113.
5. Adding the was.policy file to applications for Java 2 security.
6. Assemble the application components that you secured using an assembly tool. For more information, see Assembling applications.

Results

After securing an application, the resulting `.ear` file contains security information in its deployment descriptor. The EJB module security information is stored in the `ejb-jar.xml` file and the web module security information is stored in the `web.xml` file. The `application.xml` file of the application EAR file contains all the roles that are used in the application. The user and group-to-roles mapping is stored in the `ibm-application-bnd.xmi` file of the application EAR file.

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or

module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

- For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.
- For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

This task is required to secure EJB modules and web modules in an application. This task is also required for applications to run properly when Java 2 security is enabled. If the `was.policy` file is not created and it does not contain required permissions, the application might not be able to access system resources.

What to do next

After securing an application, you can install an application using the administrative console. When you install a secured application, refer to “Deploying secured applications” on page 120 to complete this task.

Updating and redeploying secured applications

This section addresses the way to update existing applications.

Before you begin

Before you perform this task, secure web applications, secure Enterprise JavaBeans (EJB) applications, and deploy them in WebSphere Application Server.

Procedure

1. Use the administrative console to modify the existing users and groups mapping to roles. For information on the required steps, see “Assigning users and groups to roles” on page 106.
2. Use the administrative console to modify the users for the RunAs roles. For information on the required steps, see “Assigning users to RunAs roles” on page 112.
3. Complete and save the changes.
4. Stop and restart the application for the changes to become effective.
5. Use an assembly tool. For more information, see Assembling applications.
6. Use an assembly tool to modify roles, method permissions, auth-constraints, data-constraints and so on. For more information, see Assembling applications.
7. Save the enterprise archive (EAR) file, uninstall the old application, deploy the modified application and start the application to make the changes effective.

Results

The applications are modified and redeployed. This step is required to modify existing secured applications.

What to do next

If information about roles is modified, make sure that you update the user and group information using the administrative console. After the secured applications are modified and either restarted or redeployed,

verify that the changes are effective by accessing the resources in the application.

Deploying secured applications

Deploying applications that have security constraints (secured applications) is not much different than deploying applications that do not contain any security constraints. The only difference is that you might need to assign users and groups to roles for a secured application. The secured application requires that you have the correct active user registry.

Before you begin

Before you perform this task, verify that you already designed, developed, and assembled an application with all the relevant security configurations. For more information on these tasks refer to Developing applications that use programmatic security and "Securing applications during assembly and deployment" on page 118. In this context, deploying and installing an application are considered the same task.

To deploy a newly secured application click **Applications > Install New Application** and follow the prompts to complete the installation steps. One of the required steps to deploy secured applications is to assign users and groups to roles that are defined in the application.

- If you are installing a secured application, roles will be defined in the application.
- If delegation is required in the application, you will be defining RunAs roles also.

During the installation of a new application, the role definition is completed as part of the step that maps security roles to users and groups. If this assignment has already been completed by using an assembly tool, you can still confirm the mapping by following this installation step. You can add new users and groups and modify existing information during this step.

If the application supports delegation, a RunAs role will already be defined in the application. If the delegation policy is set to `Specified Identity` during assembly, the intermediary invokes a method by using an identity setup during deployment. Use the RunAs role to specify the identity under which the downstream invocations are made. For example, if the RunAs role is assigned user `bob` and the client `alice` is invoking a servlet, with delegation set that calls the enterprise beans, the method on the enterprise beans is invoked with `bob` as the identity.

As part of the new application installation and deployment process, one of the steps is to map or modify users to the RunAs roles. Use this step to assign new users or modify existing users to RunAs roles when the delegation policy is set to `Specified Identity`.

Important: When Tivoli Access Manager (TAM) is enabled the deployment and undeployment of applications might take a long time or even time out. Disabling the ATCCache might resolve the issue. The ATCCache exists to help with performance during application deployment and undeployment. With some applications, especially those with many modules, the cache can actually have a negative impact on performance in these areas. To disable the ATCCache, navigate to the `config/cells/cell_name` directory and modify the `amwas.amjacc.template.properties` file to set com.tivoli.pd.as.atcc.ATCCache.enabled=false. Because embedded TAM is already configured, update the configuration files with that property. For each instance in the cell, go to the `profiles/<profile_name>/etc/tam` directory and modify any file ends as `amjacc.properties` to set com.tivoli.pd.as.atcc.ATCCache.enabled=false. The cell must be restarted before these changes take effect.

About this task

Note that the steps are common whether you are installing an application or modifying an existing application.

To install and deploy the application, complete the following steps.

Procedure

1. Click **Applications > Install New Application**. Complete the required steps until you see the step for mapping security roles to users and groups.
2. If RunAs roles exist in the application, assign users to RunAs roles. At this step during the installation, under Additional Properties, click **Map RunAs roles to users**. For more information, see "Assigning users to RunAs roles" on page 112.
3. Optional: Click **Correct use of System Identity** to specify RunAs roles, if needed. Complete this action if the application has delegation set to use system identity, which is applicable to enterprise beans only. System identity uses the WebSphere Application Server security server ID to invoke downstream methods. Using system identity is not recommended as this ID has more privileges than other identities in accessing WebSphere Application Server internal methods. This task is provided to make sure that the deployer is aware that the methods listed in the panel have system identity set up for delegation and to correct them if necessary. When the internalServerId feature is used, runAs with system identity is not supported; you must specify RunAs roles here.
4. Complete the remaining non-security related steps to finish installing and deploying the application.

What to do next

After a secured application is deployed, verify that you can access the resources in the application with the correct credentials. For example, if your application has a protected web module, make sure only the users that you assigned to the roles can use the application.

Session security support

You can integrate HTTP sessions and security in WebSphere Application Server. When security integration is enabled in the session management facility and a session is accessed in a protected resource, you can access that session only in protected resources from then on. Session security (security integration) is enabled by default.

You cannot mix secured and unsecured resources accessing sessions when security integration is turned on. Security integration in the session management facility is not supported in form-based login with SWAM.

Note: SWAM is deprecated in WebSphere Application Server Version 8.5 and will be removed in a future release.

Security integration rules for HTTP sessions

Only authenticated users can access sessions created in secured pages and are created under the identity of the authenticated user. Only this authenticated user can access these sessions in other secured pages. To protect these sessions from unauthorized users, you cannot access them from an unsecured page.

Programmatic details and scenarios

WebSphere Application Server maintains the security of individual sessions.

An identity or user name, readable by the com.ibm.websphere.servlet.session.IBMSession interface, is associated with a session. An unauthenticated identity is denoted by the user name `anonymous`. WebSphere Application Server includes the com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException class, which is used when a session is requested without the necessary credentials.

The session management facility uses the WebSphere Application Server security infrastructure to determine the authenticated identity associated with a client HTTP request that either retrieves or creates a session. WebSphere Application Server security determines identity using certificates, LPTA, and other methods.

After obtaining the identity of the current request, the session management facility determines whether to return the session by comparing the identity of the request with the identity of the session.

Table 15. Security integration scenarios. The following table lists possible scenarios in which security integration is enabled with outcomes dependent on whether the HTTP request is authenticated and whether a valid session ID and user name was passed to the session management facility.

Type of session ID	Unauthenticated HTTP request is used to retrieve a session	HTTP request is authenticated, with an identity of "FRED"used to retrieve a session
No session ID was passed in for this request, or the ID is for a session that is no longer valid	A new session is created. The user name is `anonymous`	A new session is created. The user name is `FRED`
A session ID for a valid session is passed in. The current session user name is "anonymous"	The session is returned.	The session is returned. session management changes the user name to `FRED`
A session ID for a valid session is passed in. The current session user name is `FRED`	The session is not returned. An UnauthorizedSessionRequestException error is created*	The session is returned.
A session ID for a valid session is passed in. The current session user name is `BOB`	The session is not returned. An UnauthorizedSessionRequestException error is created*	The session is not returned. An UnauthorizedSessionRequestException error is created*

Note: *A com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException error is created to the servlet.

Chapter 11. Securing web services

This page provides a starting point for finding information about web services.

Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. They implement a services oriented architecture (SOA), which supports the connecting or sharing of resources and data in a very flexible and standardized manner. Services are described and organized to support their dynamic, automated discovery and reuse.

Securing JAX-RS web applications

Securing JAX-RS applications within the web container

You can use the security services available from the web container to secure Representational State Transfer (REST) resources. You can configure security mechanisms that define user authentication, transport security, authorization control, and user to role mappings.

Before you begin

To appropriately define security constraints, it is important that you are familiar with your application and the RESTful resources that it exposes. This knowledge helps you to determine appropriate security roles required by your application as well as the individual resources it exposes.

To illustrate how to secure a REST application, this topic uses a sample REST application called AddressBookApp.

You must complete the installation of your application on the application server. For example, after you install the AddressBookApp application, the AddressBookApp deployment descriptor found in the *profile_root*/config/cells/cellName/applications/applicationName.ear/deployments/applicationName_war/applicationName.war/WEB-INF directory looks like the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_1255468655347">
 <display-name>Sample REST Web Application</display-name>
  <servlet>
    <servlet-name>AddressBookApp</servlet-name>
    <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
    <init-param>
      <param-name>javax.ws.rs.Application</param-name>
      <param-value>com.test.AddressBookApplication</param-value>
</init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>AddressBookApp</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

In this example, the servlet mapping indicates the REST resources are served under the /*app_root_context*/rest directory where *app_root_context* is what you configured during the installation of the application. The default root context is /.

You must enable security for WebSphere Application Server.

About this task

You can use the web container to apply authentication as well as authorization constraints to a REST application running in the application server environment. Authentication is a basic security requirement for business REST resources that require a minimum level of security and might need to further protect resources based on the identity of the caller.

© Copyright IBM Corp. 2012

You can configure the following security mechanisms for REST resources:

- Require that users authenticate to your application using either HTTP basic authentication or form login.
- Configure your application to use an SSL channel for transport when invoking REST resources.
- Define role-based authorization constraints on your REST resource patterns.
- Implement the programmatic use of the annotated SecurityContext object to determine user identity and roles.

Procedure

1. Ensure that security is enabled for the application server.
   a. Start the WebSphere Application Server administrative console.

      Start the deployment manager, and in your browser, type the address of your WebSphere Application Server, Network Deployment server. By default, the console is located at `http://your_host.your_domain:9060/ibm/console`.

      If security is currently disabled, you are prompted for a user ID. Log in with any user ID. However, if security is currently enabled, you are prompted for both a user ID and a password. Log in with a predefined administrative user ID and password.

   b. Click **Security > Global security**.

      Select **Enable application security**.

      Note: You must enable administrative security. You can only have application security enabled when administrative security is enabled.

2. Add security constraints.

   Edit the web.xml file for the application, or use an assembly tool to add security constraints to your application. The following code snippet is a security constraint applied to the AddressBookApp Sample application:

```
<!-- Security constraint for the sample application -->
  <security-constraint id="SecurityConstraint_1">
    <!-- This defines the REST resource associated with the constraint. -->
    <web-resource-collection id="WebResourceCollection_1">
      <web-resource-name>AddressBookApp</web-resource-name>
      <description>Protection area for Rest resource /addresses </description>
      <url-pattern>/rest/addresses</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>

    <!—This defines an authorization constraint by requiring Role1 for the resource. -->
    <auth-constraint id="AuthConstraint_1">
      <description>Used to guard resources under this url-pattern </description>
      <role-name>Role1</role-name>
    </auth-constraint>
  </security-constraint>
```

   In this example, there is a web resource located at `/root_context/rest/addresses` that can respond to an HTTP GET or POST request. A security constraint, `AuthConstraint_1`, is applied to the web resource. The authorization constraint specifies that role `Role1` is required for users to access the resource.

3. Choose one or more of the following security mechanisms to configure for your REST application.
   - **Enable basic HTTP authentication.**
     a. Add security constraints by editing the web.xml file as previously described.
     b. Configure the web.xml file to enable basic HTTP authentication.

        Edit the web.xml file for the application and add the following element to specify the use of basic HTTP authentication. By default, the application server security runtime environment uses this method of authentication.

```
<!-- This defines a HTTP basic authentication login configuration. -->
 <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>test realm</realm-name>
 </login-config>
```

An HTTP basic authentication method is now defined. Users attempting to access the resource are required to login with credentials.

- **Enable form login.**
  a. Add security constraints by editing the web.xml file as previously described.
  b. Edit the web.xml file for the application and add the following element to specify the use of form login:

```
<login-config>
      <auth-method>FORM</auth-method>
      <form-login-config>
         <form-login-page>/logon.jsp</form-login-page>
         <form-error-page>/logonError.jsp</form-error-page>
      </form-login-config>
</login-config>
```

It is important that you replace the logon.jsp and logonError.jsp web page values with your form login and error processing, respectively. When accessing the application, users are redirected through the logon.jsp web page to authenticate. If there is an authentication failure, users are redirected to the logonError.jsp web page. The following example illustrates the placement of logon.jsp and logonError.jsp pages in the application web application archive (WAR) file:

```
META-INF
      logon.jsp
      logonError.jsp
      WEB-INF/classes/
      WEB-INF/classes/
      WEB-INF/classes/com/
      WEB-INF/classes/com/test/
      WEB- NF/classes/com/test/AddressBookApplication.class
      WEB-INF/classes/com/test/AddressBookResource.class
```

The following code snippet illustrates a sample logon form:

```
<html>
<head>
    <title>Login Page</title>
</head>
<h2>Hello, please log in:</h2>
<br><br>
<form action="j_security_check" method=post>
    <p><strong>Please Enter Your User Name: </strong>
    <input type="text" name="j_username" size="25">
    <p><p><strong>Please Enter Your Password: </strong>
    <input type="password" size="15" name="j_password">
    <p><p>
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
</form>
</html>
```

- **Enable SSL for your application.**
  a. Add security constraints by editing the web.xml file as previously described.
  b. Edit the web.xml file for the application, and add the following element within the security-constraint element:

```
<user-data-constraint id="UserDataConstraint_1">
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
 </user-data-constraint>
```

If you do not want to use SSL, you can either skip this constraint or replace the `CONFIDENTIAL` value with `NONE`.

- **Enable authorization control to protect resources using URL patterns.**
  a. Add security constraints by editing the web.xml file as previously described.
  b. Edit the web.xml file for the application and add the following element within the security-constraint element. In the following example, `Role1` and `Role2` specify to protect the REST resources, `/rest/addresses` and `/rest/resources/{i}`, respectively:

```
<security-constraint id="SecurityConstraint_1">
    <web-resource-collection id="WebResourceCollection_1">
      <web-resource-name>AddressBookApp</web-resource-name>
      <description>Protection area for Rest Servlet</description>
      <url-pattern>/rest/addresses</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
```

```
    <auth-constraint id="AuthConstraint_1">
         <description> Role1 for this rest resource </description>
         <role-name>Role1</role-name>
      </auth-constraint>
</security-constraint>

<security-constraint id="SecurityConstraint_2">
    <web-resource-collection id="WebResourceCollection_2">
      <web-resource-name>AddressBookApp</web-resource-name>
      <description>Protection area for Rest Servlet</description>
      <url-pattern>/rest/addresses/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint id="AuthConstraint_2">
         <description> Role2 for this rest resource </description>
         <role-name>Role2</role-name>
      </auth-constraint>
</security-constraint>
```

In this example, only users that are members of Role1 are able to access `root-context/rest/addresses` and only users that are members of Role2 are able to access the resource, `root-context/rest/addresses/{i}`.

Note: It is important that you prefix the path of the protected resources with your servlet mapping in the security constraints that you define. To prevent bypassing any access checks, you can choose to map the servlet to the `/*` path. This mapping protects all resources under the root context.

Make sure to define your roles by inserting the role definition elements within the `<web-app>` element; for example:

```
<security-role id="SecurityRole_1">
    <description>This is Role1</description>
    <role-name>Role1</role-name>
</security-role>
<security-role id="SecurityRole_2">
    <description>This is Role2</description>
    <role-name>Role2</role-name>
</security-role>
```

The changes you make to the deployment descriptor are automatically picked up by the application server runtime environment, and you do not need to restart the application or the server. Other types of changes, such as the mapping URL, require that you restart the application server. It is recommended that you restart your application to make sure that your changes take effect.

- **Programmatically using the annotated security context.**

  Application developers can use the JAX-RS @SecurityContext annotation to programmatically cascade the security context down to the resource on the server side and enable the definition of security attributes during run time. The following is the functionality provided by the SecurityContext interface:

```
public String getAuthenticationScheme()
public Principal getUserPrincipal()
public boolean isUserInRole(String role)
```

The following example illustrates the SecurityContext interface:

```
package com.test;

import javax.ws.rs.GET;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.ext.*;
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.core.Context;

/**
 * A sample resource that provides access to an address book.
 *
 */
@Path(value="/addresses")
public class AddressBookResource {

 @Context private SecurityContext securityContext;
```

```
    private static String[] list = new String[] {
        "Michael",
        "Ron",
        "Jane",
        "Sam"
    };

    @GET
    @Produces(value="text/plain")
    public String getList() {
       // retrieve the authentication scheme that was used(e.g. BASIC)
       String authnScheme = securityContext.getAuthenticationScheme());
       // retrieve the name of the Principal that invoked the resource
       String username = securityContext.getUserPrincipal().getName());
       // check if the current user is in Role1
        Boolean isUserInRole = securityContext.isUserInRole("Role1");

        StringBuffer buffer = new StringBuffer();
        buffer.append("{");
        for (int i = 0; i < list.length; ++i) {
            if (i != 0)
                buffer.append(", ");
            buffer.append(list[i]);
        }
        buffer.append("}");

        return buffer.toString();
    }
}
```

- **Use the security client handler to perform basic HTTP authentication**

  You can optionally use the security client handler to perform basic HTTP authentication with a secure JAX-RS resource. The following example illustrates the simple programming model to accomplish this task:

```
/**
 * This snippet illustrates the use of the JAX-RS SecurityHandler by a
 * client to perform HTTP basic authentication with a target service.
 */

 import org.apache.wink.client.ClientConfig;
 import org.apache.wink.client.Resource;
 import org.apache.wink.client.RestClient;
 import org.apache.wink.client.handlers.BasicAuthSecurityHandler;

 ClientConfig config = new ClientConfig();
  BasicAuthSecurityHandler secHandler = new
 BasicAuthSecurityHandler();

 // Set the user credential.
 secHandler.setUsername("user1");
 secHandler.setPassword("security");

 // Add this security handler to the handlers chain.
 config.handlers(secHandler);

 // Create the REST client instance.
 RestClient client = new RestClient(config);

 // Create the resource instance to interact with
 // substitute for your resource address
 resource =
  client.resource("http://localhost:8080/path/to/resource");

// Now you are ready to call your resource.
```

When using the `BasicAuthSecurityHandler` class, ensure that you target resources using the https scheme for your URLs, and that the target application is SSL-enabled. It is highly recommended to use SSL connections when sending user credentials. You may explicitly turn off the requirement for SSL in the `BasicAuthSecurityHandler` class by invoking the `setSSLRequired` method on the security handler with the `false` value. By default, this value is `true`.

```
secHandler.setSSLRequired(false);
```

Optionally, you can also provide the user credentials on the Java command-line for your client as follows:

```
java -Duser=test_user -Dpassword=your_password  your_client_program
```

You can optionally retrieve the user credentials from a properties files whose location you specify on the Java command-line as follows:

```
java -Dclientpropsdir=directory_for_your_properties_file  your_client_program
```

where *directory_for_your_properties_file* contains the wink.client.props file where the `user` and `password` properties are set.

Results

After you define security constraints, access to the REST resources that are defined in your application is subject to successful user authentication only. Additionally, you have applied role constraints to various resource URL patterns to enable role-based access to those resources.

Example

The following example illustrates the web.xml deployment descriptor for the AddressBookApp Sample application where security constraints have been defined using the previous procedure steps:

```
<web-app id="WebApp_1255468655347">
    <display-name>Sample REST Web Application</display-name>
    <servlet>
        <servlet-name>AddressBookApp</servlet-name>
        <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.test.AddressBookApplication</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>AddressBookApp</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
    <security-constraint id="SecurityConstraint_1">
      <web-resource-collection id="WebResourceCollection_1">
        <web-resource-name>AddressBookApp</web-resource-name>
        <description>Protection area for Rest Servlet</description>
        <url-pattern>/rest/addresses</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint id="AuthConstraint_1">
         <description>Role1 for this rest servlet</description>
         <role-name>Role1</role-name>
      </auth-constraint>
      <user-data-constraint id="UserDataConstraint_1">
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
      </user-data-constraint>
    </security-constraint>
    <security-constraint id="SecurityConstraint_2">
      <web-resource-collection id="WebResourceCollection_2">
        <web-resource-name>AddressBookApp</web-resource-name>
        <description>Protection area for Rest Servlet</description>
        <url-pattern>/rest/addresses/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint id="AuthConstraint_2">
         <description>Role2 for this rest servlet</description>
         <role-name>Role2</role-name>
      </auth-constraint>
      <user-data-constraint id="UserDataConstraint_1">
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
      </user-data-constraint>
    </security-constraint>
    <security-role id="SecurityRole_1">
         <description>This is Role1</description>
         <role-name>Role1</role-name>
    </security-role>
    <security-role id="SecurityRole_2">
         <description>This is Role2</description>
         <role-name>Role2</role-name>
    </security-role>
    <login-config>
      <auth-method>FORM</auth-method>
      <form-login-config>
         <form-login-page>/logon.jsp</form-login-page>
         <form-error-page>/logonError.jsp</form-error-page>
      </form-login-config>
    </login-config>
</web-app>
```

What to do next

Use the administrative console to administer security for your JAX-RS application.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

  The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

  The default varies based on node type. Common defaults are `configuration_root/AppServer` and `configuration_root/DeploymentManager`.

configuration_root
: Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

  The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

  The default is `/wasv8config/cell_name/node_name`.

plug-ins_root
: Refers to the installation root directory for Web Server Plug-ins.

profile_root
: Refers to the home directory for a particular instantiated WebSphere Application Server profile.

  Corresponding product variables are *server.root* and *user.install.root.*

  In general, this is the same as `app_server_root/profiles/profile_name`. On z/OS, this will always be `app_server_root/profiles/default` because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
: Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

  The corresponding product variable is *smpe.install.root*.

  The default is `/usr/lpp/zWebSphere/V8R5`.

Securing JAX-RS resources using annotations

You can secure Java API for RESTful Web Services (JAX-RS) resources by using annotations that specify security settings.

Before you begin

This task assumes that you have developed the application and identified the JAX-RS resources that you want to secure using annotations for security.

About this task

You can secure JAX-RS resources using annotations for security supported by JSR 250. You can use the following annotations to add authorization semantics to your JAX-RS application resources:

- @PermitAll - specifies that all security roles are permitted to access your JAX-RS resources
- @DenyAll - specifies that no security roles are permitted to access your JAX-RS resources
- @RolesAllowed - specifies the security roles that are permitted to access your JAX-RS resources

You can choose to annotate at the class level or at the method level. The following rules govern the annotations for security:

Method level annotations take precedence over annotations at the class level.

In the following code snippet, the JAX-RS resource that is referenced by the @GET and @Path annotation of `/addresses` and the corresponding `getList()` method is not restricted and open for public consumption. However, the resource referenced by the @PUT and @Path annotations of the `/addresses` and the corresponding `updateList()` method requires the role of `Manager`; for example:

```
@Path(value="/addresses")
        @PermitAll
        public class  AddressBookResource {

          @GET
          @Produces(value="text/plain")
          public String getList() {
          }

          @PUT
          @RolesAllowed("Manager")
          to public void updateList(String[] books) {
          }
        }
```

The annotations for security are mutually exclusive.

This means that each resource is only governed by at most one of annotations for security. For example, the following example is not valid because both @PermitAll and @RolesAllowed are specified:

```
       @Path(value="/addresses")
       @PermitAll
       @RolesAllowed("Employee")
       public class  AddressBookResource {

      @GET
      @Produces(value="text/plain")
      public String getList() {
      }
   }
```

In the previous code example, the @RolesAllowed annotation takes precedence and the @PermitAll annotation is ignored. Similarly, if the @RolesAllowed annotation and @DenyAll annotation are both specified, the @DenyAll annotation takes precedence.

Similarly, if the @PermitAll and @DenyAll annotations are both specified at the method or at the class level, the @DenyAll annotation takes precedence as it ensures security by conforming to the safe default principle.

If the @PermitAll, @DenyAll and @RolesAllowed annotations are all present at the method or class level the @DenyAll annotation takes precedence over @RolesAllowed and @PermitAll. The order of precedence of these annotations is the following:

1. @DenyAll
2. @RolesAllowed
3. @PermitAll

Rule for inheritance

JSR 250 annotations that are added at the class level only affect the classes that they annotate and the corresponding methods for subresources. Annotations that are specified at the class level do not affect resources that are inherited from a superclass.

Rule for overriding method(s)

Annotations on resources that correspond to overridden methods in subclasses take precedence

over annotations that are included in the parent class. In the following snippet, the `LocalAdministrator` role is used to access the `/addresses/local` subresource; for example:

```
s@Path(value="/addresses")
@PermitAll
public class  AddressBookResource {

   @GET
   @Produces(value="text/plain")
   public String getList() {
   }

   @PUT
   @RolesAllowed("Administrator")
   public void updateList(String books) {

   }
 }

 @Path(value="/addresses")
 @PermitAll
 public class  LocalAddressBookResource
               extends AddressBookResource {

   @PUT
   @RolesAllowed("LocalAdministrator")
   @Path(value="local")
   public void updateList(String books){

   }
 }
```

@RolesAllowed consideration

You cannot have multiple @RolesAllowed annotations simultaneously on a resource. For example, you can achieve:

```
@RolesAllowed("role1")
 @RolesAllowed("role2")
  public String foo() {
  }
```

using the following code snippet:

```
@RolesAllowed({"role1", "role2"})
 public String foo() {
  }
```

Considerations for the use of annotations for security and the configuration of security constraints

Annotations for security follow the declarative security model. Security constraints that are configured in the deployment descriptor, the web.xml file, take precedence over security constraints that are programmatically annotated in the application. It is important for developers of JAX-RS resources to consider a balance across configurable security constraints and annotated security constraints. Annotated constraints are additional to any configured security constraints. The JAX-RS runtime environment checks for annotated constraints after the web container runtime environment has checked for security constraints that are configured in the web.xml file.

Configure authentication constraints in the web.xml file. In the following example web.xml file, the `SecurityConstraint_1` security constraint is defined. This constraint is used to require authentication to the application. Additionally, the `SecurityConstraint_1` security constraint defines constraints on URL patterns corresponding to JAX-RS resources. When a JAX-RS resource is accessed that corresponds to one of these constraints, authorization checks are performed. Access checks are performed for the declarative security annotations only after the configured constraints are verified.

```
<web-app id="WebApp_someID">
<servlet>
  <servlet-name>AddressBookAppSample</servlet-name>
  <servlet-class>
       org.apache.wink.server.internal.servlet.RestServlet
  </servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>jaxrs.sample.AddressBookApplication
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
  </servlet>
```

```
    <servlet-mapping>
        <servlet-name>AddressBookApp</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
    <security-constraint id="SecurityConstraint_1">
      <web-resource-collection id="WebResourceCollection_1">
        <web-resource-name>AddressBookAppSample</web-resource-name>
        <description>Protection area for Rest Servlet</description>
        <url-pattern>/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
        <http-method>PUT</http-method>
      </web-resource-collection>
      <auth-constraint id="AuthConstraint_1">
         <description>Role1 for this rest servlet</description>
         <role-name>Role</role-name>
      </auth-constraint>
      <user-data-constraint id="UserDataConstraint_1">
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
      </user-data-constraint>
    </security-constraint>
    <security-role id="SecurityRole_1">
         <description>This Role is used to drive authentication
         </description>
         <role-name>Role1</role-name>
    </security-role>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>test realm</realm-name>
 </login-config>
    </login-config>
</web-app>
```

In the previous sample web.xml file, `Role1` is used for the entire application. If you are only defining declarative security annotations and you are not using authorization constraints from the web.xml file, you can map this role for the JAX-RS application to the AllAuthenticated special subject for user authentication.

Procedure

1. Determine if there are security constraints defined by the web.xml file for your JAX-RS application.
2. Configure the web.xml file to add security constraints. Security constraints that are configured in the deployment descriptor, the web.xml file, take precedence over security constraints that are programmatically annotated in the application.
3. Determine if you want to add annotations for security, in addition to any constraints in the web.xml file. Decide if you want to add one of the @PermitAll, @DenyAll and @RolesAllowed annotations to provide additional security for your JAX-RS resources. Consider the rules for adding annotations for security such as precedence and inheritance described previously.

Results

You have defined secure JAX-RS resources using declarative security annotations.

Example

The following code snippet demonstrates how you can use security annotations to protect JAX-RS resources. In this example, the `/addresses` root resource is associated with a @PermitAll annotation and therefore the subresource that corresponds to the `@GET` and `@Produces(value="text/plain")` methods is permitted to all users because this resource does not introduce security annotations of its own. However, the subresource that corresponds to the @PUT method is associated with its own @RolesAllowed annotation and requires the `Administrator` role.

```
@Path(value="/addresses")
 @PermitAll
 public class  AddressBookResource {

   @GET
   @Produces(value="text/plain")
       public String getList() {
       }


       @RolesAllowed("Administrator")
       @PUT
```

```
    public void updateList(String books) {

    }
}
```

Securing downstream JAX-RS resources

You can secure downstream Java API for RESTful Web Services (JAX-RS) resources by configuring the BasicAuth method for authentication and by using the LTPA JAX-RS security handler to take advantage of single sign-on for user authentication.

Before you begin

This task assumes that you have completed the following steps:

- You have defined to your application server a cell profile that is federated into the deployment manager cell.
- You have installed your JAX-RS application onto the application server.
- You have enabled security for your JAX-RS application.
- You have secured your JAX-RS applications within the web container by configuring downstream JAX-RS applications to use the basic authentication (BasicAuth) method for user authentication.

About this task

When composing JAX-RS resources, a new LTPA JAX-RS security handler can be used to seamlessly authenticate on downstream resource invocations.

When invoking downstream secure JAX-RS resources, the calling application is required to authenticate to the target resource. If the target resource on a downstream server uses the BasicAuth method for security, the calling application can take advantage of single sign-on (SSO) for JAX-RS resources. Using single sign-on, an authenticated context is propagated along downstream calls. You can use the LTPA-based security client handler to authenticate to downstream resources that are distributed across servers of a cell environment.

To illustrate this scenario, assume that you have two servers in your cell and that you have deployed JAX-RS resources on both of these servers. Suppose from one resource on `server1` you need to invoke another resource that is deployed on `server2`. When `server2` resources are secured using the BasicAuth method for authentication, use the LTPA JAX-RS security handler to take advantage of single sign-on and seamlessly propagate user authentication on downstream calls without having to provide or manage user identities and passwords in the application.

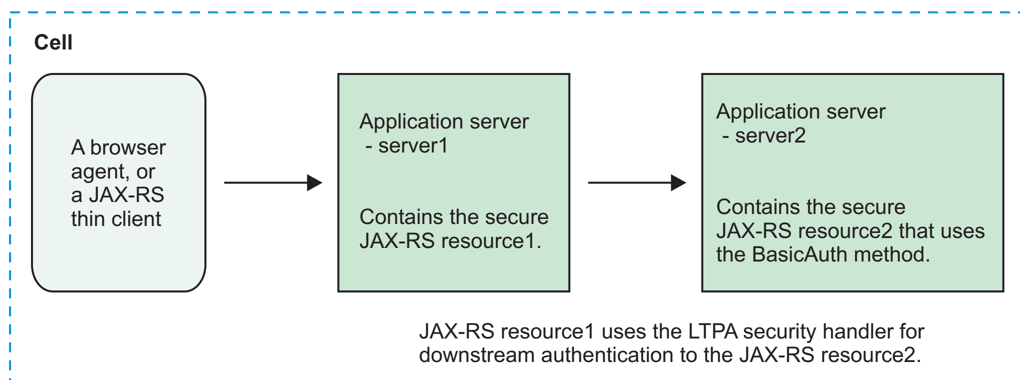


Figure 1. Securing JAX-RS downstream resources

Use the following steps to configure user authentication to a downstream server using the JAX-RS security handler at application build time.

Procedure

1. At application build time, use the LTPA-based security client handler, `LtpaAuthSecurityHandler`, to authenticate to downstream resources that are distributed across servers of a cell environment.

   When using the `LtpaAuthSecurityHandler` class, ensure that you target resources using the https scheme for your URLs, and that the target application is SSL-enabled. It is highly recommended to use SSL connections when sending user credentials, including LTPA cookies. You may explicitly turn off the requirement for SSL in the `LtpaAuthSecurityHandler` class by invoking the `setSSLRequired` method on the security handler with the `false` value. The default value is `true`.

```
yourLtpaAuthSecHandler.setSSLRequired(false);
```

2. Add the security handler to the handlers chain.
3. Create the REST client instance.
4. Create the resource instance that you want to interact with.
5. Substitute a value representing your resource address.

Results

You have defined secure JAX-RS resources within your cell environment such that when downstream resources are invoked, you can use single sign-on and seamlessly propagate user authentication on downstream calls without having to provide or manage user identities and passwords in the application.

Example

The following code snippet demonstrates how to use this security handler that is packaged as part of the JAX-RS client.

```
import org.apache.wink.client.Resource;
 import org.apache.wink.client.RestClient;
 import org.apache.wink.client.ClientConfig;
 import org.apache.wink.client.handlers.LtpaAuthSecurityHandler;

 ClientConfig config = new ClientConfig();
 LtpaAuthSecurityHandler secHandler = new LtpaAuthSecurityHandler();

 // Add this security handler to the handlers chain.
 config.handlers(secHandler);

 // Create the REST client instance.
 RestClient client = new RestClient(config);

 // Create the resource instance that you want to interact with.
 // Substitute a value representing your resource address
 resource =
  client.resource("http://localhost:8080/path/to/resource");

// Now you are ready to begin calling your resource.
```

Securing JAX-RS clients using SSL

You can secure the communications between your Java API for RESTful Web Services (JAX-RS) application and clients that invoke the application by using Secure Sockets Layer (SSL) transport layer security.

Before you begin

This task assumes that you have completed the following steps:

- You have defined a cell profile to an application server or to an application server that is federated to a network deployment manager. Read about creating cell profiles to learn how to create cell profiles that contain a federated application server node and a deployment manager.
- You have installed your JAX-RS application onto the application server.

About this task

JAX-RS client programs can take advantage of transport security using Secure Socket Layer (SSL) in order to protect requests and responses from JAX-RS resources.

If you have configured your JAX-RS application to use an SSL channel for transport level security when starting REST resources, your JAX-RS client is required to use the SSL connection to enable the client to interact with a JAX-RS resource that is deployed in the WebSphere Application Server environment. For example, if your JAX-RS application is configured to use basic authentication, it is a common practice to use SSL so that the user credentials are transported over secure connections.

To illustrate this scenario, assume that you have one application server in your cell, and that you have deployed JAX-RS resources on this server. The JAX-RS resources on this server requires the use of SSL. Suppose that you are using the Thin Client for JAX-RS, a Java-based stand-alone client that is supplied with this product, to invoke one of these secure resources that requires the use of SSL. The Thin Client for JAX-RS enables running unmanaged JAX-RS RESTful web services client applications in a non-WebSphere environment to invoke JAX-RS RESTful web services that are hosted by the application server.

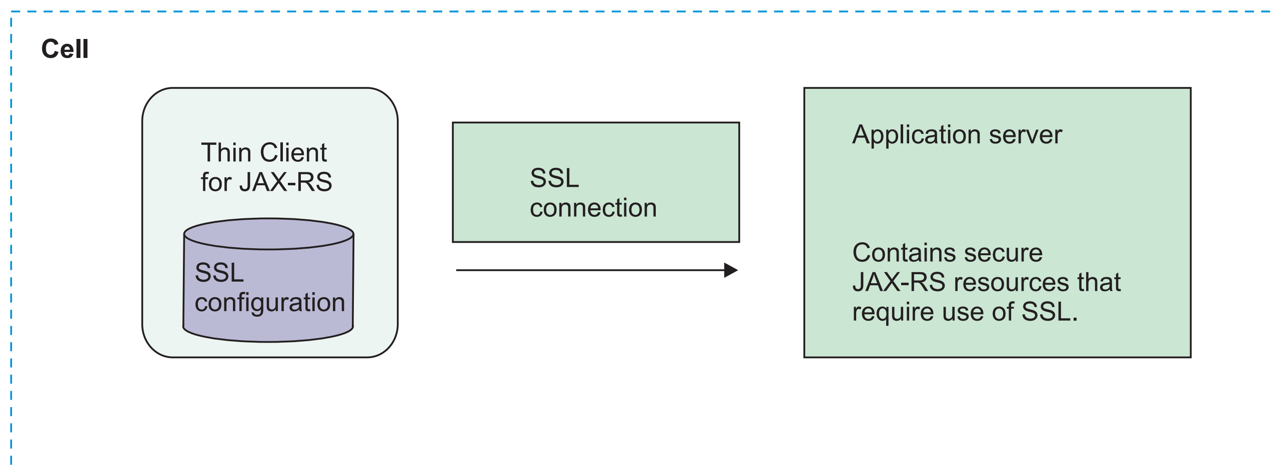


Figure 2. Securing JAX-RS clients using SSL

Important: If you are invoking JAX-RS resources from within a application that is running in a WebSphere Application Server environment, such as when you are making a downstream call, no additional configuration for SSL is necessary. You do not need to configure SSL connections for this resource because the application server SSL runtime and configuration is used.

Use the following steps to configure SSL with the Thin Client for JAX-RS.

Procedure

1. Enable security for your JAX-RS application and configure your application to use an SSL channel for transport when invoking REST resources.

At application development or deployment time, edit the web.xml file to add a security constraint that requires use of SSL for your resources. See the securing JAX-RS applications within the web container information for additional details on enabling SSL for your application.

The following element within the security-constraint element specifies to enforce SSL for your application:

```
<user-data-constraint id="UserDataConstraint_1">
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
 </user-data-constraint>
```

2. Edit the ssl.client.props file and define the keystore and the truststore properties.

   The ssl.client.props file is used to configure SSL for clients. The following code example illustrates defining the keystore and the truststore properties:

```
# keystore information
com.ibm.ssl.keystoreName=ClientDefaultKeyStore
com.ibm.ssl.keyStore= path/to/keystore/file
com.ibm.ssl.keyStorePassword=xxxxxxx
com.ibm.ssl.keyStoreType=PKCS12
com.ibm.ssl.keyStoreProvider=IBMJCE
com.ibm.ssl.keyStoreFileBased=true

# truststore information
com.ibm.ssl.trustStoreName=ClientDefaultTrustStore
com.ibm.ssl.trustStore=path/to/truststore/file
com.ibm.ssl.trustStorePassword=xxxxxx
com.ibm.ssl.trustStoreType=PKCS12
com.ibm.ssl.trustStoreProvider=IBMJCE
com.ibm.ssl.trustStoreFileBased=true
com.ibm.ssl.trustStoreReadOnly=false
```

3. Enable or disable host name verification.

   The `com.ibm.ssl.performURLHostNameVerification` property enforces URL host name verification when the value is set to `true`. When HTTP URL connections are made to target servers, the common name (CN) from the server certificate must match the target host name. Without a match, the host name verifier rejects the connection. The default value of `false` omits this check.

   The `com.ibm.ssl.validationEnabled` property validates each SSL configuration as it is loaded when the value is set to `true`. The default value of `false` omits this check.

```
com.ibm.ssl.performURLHostNameVerification=false
com.ibm.ssl.validationEnabled=false
```

4. Ensure that the signer of the server certificate is in the client truststore.

   Use the IBM ikeyman tool or the Java keytool utility to determine if the certificate is already in the truststore. If the certificate is not in the truststore, import the certificate into the truststore.

   For example, to list the certificates that are contained in truststore, `trust.p12` type the following command and ensure that you include the full path to your truststore:

```
keytool -list -v -storetype pkcs12 -keystore trust.p12
```

5. Import the certificate.

   If the signer of the server certificate is not in the client truststore, or if the server has a self-signed certificate that is not in the client truststore, import the certificate.

   To import the certificate, you can use your preferred tooling of either the IBM ikeyman or the Java keytool utility. The following examples use the Java keytool utility.

   a. Export the signer certificate for your server to a file.

      For example, use the following command to export a signer certificate from an existing truststore, `servertrust.p12`, in the entry that corresponds to the default_signer alias name into the file `mycert.cer`:

```
keytool -export -storetype pkcs12 -alias default_signer -file mycert.cer -keystore servertrust.p12
```

   b. Import the signer certificate into the truststore used by your Thin Client for JAX-RS.

      For example, use the following command to export a signer certificate from an existing truststore, `servertrust.p12`, from the entry that corresponds to the default_signer alias name into the file `mycert.cer`:

```
keytool -export -storetype pkcs12 -alias default_signer -file mycert.cer -keystore servertrust.p12
```

Results

You have defined a secure connection between the client and the target server using SSL to enable integrity and confidentiality of the communication between the JAX-RS application and your client.

Example

The following code snippet demonstrates a sample ssl.client.props file:

```
# keystore information
com.ibm.ssl.keyStoreName=ClientDefaultKeyStore
com.ibm.ssl.keyStore=c:/jaxrs/test/config/keystore.p12
com.ibm.ssl.keyStorePassword=testpasswd
com.ibm.ssl.keyStoreType=PKCS12
com.ibm.ssl.keyStoreProvider=IBMJCE
com.ibm.ssl.keyStoreFileBased=true

# truststore information
com.ibm.ssl.trustStoreName=ClientDefaultTrustStore
com.ibm.ssl.trustStore= c:/jaxrs/test/config/truststore.p12
com.ibm.ssl.trustStorePassword=testpasswd
com.ibm.ssl.trustStoreType=PKCS12
com.ibm.ssl.trustStoreProvider=IBMJCE
com.ibm.ssl.trustStoreFileBased=true
com.ibm.ssl.trustStoreReadOnly=false

# Host name verification information
com.ibm.ssl.performURLHostNameVerification=false
com.ibm.ssl.validationEnabled=false
```

Administering secure JAX-RS applications

You can use the administrative console to administer Java API for RESTful Web Services (JAX-RS) applications that have enabled security mechanisms.

Before you begin

This task assumes familiarity with the Sample REST application that is used in the Securing JAX-RS applications within the web container topic and the security mechanisms applied to this JAX-RS application.

About this task

After you have implemented security mechanisms, such as basic HTTP authentication or role-based authorization constraints on your REST resources, you can administer your JAX-RS applications by mapping defined roles to users, groups, or special subjects.

Procedure

1. In the administrative console, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Detail properties, click **Security role to user/group mapping**. A list of all the roles that belong to this application is displayed.
3. Select one of the roles you defined for your application.

   In the AddressBookApp Sample, the defined roles are `Role1` and `Role2`.
4. Determine the users, groups, or special subjects such as the **All Authenticated in Application's Realm** option to assign the appropriate roles. This option specifies that any authenticated user is able to access the resource. The security constraint in this Sample is for authentication only.
5. Repeat the previous steps for every role that you have defined in your JAX-RS application.
6. Click **OK** to save your changes.

Results

Using the administrative console, you have applied role constraints to various resource URI patterns to enable role-based access to those resources.

Defining and managing secure policy set bindings

Configuring the SSL transport policy

When working with policy sets in the administrative console, you can customize policies to ensure message security by configuring the SSL transport policy.

Before you begin

The default policy sets provided with the product cannot be edited. To configure custom policy sets, you must first copy the default policy set or create a completely new policy set in order to specify the policies for it. See creating policy sets using the administrative console.

About this task

The SSL transport policy provides the SSL transport security for the Hypertext Transfer Protocol (HTTP) protocol with web services applications. To view the default SSL transport policy set with the SSL transport policy, click **Services > Policy sets > Application policy sets > WSHTTPS default > SSL transport**.

Procedure

1. To edit the SSL transport policy, click a policy set that you have created or customized from the default. Select the SSL transport policy applicable check boxes to enable the SSL functions. The following check boxes determine how SSL security is configured for this transport:
   - **Enable for outbound service requests**

     Displays whether the SSL security transport is enabled for outbound service requests.
   - **Enable for outbound asynchronous service responses**

     Displays whether the SSL security transport is enabled for outbound asynchronous service responses.
   - **Enable for inbound service responses**

     Displays whether the SSL security transport is enabled for inbound service responses.
2. To configure the binding for the SSL transport policy, click **Services > Policy sets > General client policy set bindings** > *binding_name* > **SSL transport** or **Services > Policy sets > General provider policy set bindings** > *binding_name* > **SSL transport**. Select the setting to configure the SSL bindings. The SSL transport window displays options for configuring the SSL security bindings.
   a. Select the setting to configure the SSL bindings for the **Outbound service requests**.
      - **SSL settings**

        Specifies the SSL security transport binding that is enabled for outbound service requests. The default value for this field is **CellDefaultSSLSettings**.
      - **SSL properties file path**

        Specifies the path of the SSL properties file that is enabled for asynchronous service responses. Enter the location of the SSL properties file to enable for asynchronous service responses.

   b. Select the setting to configure the SSL bindings for the **Inbound service responses**.
      - **SSL settings**

        Specifies the SSL security transport binding that is enabled for inbound service responses. The default value for this field is **CellDefaultSSLSettings**.
      - **SSL properties file path**

Specifies the path of the SSL properties file that is enabled for inbound service responses. Enter the location of the SSL properties file to enable for inbound service responses.

c. Select the setting to configure the SSL bindings for the **Outbound asynchronous service responses**.

- **SSL settings**

  Specifies the SSL security transport binding that is enabled for asynchronous service responses. The default value for this field is **CellDefaultSSLSettings**.

- **SSL properties file path**

  Specifies the file path of the SSL properties file that is enabled for outbound service requests. Enter the location of the SSL properties file to enable for outbound service requests.

Custom properties

Click one of the following buttons to enable the action described:

Button	Resulting Action
New	Creates a new custom property entry. To add a custom property, enter the name and value.
Delete	Removes the selected custom property.
Edit	Enables you to edit a selected custom property. It is only displayed when one or more properties exist.

Results

Once you have customized the SSL transport policy, the associated policy set uses this policy to protect message transmission. Similarly, you can also configure HTTP transport with the HTTP transport policy. Read about configuring the HTTP transport policy to learn how to configure the HTTP transport with the HTTP transport policy.

What to do next

Depending on how you are using policies, you might want to configure the HTTP transport policy or the SSL transport security bindings.

SSL transport security policy settings

Use this page to define the secure sockets layer (SSL) transport policy configuration for policy sets.

To configure the SSL transport security for a policy set, click **Services > Policy sets > Application policy sets** > *policy_set_name* > **SSL transport**, where *policy_set_name*, applies to any policy set that contains SSL transport security. An example of such SSL transport security is WSHTTPS default.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Enable for outbound service requests:

Specifies whether the SSL security transport is enabled for outbound service requests when the client sends out requests.

Enable for outbound asynchronous service responses:

Specifies whether the SSL security transport is enabled for outbound asynchronous service responses when the service or server sends back the response.

Enable for inbound service responses:

Specifies whether the SSL security transport is enabled for inbound service responses when the client receives responses.

SSL transport security settings

Use this page to define the secure sockets layer (SSL) transport policy binding configuration.

1. Navigate to the general bindings collection page by clicking either **Services > Policy sets > General client policy set bindings** or **Services > Policy sets > General provider policy set bindings** path.
2. Click a general binding in the Name column.
3. Click the **SSL transport** policy in the Policies table.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Outbound service requests – SSL settings:

Specifies the SSL security transport binding that is enabled for outbound service requests when the client sends out requests. The default value for this field is `CellDefaultSSLSettings`.

Outbound service requests – SSL properties file path:

Specifies the file path of the SSL properties file that is enabled for outbound service requests. Enter the location of the SSL properties file to enable for outbound service requests.

Inbound service responses – SSL settings:

Specifies the SSL security transport binding that is enabled for inbound service responses when the client receives responses. The default value for this field is `CellDefaultSSLSettings`.

Inbound service responses – SSL properties file path:

Specifies the SSL security transport binding that is enabled for inbound service responses. Enter the location of the SSL properties file to enable for inbound service responses.

Outbound asynchronous service responses – SSL settings:

Specifies the SSL security transport binding that is enabled for asynchronous service responses when the service or server sends back the response. The default value for this field is `CellDefaultSSLSettings`.

Outbound asynchronous service responses – SSL properties file path:

Specifies the path of the SSL properties file that is enabled for asynchronous service responses. Enter the location of the SSL properties file to enable for asynchronous service responses.

Outbound asynchronous service responses – Custom properties:

Specifies the name and value pair that you define for the outbound asynchronous service responses. Click **New** to add a new custom property, or click **Delete** to delete an existing SSL custom property.

Configuring SCA web service binding for transport layer authentication

Use this task to configure a web service binding to perform transport-layer HTTP basic authentication.

Before you begin

Before you begin this task, install a Service Component Architecture (SCA) application.

About this task

Intents and policy sets can be used to configure web service bindings to achieve quality of services (QoS).

Procedure

1. Configure administrative and application security for the server.

   To secure the service so that it only accepts secure requests, and for the service to require authentication, administrative and application security must be enabled for the server.
2. Configure the service binding to require transport-layer authentication by specifying the `authentication.transport` intent for an OSOA composite, or the `clientAuthentication.transport` intent for an OASIS composite.

   OSOA example

   ```
   <service name="AccountService">
     <binding.ws
        requires="authentication.transport"
        ... />
   </service>
   ```

   OASIS example

   ```
   <service name="AccountService">
     <binding.ws
        requires="clientAuthentication.transport"
        ... />
   </service>
   ```
3. Configure the reference binding to send a username and password by attaching a policy set and client policy set binding that includes the HTTPTransport policy type.

   ```
   <reference name="AccountService">
     <binding.ws
        qos:wsPolicySet="My HTTP Policy Set"
        qos:wsReferencePolicySetBinding="My HTTP Client Binding"
        ... />
   </reference>
   ```

   The client policy set binding must be configured with the username and password to send with the request. See "Configuring the HTTP transport binding" for information about how to create a policy set and binding for the HTTP transport.

   For additional information on attaching policy sets to the `binding.ws` element, refer to "Mapping abstract intents and managing policy sets."

Results

When you finish this task, you have configured the web service binding to do SCA transport layer authentication.

What to do next

You can proceed to configuring other application specific bindings for your policy sets.

Transformation of policy and binding assertions for WSDL

Web Services Security does not fully support the OASIS WS-SecurityPolicy Version 1.2 standard. However, several of the policy and binding assertions supported by WebSphere Application Server can be transformed and represented as WS-SecurityPolicy Version 1.2 assertions. The supported assertions are transformed when a Web Services Description Language (WSDL) or Web Services Metadata Exchange (WS-MEX) request is received in a message, and also when the client receives a policy containing WS-SecurityPolicy 1.2 assertions.

When the WebSphere Application Server receives a WSDL or WS-MEX request, some policy and binding assertions are transformed into standard assertions and included in the policy that is embedded into the

WSDL. In addition, when the client receives a policy containing these WS-SecurityPolicy assertions, the assertions are transformed back into product-specific assertions so that the Application Server run time can process them. This transformation provides interoperability between Application Server and other systems that support the WS-SecurityPolicy version 1.2 standard.

The following WS-SecurityPolicy 1.2 assertions can be represented in the policy returned on WSDL, or a WS-MEX request.

EncryptSignature
: Represented in the product using XPath expressions in the encrypted elements.

EncryptBeforeSigning
: The order attribute of the encryptionInfo and signingInfo elements on the outbound section of the bindings determines the order in which the transform takes place, and whether this assertion is set. The default behavior is to sign before encrypting.

ContentEncryptedElements
: Represented using XPath expressions ending with /node() in the encrypted elements. The Application Server can consume this policy assertion, but existing XPath expressions that end with /node() are not transformed.

KerberosToken
: Represented using the custom token in the policy and bindings. The Kerberos custom token assertion specifies a local name of http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ, or http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ, depending on the desired Kerberos token type. Also, there is a custom property in the bindings, com.ibm.wsspi.wssecurity.krbtoken.requireDerivedKey, which specifies use of derived keys for Kerberos. Using the local name of the custom token, along with the derived key custom property from the product representation, an equivalent Version 1.2 representation can be created.

Require*<variable>*Reference, where *<variable>* is one of the following: KeyIdentifier, IssuerSerial, EmbeddedToken, or Thumbprint
: Rrepresented using the type attribute on keyInfo in the bindings.

MustSupportRef*<variable>*, where *<variable>* is one of the following: KeyIdentifier, IssuerSerial, EmbeddedToken, or Thumbprint
: This assertion is not represented in the WebSphere Application Server policies, but the product supports all four types of references.

Protection of the SignatureConfirmation element
: The SignatureConfirmation element is implicitly signed and encrypted. However, if nothing is encrypted on the response, then the SignatureConfirmation element is not encrypted, and if nothing is signed on the response, then the SignatureConfirmation element is not signed. All XPath expressions representing the signing or encryption of the SignatureConfirmation element are removed from the policy during transformation. The explicitlyProtectSignatureConfirmation attribute in the Web Services Security binding is provided to disable implicit signature and encryption of the SignatureConfirmation element on the response message. This provides interoperability with WebSphere Application Server Version 6.1.x. To add the attribute, check the option **Disable implicit protection for Signature Confirmation** in the Authentication and protection panel for the default policy set bindings. If the explicitlyProtectSignatureConfirmation attribute is present in the binding, all XPath expressions representing the signing or encryption of the SignatureConfirmation element remain unchanged during transformation.

SC13SecurityContextToken
: Version 1.3 of the Security Context Token is supported by specifying the local name http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512, in the binding for the security context token.

RequireImpliedDerivedKeys
This is supported by adding the custom property, com.ibm.ws.wssecurity.token.generateImpliedDerivedKey, in the token generator bindings.

ExactlyOne
This assertion is transformed when callers are used. Callers specify which token to use for authentication. The ExactlyOne assertion communicates the caller tokens that the service expects. All caller options are enclosed inside the <ExactlyOne> assertion, and each option is enclosed inside the <wsp:All> assertion. As the name implies, the client sends only one of the token types. For example, in the server side bindings, using the product representation, the following caller options are specified:

```
<caller order="1">
       <jAASConfig configName="system.wss.caller"/>
       <callerIdentity uri="" localName="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken"/>
</caller>
<caller order="2">
       <jAASConfig configName="system.wss.caller"/>
       <callerIdentity localName="LTPA" uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2" />
</caller>
```

This assertion indicates that a UsernameToken token or an LTPA token is used as the caller. The requirement to use one of these two types of tokens is communicated to the client in the transformed policy, as in the following example:

```
<sp:ExactlyOne>
<wsp:All>
<sp:SupportingTokens>
   <wsp:Policy wsu:Id="request:token_auth">
     <sp:UsernameToken sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/IncludeToken/AlwaysToRecipient">
       <wsp:Policy>
         <sp:WssUsernameToken10 />
       </wsp:Policy>
     </sp:UsernameToken>
 </wsp:Policy>
</sp:SupportingTokens>
</wsp:All>
<wsp:All>
<sp:SupportingTokens>
<wsp:Policy wsu:Id="request:token_auth">
     <spe:LTPAToken sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/IncludeToken/AlwaysToRecipient" />
</wsp:Policy>
 </sp:SupportingTokens>
<wsp:All>
</sp:ExactlyOne>
```

The product-specific policy assertions LTPAToken and LTPAPropagationToken are not altered during transformation. These assertions are included in the embedded policy in the WSDL if they are present in the policy being transformed. This allows a WebSphere Application Server client and WebSphere Application Server service provider to interoperate.

Securing message parts using the administrative console

If you are working with policy sets, then you can secure message parts using the administrative console. To secure message parts with WS-Security using policy sets, you must define the elements for the message parts to be protected in the WS-Security policy within a policy set.

Before you begin

Before you can start this task, you must have a policy set defined for your application or service artifact. Also, if none of the default policy sets contain the necessary policy definitions, then you must create a custom policy set with the necessary definitions.

About this task

This task assumes that you are using policy sets and you want to secure message parts within that context.

Procedure

1. Open the administrative console.
2. Select the policy set containing the message parts that you want to secure.
   - To secure message parts using application policy sets click **Services > Policy sets > Application policy sets**.
   - To secure message parts using system policy sets click**Services > Policy sets > System policy sets**.
3. Select the policy set that you want to use.
4. If the WS-Security policy is not listed, then click **Add** and select that policy from the list.
5. Click the **WS-Security** link.
6. Click **Main policy** or **Bootstrap policy**. The bootstrap policy is available when Secure Conversation is used. If you want to use the bootstrap policy, then select the SecureConversation policy set in step three.
7. Make sure that Message level protection is selected, then click **Request message part protection** or **Response message part protection**. When the Message level protection checkbox is unchecked, the link to Response message part protection is not available, because the configuration information associated with message level security is removed when Message level protection is deselected.
8. Click **Add** for either Encrypted parts or Signed parts depending on the level of security that you want.
9. Specify a part name and add the elements to be signed or encrypted, or both. The elements can be the message body, XPath expression, or a QName which is for SOAP header elements only. Click **OK**. Recommendation for when to use QName or XPath: If you are encrypting or signing SOAP headers, you can use QName to select which SOAP headers to be signed or encrypted.

   Note: The elements must be a direct child of the SOAP headers.

   If you wanted to sign and encrypt other elements in the SOAP message, then you can use XPath expression. Use this XPath example to select, `MyElement` in a namespace, `http://xyz.acme.com` with `MyHeader`, `http://acme.com`.

```
/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope' and local-name()='Envelope']/*[namespace-uri()=
'http://www.w3.org/2003/05/soap-envelope' and local-name()='Header']/*[namespace-uri()='http://acme.com' and local-name()=
'MyHeader']/*[namespace-uri()='http://xyz.acme.com' and local-name()='MyElement']
```

10. Repeat steps 8 and 9 to sign or encrypt each message part.
11. To save your changes to the master configuration, click **Save**.

Results

When you finish this task, you have configured the policy set that contains the quality of service definitions required for signing and encrypting message parts.

Example

If you have the policy set, **myPolicy** and you want to specify request message bodies that must be signed, you can perform the following:

1. Locate the policy set in the **Services > Policy sets > Application policy sets** collection and click the policy set name.
2. Click the **WS-Security** link. If the link does not exist, click **Add** and then select **WS-Security** from the list.
3. Click **Main policy** > **Request message part protection**
4. Click **Add** under the Integrity protection and Signed parts section.
5. Specify the name, `messageBody`.
6. Select **Protect message body**, click **Add Specified Elements**, and click **OK**.
7. Click **Save** to save your changes to the master configuration.

What to do next

You can proceed to signing and encrypting message parts using policy sets.

Signing and encrypting message parts using policy sets

With web services, you can sign message parts, encrypt message parts, or both, based on the quality of service defined for a policy set. You can accomplish these actions by defining the binding information in a custom attachment binding.

Before you begin

Before you begin this task, attach a policy set to a service artifact such as an application, service or endpoint and create a custom attachment binding. Read about creating custom attachment bindings for policy sets. The policy set that is attached to the service artifact must include a WS-Security policy that specifies message parts to be signed or encrypted. Read about securing message parts using the administrative console.

About this task

To sign message parts, encrypt message parts, or both, based on the quality of service defined for a policy set, perform the following steps:

Procedure

1. Open the administrative console.
2. To sign and encrypt message parts for a service provider, click **Applications > Enterprise applications** > ***application_name*** > **Service provider policy sets and bindings**. To sign and encrypt message parts for a service client, click **Applications > Enterprise applications** > ***application_name*** > **Service client policy sets and bindings**.
3. Click the binding name link of the service artifact with a custom attachment binding.
4. If the binding does not contain WS-Security policy set bindings, then click **Add** and select **WS-Security** from the list.
5. Click **WS-Security** policy set bindings.
6. Click **Authentication and protection**. The resulting panel contains the following four tables:
   - Protection tokens: Specifies the tokens that are defined for the symmetric or asymmetric signature and encryption policies in the policy set.
   - Authentication tokens: Specifies the tokens that are defined for the request and response token policies.
   - Request message signature and encryption protection: Specifies the message parts that are defined in the Request message part protection for the policy set.
   - Response message signature and encryption protection: Specifies the message parts that are defined in the response message part protection in the policy set.

   Initially, each table displays information that is generated based on the policy set which is attached to the service artifact. The possible configuration objects based on the policy set are displayed. The Status column indicates whether the object is currently configured in the custom attachment binding.
7. If the protection tokens have a status of **Not configured**, then create the protection tokens by clicking the default name, verifying the default values. Click **OK**.
8. [Optional] If you use the X.509 protection tokens, then you must configure the keystores and keys to be used to sign, verify, encrypt or decrypt message parts. You might need to also configure keystores and keys when using custom protection tokens, depending on the requirements of the custom tokens. When using a security context token for protection (secure conversation), you do not need to configure keystores or keys. If you need to configure the keystores and keys, then perform the following actions:

a. Click the token name link.
b. Click the **Callback handler** link under Additional bindings. If the Callback handler link is not click-able, click **Apply**, then click the **Callback handler** link.
c. Either use a predefined keystore or custom keystore. To use a predefined keystore, select the keystore from the list. To use a custom keystore, select **Custom** from the list and click the **Custom key store configuration** link to specify the configuration.
d. Click **OK**.

9. Click the name of the request or response message part reference to be signed or encrypted. The Protection column displays whether the message part is signed or encrypted based on the policy set.
10. Specify a name for the message part.
11. For encrypted parts, select the type of encryption from **Usage of key information references**. For asymmetric encryption, or X.509, select **Key encryption**. For symmetric encryption, or secure conversation, select **Data encryption**.
12. [Optional] For encrypted parts, select the **Include time stamp** or **Include nonce** options to include a time stamp or nonce in the encrypted message part. You can include one or both of these options in the encrypted message part.
13. For signed parts, specify one or more Message part references. Select a reference from the Available column and click **Add**.
14. [Optional] For signed parts, you can also choose to add a time stamp or nonce to the signed message part. Select a Message part reference from the Assigned column and click **Edit**. Select the **Include time stamp** or the **Include nonce** options to include a time stamp or nonce in the signed message part. You can select one or both of these options in the signed message part.
15. If there are no available key information entries, then create one using the following actions:
    a. Click **New**.
    b. Specify a name.
    c. Select a protection token from the Token generator or Consumer name list.
    d. Click **OK**.
16. Select a key information entry from the Available list and click **Add**.
17. [Optional] Specify custom properties if needed.
    a. To use Message Transmission Optimization Mechanism (MTOM) for the cipher text of the encrypted data, add the custom property, com.ibm.wsspi.wssecurity.enc.MTOM.Optimize, with value `true` to outbound encrypted parts for client requests or server responses.
    b. To use encryption headers as described in the WS-Security 1.0 specification instead of the encrypted header support described in WS-Security 1.1, add the custom property, com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.0, with value `true` to outbound encrypted parts for client requests or server responses.

       For Web Services Security Version 1.1 behavior that is equivalent to WebSphere Application Server versions prior to version 7.0, specify the com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.1.pre.V7 property with a value of `true` on the <encryptionInfo> element in the binding. When this property is specified, the <EncryptedHeader> element includes a wsu:Id parameter and the <EncryptedData> element omits the Id parameter. This property should only be used if compliance with Basic Security Profile 1.1 is not required.
18. Click **OK**.
19. Click **Save**, to save the changes to the master configuration.

Results

When you finish this task, the message parts are signed and encrypted, or both, based on the configuration used when communicating with the service artifact.

Example

You have an application, `app1`, with an attached policy set, RAMP default and a custom attachment binding, `myBinding`, and you want to sign and encrypt the message parts.

1. Click the `app1` application in the **Applications** > **Enterprise Applications** collection.
2. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** link.
3. Click the `myBinding` link.
4. [Optional] If WS-Security is not listed, then select **Add** > **WS-Security**.
5. Click the **WS-Security** link.
6. Click the **Authentication and protection** link.
7. In the Protection tokens table, click each of the four links and **OK** on the resulting panel. Each entry is now shown as **Configured** in the Status column.
8. In the Request message signature and encryption protection table, click **request:app_encparts**. Specify the name, `requestEncParts`.
9. Click **New** from Key information. Specify the name, `requestEncKeyInfo`.
10. Select **SymmetricBindingRecipientEncryptionToken**, and click **OK**.
11. Select **requestEncKeyinfo** in the Available list, and click **Add**. Click **OK**.
12. In the Request message signature and encryption protection table, click **request:app_signparts**.
13. Specify the name, `requestSignParts`.
14. Click **New** from Key information. Specify a name of `requestSignKeyInfo`.
15. Select **SymmetricBindingInitiatorSignatureToken**, and click **OK**.
16. Select **requestSignKeyinfo** in the Available list, and click **Add**. Click **OK**.
17. Repeat steps 8 to 16 for the links in the Response message signature and encryption protection table.
18. Click **Save**, to save the changes to the master configuration.

What to do next

Start the application.

Signed or Encrypted message part settings

Use this page to configure or create new signed or encrypted message parts. Message part bindings define how the part (which is defined in a policy set) is handled.

You can configure or create new signed or encrypted message parts when you are editing a default cell or server binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a default cell binding, complete the following actions:

1. Click **Services > Policy sets > Default policy set bindings**.
2. Click the **WS-Security** policy in the Policies table.
3. Click the **Authentication and protection** link in the Main message security policy bindings section.
4. Select a signature or an encrypted message part in the Request message signature and encryption protection section or the Response message signature and encryption protection section.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.

2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Authentication and protection** link in the Main message security policy bindings section.
7. Select a signature or an encrypted message part in the Request message signature and encryption protection section or the Response message signature and encryption protection section.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Name:

Specifies the name of the message part reference. The name field displays the name of the part reference you are editing, or you can enter a name if you are creating a message part reference.

Include time stamp:

This check box is available on this panel if you are configuring encryption protection and it specifies whether to include a time stamp. Select this check box to indicate that a time stamp is included or leave it unchecked to indicate that the time stamp is not included with the part reference.

For default bindings, to specify if a time stamp is included for signature protection, click the Signed part reference default link under the Additional bindings section.

For application specific bindings, to specify if a time stamp is included for signature protection, highlight an assigned signature message part reference and click **Edit**. The time stamp check box is located in the Reference section.

Include nonce:

This check box is available on this panel if you are configuring encryption protection and it specifies whether to include nonce. Select this check box to indicate that a nonce is to be used or leave it unchecked to indicate that nonce is not to be included with this part reference.

For default bindings, to specify if a nonce is included for signature protection, click the Signed part reference default link under the Additional bindings section.

For application specific bindings, to specify if a nonce is included for signature protection, highlight an assigned signature message part reference and click **Edit**. The nonce check box is located in the Reference section.

Usage of key information reference:

This field is available on this panel if you are configuring encryption protection and it specifies that the encryption key information is either data encryption key information or key encryption key information. Select **Data encryption** for symmetric algorithms and **Key encryption** for asymmetric algorithms.

Click one of the following radio buttons:

Data encryption
Indicates that the key information is used for data encryption.

Key encryption
Indicates that the key information is key encryption key information.

Key information (Request):

If you are configuring a request message signature or encryption protection, this field specifies the key information for a token request message part. This section provides interactive fields to assign the key information.

The **Available** field contains a listing of available key information entries for the message part. The **Assigned** field contains a listing of one or more of the key information entries that are assigned to the message part. Use the following actions to work with multiple request message part key information entries:

Button	Resulting action
Add	Add the selected key information entry in the **Available** list to the **Assigned** list.
New	Create a new key information entry.
Remove	Remove the selected key information entry from the **Assigned** list.

Key information (Response):

If you are configuring a response message signature or a response encryption protection, this field specifies the key information for a token response message part. This field provides a menu used to assign the key information. You can only assign one key information entry for response message parts. The **New** button enables you to add a new key information entry to the menu for selection.

Custom properties – Name:

Specifies the name of the custom property to be used.

Custom properties are not initially displayed in this column. The following actions are available:

Button	Resulting Action
New	Creates a new custom property entry. To add a custom property, enter the name and value.
Edit	Specifies that you can edit the selected custom property. Select this action to provide input fields and create the listing of cell values for editing. The Edit button is not available until at least one custom property has been added.
Delete	Removes the selected custom property.

Custom properties – Value:

Specifies the value of the custom property to be used. With the Value entry field, you can edit, enter or delete the value for a custom property.

Additional bindings – Signed part reference default:

If you are configuring signature protection, this section is displayed on this panel. It links to a panel where you can configure part reference properties such as including a time stamp or nonce and transform algorithms. Part reference properties include the transform algorithms used to protect the message part.

Configuring the callers for general and default bindings

The caller specifies the token or message part that is used for authentication.

Before you begin

Before you can complete this task, you must create a new policy set and attach it to a service, or copy and edit one of the sample system policy sets. For more information, read the topics Creating policy sets using the administrative console and Attaching a policy set to a service artifact.

About this task

The caller is used to indicate which of the tokens on the incoming message is the caller of the request. This information is used to create authentication credentials. You can use the administrative console to access, view and configure caller settings for tokens and message parts. The product provides support for multiple callers. The caller token used for authentication is the one with highest priority, based on decreasing order of preference. You can modify the order of the callers, as described in the topic Changing the order of the callers for a token or message part.

Procedure

1. Create a new policy set and attach it to a service, or copy and edit a sample system policy set. Add the WS-Security policy, as described in the topic Creating policy sets using the administrative console.
2. Edit the general or default bindings for the WS-Security policy.
   - To edit general provider bindings for WebSphere Application Server version 7.0 and later, click **Services** > **Policy Sets** > **General Provider policy set bindings**. A caller is specified for the provider bindings only, not for the client bindings.
   - To edit default bindings for WebSphere Application Server Version 6.x, click **Services** > **Policy Sets** > **Default policy set bindings**.
3. Navigate to the Callers panel by clicking on the **WS-Security** policy, then click the **Caller** link.
4. Click **New** to create a new caller.
5. Enter the Name and Caller identity local part information for the new caller. For more information, read about caller settings.
6. When you have finished entering the configuration information for the caller, click **Apply** to save the caller.
7. If this is the first caller created for the policy set, the caller is automatically assigned as the highest priority caller, with an order of 1 (one). If other callers are already defined, the new caller is added at the end of the ordered list and is automatically assigned the lowest priority. You can change the order of the callers using the Move up and Move down buttons.

Results

When assigning orders to callers for migrated bindings, the callers are initially displayed with no order attribute. You cannot save the bindings until you assign order attributes to all the callers. Use the **Move up** and **Move down** buttons to change the order of the callers until they are in the correct order.

Changing the order of the callers for a token or message part

Specifying a caller in default and general bindings indicates which token or tokens to use to create authentication credentials. When there are multiple tokens on an incoming message, the order of the callers determines which token is used for the credentials. You can rearrange the order of the callers using the administrative console.

Before you begin

Before you can complete this task, you must create a new policy set and attach it to a service, or copy and edit one of the sample system policy sets. For more information, read the topics Creating policy sets using the administrative console and Attaching a policy set to a service artifact. You can also create multiple new callers in the default provider bindings associated with the policy set, as described in the topic Configuring the callers for general and default bindings

Procedure

1. Edit the bindings for the policy by clicking **Services** > **Policy sets** > **General Provider policy set bindings**, then click on the name of the bindings. A caller is specified for the provider bindings only, not for the client bindings.
2. Navigate to the Callers panel by clicking on the policy name, then click the **Caller** link.
3. In the caller collection table, the callers are listed with the order of each caller displayed in the **Order** column. The order number indicates the order of preference in which the callers are used when multiple authentication tokens are received on an incoming message. Use the **Move up** and **Move down** buttons to change the order of the callers.
   a. To change the order of a caller to a higher priority, click the selection box next to the caller name, then click the **Move up** button. When a caller is moved up in priority, a caller that is above it will be moved down.
   b. To change the order of a caller to a lower priority, click the selection box next to the caller name, then click the **Move down** button.

Example

The order attribute is assigned only for callers on bindings in WebSphere Application Server Version 7.0 or later. Bindings created with earlier versions of WebSphere Application Server may have callers, but these callers do not have an order attribute. These callers can appear in the Callers collection table, but do not have an order number in the order column. If these bindings were migrated to Version 7.0 or later, then order attributes must be assigned before saving and using these bindings. You can use the **Move up** and **Move down** buttons to assign orders to the callers.

Configuring SCA web service binding to use SSL

Use this task to specify abstract intents in the Service Component Architecture (SCA) composite file to achieve a quality of service for secure connection using Secure Sockets Layer (SSL).

Before you begin

Determine whether your application requires the use of SSL.

About this task

Intents and policy sets can be used to configure web service bindings to achieve a secure connection.

Procedure

1. Configure administrative and application security for the server.

   To secure the service so that it only accepts secure requests, administrative and application security must be enabled for the server.
2. Configure the service binding to require an SSL connection by requiring the `confidentiality.transport` intent.

```
<service name="AccountService">
  <binding.ws
     requires="confidentiality.transport"
     ... />
</service>
```

An SSL connection is also required if an attached web service policy set includes the SSLTransport policy type. For information about attaching policy sets, refer to "Mapping abstract intents and managing policy sets."

Services in OASIS composites that are wired using an SCA target must require the `confidentiality.transport` intent.

3. 3. Configure the reference binding to require an SSL connection by requiring the `confidentiality.transport` intent.

```
<reference name="AccountService">
  <binding.ws
     requires="confidentiality.transport"
     ... />
</reference>
```

An SSL connection is also required if an attached web service policy set includes the SSLTransport policy type. For information about attaching policy sets, refer to "Mapping abstract intents and managing policy sets."

If you are not using an SCA target to wire the reference to a service, the `confidentiality.transport` intent simply enforces that the endpoint address specified in the composite file or WSDL file uses the https protocol. If you are using an SCA target, the `confidentiality.transport` intent causes SCA to use the SSL port of the target service. Services in OASIS composites that are wired using an SCA target must require the `confidentiality.transport` intent.

Results

When you finish this task, you have configured web service bindings to use SSL.

What to do next

You can proceed to configuring other application specific bindings for your policy sets.

Configuring web service binding for LTPA authentication

Use this task to configure a web service binding to perform authentication using Lightweight Third-Party Authentication (LTPA) tokens.

Before you begin

Before you begin this task, install Service Component Architecture (SCA) application.

About this task

Policy sets can be used to configure web service bindings to perform authentication using LTPA tokens.

Procedure

1. Configure the administrative and application security for the server.

   In order to secure the service so that it only accepts secure requests, and for the service to require authentication, administrative and application security must be enabled for the server. See Securing JAX-WS web services using message-level security.

2. Configure the service to require message layer authentication by attaching the LTPA WSSecurity default policy set.

   To attach the LTPA WSSecurity default policy set, perform the task, mapping abstract intent to policy sets and policy management.

In addition to attaching the policy set, you must configure the WS-Security policy to add a caller binding in order for the received subject to be propagated to the thread. To update the default binding to support the caller function, open the administrative console and navigate to **Services** > **Policy sets** > **General provider policy set bindings** > **Provider sample** > **WS-Security** > **Callers**. Create a new Caller with the following values:

Name: Specify any name for this configuration
Caller identity local part: LTPAv2
Caller identity namespace URI: http://www.ibm.com/websphere/appserver/tokentype

For additional information on LTPA WSSecurity default policy set review the topic, WSSecurity default policy sets. Read also the article about configuring the WS-Security policy.

The following code is an example of configuring the service to support LTPA authentication.

```
<service name="AccountService">
  <binding.ws
     qos:wsPolicySet="LTPA WSSecurity default" qos:wsServicePolicySetBinding="Provider sample"
     ... />
</service>
```

3. Configure the client by attaching the LTPA WSSecurity default policy set to a reference.

   An example of how to attach the LTPA WSSecurity default policy set to a reference is shown in the code block in this task step. Attaching the LTPA WSSecurity default policy set to a reference by default propagates any existing LTPA tokens on the thread with the request. It is also possible to configure the policy set to create a token for a specific user and send that token with all requests. Refer to the article, WSSecurity default policy sets for detail information.

```
<reference name="AccountService">
  <binding.ws
     qos:wsPolicySet="LTPA WSSecurity default"
     ... />
</reference>
```

Results

When you finish this task, you have configured web service bindings to do LTPA authentication.

What to do next

You can proceed to configuring other application specific bindings.

Policy set bindings settings for WS-Security

Use this page to view, define or configure general bindings and application specific properties for the WS-Security policy. You can configure the main policy or the secure conversation bootstrap policy by editing the general bindings.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

To use this administrative console page to view the general default bindings, click **Services** > **Policy sets** > **Default policy set bindings**. You can use this navigation path for viewing only. To edit or configure the general default bindings, complete the following steps:

1. Navigate to the general bindings collection panel by clicking**Services** > **Policy sets** > **General client policy set bindings** or **Services** > **Policy sets** > **General provider policy set bindings** path.
2. Click a general binding in the Name column.
3. Click the **WS-Security** policy in the Policies table.

If you choose to use a sample binding that is provided in the product, you must edit the sample user name and password that are provided for the Username token and LTPA token. The values provided are only examples; to use them successfully, you must modify the values for your own environment. You can change the user ID and password for authentication using a scripting command or by editing a copy of the general binding.

The following configuration links are provided for both the main security policy and for secure conversation bootstrap policy bindings.

Authentication and protection

Links to the collection of policy authentication and protection configuration settings. Click this link to access the collection of authentication and protection settings where you can configure authentication, signature, and encryption information that the policy requires.

Keys and certificates

Links to the collection of WS-Security policy keys and certificates.

Caller

Links to a panel to configure the caller settings. The caller specifies the token or message part that represents the identity to be set in the caller subject of the service.

The caller settings are available only for the service provider policy sets and bindings. The caller settings are not available for service client policy sets and bindings.

Message expiration

Links to a panel to define settings for message expiration. When you enable message expiration, the message expires after the specified interval.

Custom properties

Links to a panel where you can specify custom properties that apply to both inbound and outbound messages or specify properties that apply only to inbound or only to outbound messages.

Inbound and outbound custom properties

Use this page to set additional properties for inbound and outbound messages. You can specify custom properties that apply to both inbound and outbound messages or custom properties that apply to inbound messages only or outbound messages only.

The inbound and outbound custom properties are available in multiple locations throughout the administrative console. You can set these custom properties on the administrative console when you modify policy sets for Web Services Security. The following steps provide one method to set these custom properties:

1. Expand **Services > Policy sets**.
2. Click **Default policy set bindings >** `binding_name`.
3. Under the Policy heading, click **WS-Security**.
4. Under the Main message security policy bindings heading, click **Custom properties**.

Important: When you set custom properties in the Inbound Custom Properties or Outbound Custom Properties fields, those custom properties take precedence over the custom properties that are set in the Inbound and Outbound Custom Properties field.

You can also set these custom properties for custom bindings that are associated with applications.

Inbound and Outbound Custom Properties:

Add the name and value custom property pairs that affect both inbound and outbound messages.

Inbound Custom Properties:

Add the name and value custom property pairs that affect inbound messages only.

Outbound Custom Properties:

Add the name and value custom property pairs that affect outbound messages only.

Keys and certificates

Use this page to link to key and certificate binding configuration panels. This panel defines key and certificate bindings for JAX-WS web services only. These keys and certificates can be centrally managed by the product or in an external keystore.

You can define key and certificate bindings for message parts when you are editing a default cell or server binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a default cell binding, complete the following actions:

1. Click **Services** > **Policy sets** > **General provider policy set bindings** (for provider bindings), or **Services** > **Policy sets** > **General client policy set bindings** (for client bindings).
2. Click the **WS-Security** policy in the Policies table.
3. Click the **Keys and certificates** link in the Main message security policy bindings section.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Keys and certificates** link in the Main message security policy bindings section.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Key information – Name

Specifies the key information name. The key names listed in this field are links that are used to define key information attributes. Key information attributes define how cryptographic keys are generated or consumed.

Use the following buttons to work with this table:

Button	Resulting Action
New Inbound	Creates a new inbound key information name.
New Outbound	Creates a new outbound key information name.
Delete	Removes the selected key information name listing.

Key information – Type

Specifies the type of key information.

Key information – Direction

Specifies the whether the direction of the key is inbound or outbound. .

Certificate store – Name

Specifies the certificate store name. The certificate store names listed in this table are used to configure certificate stores.

Use the following actions to work with this table:

Button	Resulting Action
New Inbound	Creates a new inbound certificate store.
New Outbound	Creates a new outbound certificate store.
Delete	Removes the selected certificate store.

Certificate store – Direction

Specifies whether the direction of the certificate store is inbound or outbound.

Trust anchor – Name

Specifies the trust anchor name. The trust anchor names in this table are links that are used to configure trust anchor certificate stores.

You can use the following buttons to work with this table:

Button	Resulting Action
New	Creates a new trust anchor entry.
Delete	Removes the selected trust anchor.

Trust anchor – Keystore

Specifies the type of keystore for the trust anchor.

Key information settings

Use this page to configure the key information for the selected policy set binding. Key information attributes define how cryptographic keys are generated or consumed.

You can configure the key information for the selected policy set binding when you are editing a default cell or server binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a default cell binding, complete the following actions:

1. Click **Services** > **Policy sets** > **General provider policy set bindings** or **General client policy set bindings**.
2. Click on a binding name in the **Name** column.
3. Click the **WS-Security** policy in the Policies table.
4. Click the **Keys and certificates** link in the Main message security policy bindings section.
5. Click a key in the **Name** column of the Key information table.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Keys and certificates** link in the Main message security policy bindings section.
7. Click a key in the **Name** column of the Key information table.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Name:

Specifies the unique name for the key information configuration.

The key information name field displays the unique name of the key that is being configured if you are editing a key. If you are creating one, enter a unique name.

Type:

Lists the type of key reference.

This field appears only if you selected an encryption or signing key for the generator binding, such as gen_signkeyinfo, gen_signsctkeyinfo, gen_encsctkeyinfo, or gen_enckeyinfo.

You can select one of the following key types from this list:

Key identifier
The associated attribute in the binding file is KEYID.

Security token reference
The associated attribute in the binding file is STRREF.

Embedded token
The associated attribute in the binding file is EMB.

X.509 issuer name and issuer serial
The associated attribute in the binding file is X509ISSUER.

Thumbprint
The associated attribute in the binding file is THUMBPRINT.

The Thumbprint key information type requires a keystore with the public and private key pair instead of a shared key.

Information	Value
Data type:	Selection list

Token generator or consumer name:

Specifies the name of the token generator or consumer. Specifies a unique name for the token configuration.

The token generator or consumer name field displays the name of the pre-configured tokens that can be used in the key information configuration if you are editing a key or creating a new key.

You can select a token generator or consumer name from this list. The list of names changes, depending on whether the key information selected is for inbound (consumer) keys or outbound (generator) keys. For keys with outbound direction, the list of defined token generators is displayed. For keys with inbound direction, the list of defined token consumers is displayed.

Information	Value
Data type:	String

Direction:

Specifies whether the direction of the key is inbound or outbound.

The direction of generator tokens are outbound whereas the direction for consumer tokens and decryption keys are inbound.

Information	Value
Data type:	String
Default values:	Inbound (for consumer bindings) or Outbound (for generator bindings)

Requires derived keys:

Specifies whether the key information requires derived keys.

Explicit derived keys
: Requires that derived keys be explicitly specified with a WS-SecureConversation <DerivedKeyToken> element.

Implicit derived keys
: Requires that derived keys be implicitly specified with a WS-SecureConversation Nonce attribute on the WS-Security <SecurityTokenReference> element.

Override Defaults:

Specifying derived key values overrides the derived key information that the runtime generates by default.

Note: It is recommended that you do not override the following optional attributes. Web Services Security automatically provides default values for each attribute. Overriding the default values might be required if the service is running cross-vendors. The vendors can use different attribute values for derived key generation.

Key length
: Specifies the derived key length. If an override value is not specified, the default value is provided based on the algorithm suite policy assertion. It is recommended that you leave this field empty so the default value can be used. Valid values for the key length range between 16 and 32.

Nonce length
: Specifies the nonce length. A nonce is generated for each request, and included for derived key generation. This value is optional, and if an override value is not specified, a default value is used to generate the nonce. A valid value for the nonce length is any integer between 16 and 128.

Client label
: Specifies the client label. The label is used in the P_SHA-1 function to generate the derived key. If unspecified, the default value used is `WS-SecureConversation`.

Service label
: Specifies the service label. The label is used in the P_SHA-1 function to generate the derived key. If unspecified, the default value used is `WS-SecureConversation`.

Custom properties:

Specifies additional configuration settings that token types might require.

Custom properties are arbitrary name-value pairs of data.

This table lists custom properties. Use custom properties to set internal system configuration properties. You are not required to define a custom property when you define a custom token.

Select:

Specifies custom properties that you can add, edit, or delete from policy set bindings.

Click **New** to add and define a new custom property.

For existing custom properties, select the check box for the name of the custom property, and click one of the following actions:

Action	Description
New	Creates a new custom property entry. To add a custom property, enter the name and value.
Edit	Specifies that you can edit the selected custom property. Click this option to provide input fields and create the list of cell values to edit. At least one custom property must exist before the Edit option is displayed.
Delete	Removes the selected custom property.

Information	Value
Data type:	Check box (unchecked)

Name:

Specifies the name of the custom property that you can use with default policy set bindings.

Custom properties are arbitrary name-value pairs of data. Custom properties are not initially displayed in this column until at least one custom property has been added.

Information	Value
Data type:	String

Value:

Specifies the custom property value.

This column displays the value for the custom property (for example, `true`). The value can be a string or the value can be a true or false Boolean value.

Information	Value
Data type:	String or Boolean

Certificate store settings

Use this page to specify the location where certificates are stored. You can reference certificate revocation for service generators or consumers.

You can specify the location where certificates are stored when you are editing a default cell or server binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a default cell binding, complete the following actions:

1. Click **Services > Policy sets > Default policy set bindings**.
2. Click the **WS-Security** policy in the Policies table.
3. Click the **Keys and certificates** link in the Main message security policy bindings section.
4. Click the ***certificate_store_name*** link in the Certificate store section.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Keys and certificates** link in the Main message security policy bindings section.
7. Click the ***certificate_store_name*** link in the Certificate store section.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Name:

Specifies the name of the certificate store. The name field displays the name of the certificate store if you are editing a certificate store, or enter a name if you are creating a new certificate store.

Revoked certificates – Full path:

Specifies, in the Revoked certificates table, the paths for any certificates that are revoked. The Full Path column of this table lists any certificates that have been revoked.

You can add, edit, or remove these entries with the following buttons:

Button	Resulting Action
New	Creates a revoked generator or consumer certificate store.
Delete	Removes the selected revoked generator or consumer certificate store.
Edit	Allows you to edit the applied entries selected in the checkbox. This button is only displayed if revoked certificates exist in your configuration.

Intermediate X.509 certificates – Full Path:

Specifies, for the consumer certificate store only, the paths for any intermediate X.509 certificates. The Full Path column of this table lists any intermediate X.509 certificate stores. This table is only displayed for the consumer version of this panel. It is not valid for generator certificate stores.

Note: If the certificate store is outbound, the Intermediate X.509 certificates field is not displayed.

You can create, edit, or remove intermediate X.509 certificates with the following buttons:

Button	Resulting Action
New	Creates an intermediate X.509 consumer certificate store.
Delete	Removes an intermediate X.509 consumer certificate store.
Edit	Allows you to edit the applied entries selected in the checkbox.

Trust anchor settings

Use this page to specify the trust anchor configuration. These trust anchor certificates are used to validate the X.509 certificate that is embedded in the SOAP message.

Use this information to configure a trust anchor. Trust anchors point to keystores that contain trusted root or self-signed certificates. This information enables you to specify a name for the trust anchor and the information that is needed to access a keystore. The application binding uses this name to reference a predefined trust anchor definition in the binding file (or the default).

You can configure a trust anchor when you are editing a default cell or server binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a default cell binding, complete the following actions:

1. Click **Services** > **Policy sets** > **Default policy set bindings**.
2. Click the **WS-Security** policy in the Policies table.
3. Click the **Keys and certificates** link in the Main message security policy bindings section.
4. Click a name link in the **Name** column of the Trust anchor table.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Keys and certificates** link in the Main message security policy bindings section.
7. Click a name link in the **Name** column of the Trust anchor table.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Name:

Specifies the unique name that is used by the application binding to reference a predefined trust anchor definition in the default binding.

A trust anchor specifies the keystore that contains trusted root certificates. This field displays the name for the trust anchor that is being edited. If you are creating a new trust anchor configuration, enter a unique name.

Keystore files contain public and private keys, root certificate authority (CA) certificates, the intermediate CA certificate, and so on. Keys that are retrieved from the keystore files are used to sign and validate or encrypt and decrypt messages or message parts.

Information	Value
Data type:	String

Centrally managed keystore:

Specifies to use a centrally managed keystore. After selecting the **Centrally managed keystore** option, choose one of the centrally managed keystore names from the list. Centrally managed keystores can be managed in the administrative console by clicking these links: **Security** > **SSL certificate and key management > Key stores and certificates**.

Click the radio button to enable the **Name** field. Select a keystore from the list.

Information	Value
Data type:	Radio button
Default value:	Unselected

External keystore:

Specifies a keystore using a keystore path, keystore type and keystore password. The keystore file format is determined by the keystore type. The default trust anchor in the default binding uses an external keystore.

Select the radio button to enable an external keystore.

Information	Value
Data type:	Radio button
Default value:	Selected

Full path

Specifies the full path to the location of the keystore.

If the keystore is file-based, the location can reference any path in the file system of the node where the trust anchor keystore is located. The trust anchor defined in the default bindings is:

```
${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
```

Attention: Do not use the sample keystore files in a production environment. These samples are provided for testing purposes only.

Information	Value
Data type:	String

Type Specifies the type of keystore when the external keystore is enabled.

The type specifies the implementation for keystore management. Click a keystore type from the list provided. The selection list is returned by java.security.Security.getAlgorithms("KeyStore").

The IBM Java Cryptography Extension (IBMJCE) supports the following file-based keystore types: JKS, JCEKS,PKCS12, and CMSKS.

- Use the JKS option if you are not using Java Cryptography Extensions (JCE).
- Use the JCEKS option if you are using Java Cryptography Extensions.
- Use the PKCS12 option if your keystore uses the PKCS#12 file format.
  - A key.p12 file or a trust.p12 file are examples of PKCS12 type keystores.
- Use the CMSKS option if your keystore uses the Certificate Management Services (CMS) format.

Password

Specifies the password that is needed to access the keystore file.

Use the password to protect the keystore. The password is used to access the named keystore and the password is also the default password that is used to store keys within the keystore.

The default trust anchor in default binding uses an external keystore. The password for the external keystore is: `server`. It is recommended that you change the default password as soon as possible.

Information	Value
Data type:	String
Default value:	WebAS or cell name

Confirm password

Confirms the password entered in the Password field.

Enter the password that is used to open the keystore file or device again. By entering the same password that was entered in the Password field again, you confirm the password.

Information	Value
Data type:	String

WS-Security authentication and protection

Use the links on this page to configure authentication, protection, signature, and encryption information that the policy requires.

You can configure authentication, protection, signature, and encryption information for tokens and message parts when you are editing a default cell or server binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a general provider policy set binding or general client policy set binding, complete the following actions:

1. Click **Services** > **Policy sets** > **General provider policy set bindings** or **Services** > **Policy sets** > **General client policy set bindings**.
2. Click **Provider sample** or **Client sample**.
3. Click **WS-Security** in the Policies table.
4. Click the **Authentication and protection** link in the Main Message Security Policy Bindings section.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Select an application that contains JAX-WS web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** link in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned an application specific binding.
5. Click **WS-Security** in the Policies table.
6. Click the **Authentication and protection** link in the Main message security policy bindings section.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

WS-Security authentication and protection for general bindings

Use the links on this page to configure authentication, protection, signature, and encryption information that the policy requires when using general bindings.

You can configure authentication, protection, signature, and encryption information for tokens and message parts when you are editing a general binding.

To view this administrative console page when you are editing a general binding at the cell level, complete the following actions:

1. Click **Services > Policy sets > General provider policy set bindings** or **General client policy set bindings**.
2. Click on the name of the bindings you want to edit.
3. Click **WS-Security** policy in the Policies table.
4. Click the **Authentication and protection** link in the Main message security policy bindings section.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Disable implicit protection for signature confirmation:

Specifies whether implicit protection of the SignatureConfirmation element is enabled or disabled.

The explicitlyProtectSignatureConfirmation attribute in the Web Services Security binding is provided to disable implicit signature and encryption of the SignatureConfirmation element on the response message. If this checkbox is selected, the attribute is added and implicit protection is disabled. This provides interoperability with earlier versions of WebSphere Application Server.

Information	Value
Default:	Not selected (implicit protection is enabled)

Protection tokens – Protection token name:

Specifies a list of protection tokens that can be configured in the Protection tokens table.

The following actions are available for general bindings:

Button	Resulting Action
New Token	Creates a new protection token type.
Delete	Removes the selected protection token type.

Protection tokens – Usage:

Specifies the policy assertion usage names that you can customize in the Protection tokens table.

For the usage field, the following options are available for the general bindings:

- Asymmetric encryption generator
- Asymmetric encryption consumer
- Asymmetric signature generator
- Asymmetric signature consumer
- Symmetric generator
- Symmetric consumer
- Custom generator
- Custom consumer

Authentication tokens – Authentication token name:

Specifies a list of authentication tokens that you can customize in the Authentication tokens table when using general bindings.

If you are working with a Username token or LTPA token that is using general bindings, the user names and passwords might have been provided as examples. When you click a Username token or LTPA token link, you need to update the values for these token types using the Callback handler link found on the Authentication token settings page.

The following actions are available for general bindings:

Button	Resulting Action
New Token	Creates a new authentication token type.
Delete	Removes the selected authentication token type.

Authentication tokens – Usage:

Specifies the usage names for the Authentication tokens table for general bindings.

The following options are available for general bindings:
- Inbound
- Outbound

Request message signature and encryption protection – Name:

Specifies a unique name to identify the request message part from the Request message signature and encryption protection table that is protected.

The following actions are available for general bindings. The Move up and Move down actions are available only when using service client policy sets and bindings.

Button	Resulting Action
New Signature	Creates a new signature.
New Encryption	Creates a new encryption protection.
Delete	Removes the selected request message part.
Move up	Moves the selected request message part up in the order.
Move down	Moves the selected request message part down in the order.

Request message signature and encryption protection – Protection:

Specifies the type of protection from the Request message signature and encryption protection table. This field displays the type of protection enabled for the general binding.

Response message signature and encryption protection – Name:

Specifies a unique name to identify the response message part from the Response message signature and encryption protection table that is protected.

The following actions are available for general bindings. The Move up and Move down actions are available only when using service provider policy sets and bindings.

Button	Resulting Action
New Signature	Creates a new response message signature.

Button	Resulting Action
New Encryption	Creates a new encryption.
Delete	Removes the selected response message part.
Move up	Moves the selected response message part up in the order.
Move down	Moves the selected response message part down in the order.

Response message signature and encryption protection – Protection:

Specifies the type of protection enabled from the Response message signature and encryption protection table. This field displays the type of protection enabled for the response message part.

Response message signature and encryption protection – Order:

Specifies the order in which the signatures and encryptions occur. Use the **Move up** and **Move down** actions to order the list of protection types in this table.

WS-Security authentication and protection for application specific bindings

Use the links on this page to configure authentication, signature, and encryption information that the policy requires when using application specific bindings.

You can configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Authentication and protection** link in the Main message security policy bindings section.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Disable implicit protection for Signature Confirmation:

Specifies whether implicit protection of the SignatureConfirmation element is enabled or disabled.

The explicitlyProtectSignatureConfirmation attribute in the Web Services Security binding is provided to disable implicit signature and encryption of the SignatureConfirmation element on the response message. If this checkbox is selected, the attribute is added and implicit protection is disabled. This provides interoperability with earlier versions of WebSphere Application Server.

Information	Value
Default:	Not selected (implicit protection is enabled)

Protection tokens – Protection token name:

Specifies a list of protection tokens that can be configured in the Protection tokens table for application specific bindings.

The following actions are available for application specific bindings:

Button	Resulting Action
Unconfigure	Removes the selected protection token from the binding.

Protection tokens – Protection token type:

Specifies the protection token type for application specific bindings.

Protection tokens – Usage:

Specifies the policy assertion usage names that you can customize in the Protection tokens table.

For the usage field, the following options are available for the application specific bindings:
- Asymmetric encryption generator
- Asymmetric encryption consumer
- Asymmetric signature generator
- Asymmetric signature consumer
- Symmetric encryption generator
- Symmetric encryption consumer
- Symmetric signature generator
- Symmetric signature consumer

Protection tokens – Status:

Specifies the status of the protection token when using application specific bindings. The valid values are configured, not configured, or incompatible.

Authentication tokens – Security token reference:

Specifies a list of authentication tokens that you can customize in the Authentication tokens table when using application specific bindings.

The following actions are available for application specific bindings:

Button	Resulting Action
Unconfigure	Removes the selected authentication token from the binding.

Authentication tokens – Authentication token type:

Specifies the authentication token type for the security token reference when using application specific bindings.

Authentication tokens – Usage:

Specifies the usage names from the Authentication tokens table for application specific bindings.

The following options are available for application specific bindings:
- Inbound request

- Outbound request
- Inbound response
- Outbound response

Authentication tokens – Status:

Specifies the status of the authentication token form the Authentication tokens table for application specific bindings. The valid values are configured, not configured, or incompatible.

Request message signature and encryption protection – Request message part reference:

Specifies the name of the request message part in the policy from the Request message signature and encryption protection table that is protected.

The following actions are available for application specific bindings. The **Move up** and **Move down** actions are available only when using Service client policy sets and bindings.

Button	Resulting Action
Unconfigure	Removes the selected request message part from the binding.
Move up	Moves the selected request message part up in the order.
Move down	Moves the selected request message part down in the order.

Request message signature and encryption protection – Protection:

Specifies the type of protection from the Request message signature and encryption protection table. This field displays the type of protection enabled for the application specific binding.

Request message signature and encryption protection – Order:

Specifies the order in which signatures and encryptions occur when using service client policy sets and bindings. Use the **Move up** and **Move down** actions to order the list of protection types in this table.

Request message signature and encryption protection – Status:

Specifies the status of the request message signature and encryption protection token when using application specific bindings. The valid values are configured, not configured, or incompatible.

Response message signature and encryption protection – Response message part reference:

Specifies the name of the response message part in the policy from the Response message signature and encryption protection table that is protected.

The following actions are available for application specific bindings. The **Move up** and **Move down** actions are available only when using Service provider policy sets and bindings.

Button	Resulting Action
Unconfigure	Removes the selected response message part from the binding.
Move up	Moves the selected response message part up in the order.
Move down	Moves the selected response message part down in the order.

Response message signature and encryption protection – Protection:

Specifies the type of protection enabled from the Response message signature and encryption protection table. This field displays the type of protection enabled for the response message part.

Response message signature and encryption protection – Order:

Specifies the order in which signatures and encryptions occur when using service provider policy sets and bindings. Use the **Move up** and **Move down** actions to order the list of protection types in this table.

Response message signature and encryption protection – Status:

Specifies the status of the response message signature and encryption protection token when using application specific bindings. The valid values are configured, not configured, or incompatible.

Protection token settings (generator or consumer)

Use this page to configure protection tokens. Protection tokens sign messages to protect integrity or encrypt messages to provide confidentiality.

You can add protection token settings for message parts when you are editing general provider or client policy set bindings. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a general provider binding, complete the following actions:

1. Click **Services** > **Policy sets** > **General provider policy set bindings**.
2. Click on the name of the binding you want to edit.
3. Click the **WS-Security** policy in the Policies table.
4. Click the **Authentication and protection** link in the security policy bindings section.
5. Click **New token** to create a new token generator or consumer, or click an existing consumer or generator token link from the Protection Tokens table.

To view this administrative console page when you are editing a general client binding, complete the following actions:

1. Click **Services** > **Policy sets** > **General client policy set bindings**.
2. Click on the name of the binding you want to edit.
3. Click the **WS-Security** policy in the Policies table.
4. Click the **Authentication and protection** link in the Main message security policy bindings section.
5. Click **New token** to create a new token generator or consumer or click an existing consumer or generator token link from the Protection Tokens table.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Websphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned a binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Authentication and protection** link in the security policy bindings section.
7. Click a consumer or generator token link from the Protection Tokens table.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Name:

Specifies the token generator or consumer name. Enter a name in this field when you create a new token.

Token type:

Specifies the type of token. When using bindings, the token type is determined from the policy and cannot be edited.

Valid values are:
- LPTA Token V2.0
- Secure Conversation Token V1.3
- Secure Conversation Token V200502
- X509V3 Token V1.1
- X509V3 Token V1.0
- X509PKCS7 Token V1.1
- X509PKCS7 Token V1.0
- X509PkiPathV1 Token V1.1
- X509PkiPathV1 Token V1.0
- X509V1 Token V1.1
- Custom Token

The Secure Conversation Token v200502 token type for the WS-Security policy represents the requirement for a Security Context Token as defined in the February 2005 level of the WS-SecureConversation specification.

Enforce token version:

When LTPA Token v2.0 is selected as the token type, both LTPA version 1 and LTPA version 2 tokens can be consumed. Select this checkbox to restrict token consumption to the LTPA Token v2.0 token type.

Local name:

Specifies the local name of the custom token generator or consumer. The **Local name** field is populated based on the token type displayed. Use this field to edit custom token types only.

If the custom token type is used to generate a Kerberos token as defined in the OASIS Web Services Security Specification for Kerberos Token Profile V1.1, use one of the values listed below for the local name. The value you choose depends on the specification level of the Kerberos token generated by the Key Distribution Center (KDC). The following table lists the values and the specification level associated with each value. For purposes of interoperability, the Basic Security Profile V1.1 standard requires the use of the local name http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ.

Local Name Value for Kerberos Token	Associated Specification Level
http://docs.oasis-open.org/wss/oasiswss-kerberos-token-profile-1.1#Kerb erosv5_AP_REQ	Kerberos v5 AP-REQ as defined in the Kerberos specification. Use this value when the Kerberos ticket is an AP Request.

Local Name Value for Kerberos Token	Associated Specification Level
http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ	GSS-API Kerberos V5 mechanism token containing a KRB_AP_REQ message as defined in RFC-1964 [1964], Sec. 1.1 and its successor RFC-4121, Sec. 4.1. Use this value when the Kerberos ticket is an AP Request (ST + Authenticator).
http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510	Kerberos v5 AP-REQ as defined in RFC1510. Use this value when the Kerberos ticket is an AP Request per RFC1510.
http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510	GSS-API Kerberos V5 mechanism token containing a KRB_AP_REQ message as defined in RFC-1964, Sec. 1.1 and its successor RFC-4121, Sec. 4.1. Use this value when the Kerberos ticket is an AP Request (ST + Authenticator) per RFC1510.
http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120	Kerberos v5 AP-REQ as defined in RFC4120. Use this value when the Kerberos ticket is an AP Request per RFC4120.
http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120	GSS-API Kerberos V5 mechanism token containing an KRB_AP_REQ message as defined in RFC-1964, Sec. 1.1 and its successor, RFC-4121, Sec. 4.1. Use this value when the Kerberos ticket is an AP Request (ST + Authenticator) per RFC4120.

URI:

Specifies the uniform resource identifier (URI) of the custom token generator or consumer. The **URI** field is populated based on the token type displayed. Use this field to edit custom token types only.

Leave this field blank if the custom token type is used to generate a Kerberos token as defined in the OASIS Web Services Security Specification for Kerberos Token Profile V1.1.

JAAS login:

Specifies the Java Authentication and Authorization Service (JAAS) application login information. Click **New** to add a new JAAS application login or JAAS system login entry.

If the server is in a security domain that includes specific system or application logins, these logins are listed in the JAAS login menu, in addition to the global logins.

New Application Login:

Click to go to the effective JAAS login collection for the current security domain.

Custom properties – Name:

Specifies the name of the custom property. Custom properties are not initially displayed in this column until they are added.

Select one of the following actions for custom properties:

Button	Resulting Action
New	Creates a new custom property entry. To add a custom property, enter the name and value.

Button	Resulting Action
Edit	Specifies that you can edit the selected custom property. Select this action to provide input fields and create the listing of cell values for editing. The **Edit** button is not available until at least one custom property has been added.
Delete	Removes the selected property.

If the custom token type is used to generate a Kerberos token, specify the following custom properties:

Custom property name	Value
`Specify the name of the target service. com.ibm.wsspi.wssecurity.krbtoken.targetServiceName`	Specifies the name of the target service. This property is required.
`com.ibm.wsspi.wssecurity.krbtoken.targetServiceHost`	Specifies the host name that is associated with the target service in the following format: `myhost.mycompany.com`. This property is required.
`com.ibm.wsspi.wssecurity.krbtoken.targetServiceRealm`	Specifies the name of the realm that is associated with the target service. This property is optional for a single Kerberos realm. If the targetServic In a cross or trusted realm environment, you must provide a value for t

For the token generator, the combination of the target service name and target hostname forms the Service Principal Name (SPN), which represents the target Kerberos service principal name. The Kerberos client requests the initial Kerberos AP_REQ token for the SPN.

If an application generates or consumes a Kerberos V5 AP_REQ token for each web services request message, set the `com.ibm.wsspi.wssecurity.kerberos.attach.apreq` custom property to `true` in the token generator and the token consumer bindings for the application. For more information, see the Web Services Security troubleshooting tips topic.

Custom properties – Value:

Specifies the value of the custom property. Use the **Value** field to enter, edit, or delete the value for a custom property.

Callback handler:

After all other configurations on the protection token page are applied or saved, this section is displayed and links to the configuration settings for the callback handler. Click this link to specify callback handler settings that determine how security tokens are acquired from message headers.

Tolerate secure conversation token V200502:

The secure conversation token V200502 token type for the WS-Security policy represents the requirement for a secure conversation token as defined the in the February 2005 level of the WS-SecureConversation specification. This option specifies whether the provider handles both secure conversation token V1.3 and secure conversation token V200502. By default, the provider handles both versions. You can change this behavior by clicking to remove the check box selection so that the provider handles only the V1.3 token.

Note: This checkbox is displayed only in the service provider token consumer panel.

Information	Value
Data type	Check box
Range	Selected or cleared

Information	Value
Default value	Selected

Authentication generator or consumer token settings

Authentication tokens are used to prove or assert an identity. Use the administrative console to add authentication token settings for message parts when you are editing a general binding.

To configure authentication tokens, complete the following steps:

1. To view and select the general bindings that are set as the global security default policy set bindings, click **Services > Policy sets > Default policy set bindings**. The specified bindings are used unless overridden at the attachment point, at the server, or at a security domain.
2. To access and configure the general bindings and to add authentication token settings for message parts, click **Services > Policy sets > General provider policy set bindings**.
3. Click the **WS-Security** policy in the Policies table.
4. Click the **Authentication and protection** link in the Main message security policy bindings section.
5. Click **New token** to create a new token generator or consumer, or click an existing consumer or generator token link from the Authentication Tokens table.

To view and configure application-specific bindings for tokens and message parts that are required by a policy set, complete the following steps:

1. Click **Applications > Application Types > WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** link in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned an application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Authentication and protection** link in the Main message security policy bindings section.
7. Click a consumer or generator token link from the Protection Tokens table.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Name:

Specifies the name of the token being configured. When using application specific bindings, this field is not displayed.

Token type:

Specifies the type of token being configured.

When you are using application specific bindings, the token type is obtained from the policy file and it is read-only. When you are using general bindings, select a token type from the list. The following token types are available:

- X509V3 Token V1.1
- X509V3 Token V1.0
- Username Token V1.1
- Username Token V1.0
- X509PKCS7 Token V1.1
- X509PKCS7 Token V1.0

- X509PkiPathV1 Token V1.1
- X509PkiPathV1 Token V1.0
- LTPA Propagation Token
- X509V1 Token V1.1
- LTPA Token
- LTPA Token V2.0
- Custom Token

Note: The LTPA Token V2.0 token type is available only for bindings using the namespace as supported in IBM WebSphere Application Server, Version 7.0 or later. When you select LTPA Token V2.0 as the token type for the token consumer, both LTPA tokens and LTPA V2.0 tokens can be consumed. To restrict the token consumer to LTPA V2.0 tokens only, select the **Enforce token version** check box.

If you select LTPA Token as the token type for the token generator, single sign-on interoperability mode must be enabled. This is a setting in global security from Web and SIP security. If the interoperability flag is not set to enabled (true), an error occurs when the application that is attached to these bindings is started. If you want to use the LTPA token without checking the state of the interoperability flag, you can set the custom property, com.ibm.wsspi.wssecurity.tokenGenerator.ltpav1.pre.v7, on the token generator. Set the property using the administrative console, as described in the topic Enabling or disabling single sign-on interoperability mode for the LTPA token. The property can not be set using the Web Services Security API.

Local name:

Specifies the local name for the authentication token generator or consumer. The **Local name** field is populated based on the token type displayed. Use this field to edit custom token types only.

URI:

Specifies the uniform resource identifier (URI) of the authentication token generator or consumer. The **URI** field is populated based on the token type displayed. Use this field to edit custom token types only.

Leave this field blank if the custom token type is used to generate a Kerberos token as defined in the OASIS Web Services Security Specification for Kerberos Token Profile v1.1.

Security token reference:

Specifies the security token reference. The security token reference field is displayed only for authentication tokens in application-specific bindings. This field is not available for default bindings.

JAAS login:

Specifies a list of application and system Java Authentication and Authorization Service (JAAS) logins that are effective for the domain to which the binding is scoped.

If an application is scoped to the global security or if it is scoped to a domain that does not customize its own JAAS logins, then the list of global logins are displayed in the menu list. Click **New Application Login** to access the global JAAS application login collection. The JAAS login menu list and **New Application Login** button behavior depend on whether the binding is being created in association with an attachment. Use caution when changing security domains, since a previously-referenced security configuration, such as JAAS logins, might not be accessible in a different security domain.

Custom properties – Name:

Specifies the name used for the custom property.

Custom properties are not initially displayed in this column. Click one of the following buttons to enable the actions described:

Button	Resulting Action
New	Creates a new custom property entry. To add a custom property, enter the name and value.
Edit	Enables the selected custom property to be edited. Clicking this button provides input fields and creates the listing of cell values to be edited. The Edit button is not available until at least one custom property has been added.
Delete	Removes the selected custom property.

Custom properties – Value:

Specifies the value of the custom property to be used. Use the **Value** field to enter, edit, or delete the value for a custom property.

If the custom token type is used to generate a Kerberos token, specify the following custom properties:

Custom property name	Value
`com.ibm.wsspi.wssecurity.krbtoken.targetServiceName`	Specifies the name of the target service. This property is required.
`com.ibm.wsspi.wssecurity.krbtoken.targetServiceHost`	Specifies the host name that is associated with the target service in the following format: `myhost.mycompany.com`. This property is required.
`com.ibm.wsspi.wssecurity.krbtoken.targetServiceRealm`	Specifies the name of the realm that is associated with the target service. This property is optional for a single Kerberos realm. If the targetSer In a cross or trusted realm environment, you must provide a value fo
`com.ibm.wsspi.wssecurity.krbtoken.clientRealm`	Specifies the name of the Kerberos realm associated with the client. This property is optional for a single Kerberos realm environment. When implementing Web Services Security in a cross or trusted Kerb
`com.ibm.wsspi.wssecurity.krbtoken.loginPrompt`	Enables the Kerberos login when the value is `True`. The default value is `False`. This property is required.

For the token generator, the combination of the target service name and target hostname forms a Service Principal Name (SPN) which represents the target Kerberos service principal name. The Kerberos client requests the initial Kerberos AP_REQ token for the SPN.

If an application generates or consumes a Kerberos V5 AP_REQ token for each web services request message, set the `com.ibm.wsspi.wssecurity.kerberos.attach.apreq` custom property to `true` in the token generator and the token consumer bindings for the application. For more information, see the Web Services Security troubleshooting tips topic.

Callback handler:

Links to the Callback handler page where you can configure callback handlers. Callback handler settings determine how security tokens are acquired from messages headers.

If you are working with a Username token or LTPA token that is using default bindings, the user names and passwords might have been provided as examples. You need to update the values for these token types.

Callback handler settings for JAX-WS

Use this page to configure callback handler settings for JAX-WS, which determine how security tokens are acquired from messages headers.

You can configure callback handler settings when you are editing a general cell-level or server-level binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

gotcha: Before you specify values for the **Keystore** and **Key** properties on this page, you must understand that the keystore/alias information that you provide for the generator, and the keystore/alias information that you provide for the consumer are used for different purposes. The main difference applies to the alias for an X.509 callback handler:

Generator

When used in association with an encryption generator, the alias supplied for the generator is used to retrieve the public key to encrypt the message. A password is not required. The alias that is entered on a callback handler associated with an encryption generator must be accessible without a password. This means that the alias must not have private key information associated with it in the keystore. When used in association with a signature generator, the alias supplied for the generator is used retrieve the private key to sign the message. A password is required.

Consumer

When used in association with a encryption consumer, the alias supplied for the consumer is used retrieve the private key to decrypt the message. A password is required.

When an X.509 certificate is sent in the SOAP security header as a BinarySecurityToken, if there is a keystore/alias configured on the X.509 token consumer associated with a signature consumer, the certificate that is configured on the consumer will be compared against the one that is passed in the message. If they do not match, the message will be rejected. This behavior is different than JAX-RPC. The certificate associated with the alias configured on the X.509 token consumer is not used to evaluate trust on the inbound certificate. Only the trust store and cert stores are used for that purpose.

If you want the certificate configured on the X.509 token consumer associated with a signature consumer to be availble for KeyInfo resolution, but not reject X.509 certificates that are passed in the message that do not match, you can add the following custom property to the X.509 token consumer callback handler:
com.ibm.wsspi.wssecurity.consumer.callbackHandlerKeystoreLimitsAccess=false

See the topic *Key information settings* for more information about the key identifier, X.509 issuer/serial, and thumbprint.

To view this administrative console page when you are editing a general cell-level binding, complete the following actions:

1. Click **Services** > **Policy sets** > **Default policy set bindings**. The bindings panel indicates which binding is set as the default binding, for example, the Provider sample binding.
2. To edit this default binding, click **Services** > **Policy sets** > **General provider policy set bindings**.
3. Click the name of the default binding as determined in the first step. For example, **Provider sample**.
4. Click the **WS-Security** policy in the Policies table.
5. Click the **Authentication and protection** link in the Main message security policy bindings section.
6. Click the ***name_of_token*** link in the Protection tokens section or the Authentication tokens section.
7. Click the **Callback handler** link.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned an application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Authentication and protection** link in the Main message security policy bindings section.
7. Click the ***name_of_token*** link in the Protection tokens section or the Authentication tokens section.
8. Click the **Callback handler** link.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

The Callback Handler displays fields differently for different tokens being configured. Depending on whether you are configuring generator or consumer tokens for protection or you are configuring inbound or outbound tokens for authentication, the sections and fields on this panel display some or all of the fields explained in this topic, as noted in the description of each field.

Class name:

The fields in the Class name section are available for all types of token configuration.

Select the class name to use for the callback handler. Select the **Use built-in default** option for normal operation. Use the **Use custom** option only if you are using a custom token type.

For the Kerberos custom token type, use the class name, `com.ibm.websphere.wssecurity.callbackhandler.KRBTokenGenerateCallbackHandler`, for token generator configuration. Use `com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler` for token consumer configuration.

Use built-in default:

Specifies that the default value is used for the class name. Use the default value (shown in the field) for the class name when you select this radio button. This name is based on the token type and whether the callback handler is for a token generator or a token consumer. This option is mutually exclusive to the **Use custom** option.

Use custom:

Specifies that a custom value is used for the class name. Select this radio button and enter the name in the field to use a custom class name.

No default value is available for this entry field. Use the information in the following table to determine this value:

Table 16. Custom class names for the callback handler and associated token types. The callback handler determines how security tokens are acquired from message headers.

Token Type	Consumer or Generator	Callback Handler Class Name
UsernameToken	consumer	com.ibm.websphere.wssecurity.callbackhandler.UNTConsumeCallbackHandler
UsernameToken	generator	com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler

Table 16. Custom class names for the callback handler and associated token types (continued). The callback handler determines how security tokens are acquired from message headers.

Token Type	Consumer or Generator	Callback Handler Class Name
X509Token	consumer	com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler
X509Token	generator	com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler
LTPAToken/LTPAPropagationToken	consumer	com.ibm.websphere.wssecurity.callbackhandler.LTPAConsumeCallbackHandler
LTPAToken/LTPAPropagationToken	generator	com.ibm.websphere.wssecurity.callbackhandler.LTPAGenerateCallbackHandler
SecureConversationToken	consumer	com.ibm.ws.wssecurity.impl.auth.callback.SCTConsumeCallbackHandler
SecureConversationToken	generator	com.ibm.ws.wssecurity.impl.auth.callback.WSTrustCallbackHandler

This button is mutually exclusive to the **Use built-in default** option.

Certificates (generator):

The fields in the Certificates section are available if you are configuring a protection token. For a generator token, you can click to select a certificate store from the listing, or click the **New** button to add a certificate store.

Certificates (consumer):

The fields in the Certificates section are available if you are configuring a protection token. For a consumer token, you can use the Trust any certificate option, or the Certificate store option, to configure the certificate store.

Certificates - Trust any certificate (consumer):

This option is applicable only to the token consumer. This option indicates that the system will trust all certificates, and does not define a specific certificate store. This option is mutually exclusive to the **Certificate store** option.

Certificates - Certificate store (consumer):

This option is applicable only the to the token consumer. Use this option to specify a certificate store collection containing intermediate certificates, which can include certificate revocation lists (CRLs). Select this option to trust the certificate store or stores specified in the entry field. This option is mutually exclusive to the **Trust any certificate** option. When you select the **Certificate store** option, the **New** button is enabled so that you can configure a new certificate store and trusted anchor store.

You can set the value of the certificate store field to the default value, which is **None**. However, the trusted anchor store value must be set to a specific value. There is no default value. The trusted anchor is required if the Trust any certificate option is not selected.

Basic authentication:

The fields in the Basic authentication section are available if you are configuring an authentication token that is not an LTPA propagation token.

For the Kerberos custom token type, you must complete the Basic Authentication section for the Kerberos login.

User name:

Specifies the user name that you want to authenticate.

Password:

Specifies the password to be authenticated. Enter a password to authenticate in this entry field.

Confirm password:

Specifies the password that you want to confirm.

Keystore:

The keystore fields are not available when the run times determines that they are not needed.

In the Keystore name list, you can click **Custom** to define a custom keystore, click one of the externally defined keystore names, or click **None** if no keystore is required.

Keystore - Name:

Specifies the name of the keystore that you want to use.

Click the name of a keystore name from this menu or select one of the following values:

None Specifies to not use a keystore.

Custom
Specifies to use a user-defined keystore. Click the **Custom keystore configuration** link to configure custom keystore and key settings.

Key:

Specifies the attributes of the key to be retrieved from the configured key store. Some fields in the Key section are not available when the run times determines that they are not needed.

When a centrally managed keystore is selected for the Keystore, the fields in the Key section are available.

Name:

Specifies the name of the key to use. The list is populated with the keys available in the selected centrally managed keystore. Select the name of the key that you want to use, or '(none)' if no key is to be used.

Alias:

Displays the alias of the key name selected

Password:

Specifies the password for the key that you want to use. This field will only be available when the run time determines that it is needed.

You cannot set a password for public keys for asymmetric encryption generator or asymmetric signature consumer. Please refer to 'avoid trouble' at the beginning of the article.

Confirm password:

Specifies the confirmation of the password for the key that you want to use. This field will only be available when the run time determines that it is needed.

Do not provide a key confirm password for public keys for asymmetric outbound encryption or inbound signature.

Keystore - Custom keystore configuration:

Specifies a link to create a custom keystore. Click this link to open a panel where you can configure a custom keystore.

Key store password:

Specifies the password that is used to access the keystore file.

Key store path:

Specifies the location of the keystore file.

Use ${*USER_INSTALL_ROOT*} in the path name because this variable expands to the product path on your machine. To change the path used by this variable, click **Environment > WebSphere variables** and click **USER_INSTALL_ROOT**.

Key store type:

Specifies the type of keystore file format

Choose one of the following values for this field:

JKS Use this option if the keystore uses the Java Keystore (JKS) format.

JCEKS
: Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in the application server. This option provides stronger protection for stored private keys by using Triple DES encryption.

JCERACFKS
: Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

PKCS12KS (PKCS12)
: Use this option if your keystore file uses the PKCS#12 file format.

Custom properties:

The fields in the Custom properties section are available for all types of token configuration.

You can add custom properties needed by the callback handler using name-value pairs.

To implement signer certificate encryption when using the JAX-WS programming model, add the custom property `com.ibm.wsspi.wssecurity.token.cert.useRequestorCert` with the value `true` on the callback handler of the encryption token generator. This implementation uses the certificate of the signer of the SOAP request to encrypt the SOAP response. This custom property is used by the response generator.

For a Kerberos custom token based on OASIS Web Services Security Specification for Kerberos Token Profile V1.1, specify the following property for token generation: `com.ibm.wsspi.wssecurity.krbtoken.clientRealm`. This specifies the name of the Kerberos realm associated with the client and allows the Kerberos client realm to initiate the Kerberos login. If not specified, the default Kerberos realm name is used. This property is optional for a single Kerberos realm environment.

The Kerberos custom property, `com.ibm.wsspi.wssecurity.krbtoken.loginPrompt`, enables the Kerberos login when the value is `true`. The default value is `false`. This property is optional.

When configuring a username token for the JAX-WS programming model, to protect against replay attacks it is strongly recommended that you add the following custom properties to the callback handler configuration. These custom properties enable and verify the nonce and timestamp for message authentication.

Property name (generator)	Property value
com.ibm.wsspi.wssecurity.token.username.addNonce	true
com.ibm.wsspi.wssecurity.token.username.addTimestamp	true

Property name (consumer)	Property value
com.ibm.wsspi.wssecurity.token.username.verifyNonce	true
com.ibm.wsspi.wssecurity.token.username.verifyTimestamp	true

Name:

Specifies the name of the custom property to use.

Custom properties are not initially displayed in this column. Click one of the following actions for custom properties:

Button	Resulting action
New	Creates a new custom property entry. To add a custom property, enter the name and value.
Delete	Removes the selected custom property.

Value:

Specifies the value of the custom property to use. With the **Value** entry field, you can enter or delete the value for a custom property.

Custom keystore settings:

Use this page to configure custom keystore files. Custom keystore files are alternatives to the key management support built into the WebSphere Application Server. The callback handler uses the custom version of the keystore configuration that includes keys.

You can configure custom keystore files for message parts when you are editing a default cell or server binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a default cell binding, complete the following actions:

1. Click **Services > Policy sets > Default policy set bindings**.
2. Click the **WS-Security** policy in the Policies table.
3. Click the **Authentication and protection** link in the Main message security policy bindings section.
4. Click a ***protection_token*** link in the Protection tokens table.
5. Click the **Callback handler** link in the Additional bindings section.
6. Select **Custom** from the list in the Keystore section.
7. Click the **Custom keystore configuration** link.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.

2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Authentication and protection** link in the Main message security policy bindings section.
7. Click a ***protection_token*** link in the Protection tokens table.
8. Click the **Callback handler** link in the Additional bindings section.
9. Select **Custom** from the list in the Keystore section.
10. Click the **Custom keystore configuration** link.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Keystore:

Use this section to specify information about the custom keystores.

Full path:

Specifies the full path to where the keystore file is located. Enter the path to the keystore file in this required field. You can use system variables for portions of the path. For example you might enter `${USER_INSTALL_ROOT}/etc/ws-security/myKeyStore.jks`. This field is required for the custom keystore configuration.

Type:

Specifies the type of the keystore file to use.

Password:

Specifies the password to use.

Confirm password:

Specifies the password to be use and confirms the one entered in the **Password** field.

Key:

Use this section to specify information about the key.

Name:

Specifies the name of the key to use. Enter the name of the key to be used in this required field.

Alias:

Specifies the alias name of the key that you want to use. Enter the alias of the name of the key to use in this required field.

Password:

Specifies the password for the key that you want to use.

You cannot set a password for public keys for asymmetric signature inbound and encryption outbound. The Password and Confirm Password fields display only for the following:

Table 17. Keystore configuration for password and confirm password fields. The keystore is used for message authentication and protection.

Client or server	Asymmetric value	Key
client	asymmetric signature outbound	AsymmetricBindingInitiatorSignatureToken0
client	asymmetric encryption inbound	AsymmetricBindingInitiatorEncryptionToken0
server	asymmetric signature outbound	AsymmetricBindingRecipientSignatureToken0
server	asymmetric encryption inbound	AsymmetricBindingRecipientEncryptionToken0

Confirm password:

Specifies the confirmation of the password for the key that you want to use. Enter the password that you entered in the Password field to confirm.

Similar to the Password field, you cannot confirm the password for public keys for asymmetric signature inbound and encryption outbound.

Caller settings

Use this page to configure the caller settings. The caller specifies the token or message part that is used for authentication.

You can configure the caller settings for message parts when you are editing a default cell or server binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a general provider binding, complete the following actions:
1. Click **Services > Policy sets > General provider policy sets bindings**.
2. Click the **WS-Security** policy in the Policies table.
3. Click the **Authentication and protection** link in the Main message security policy bindings section.
4. Click the **Caller** link in the Main message security policy bindings section.
5. Click **New**.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link in the Web Services Properties section. The caller settings are available only for the service provider policy sets and bindings. The caller settings are not available for service client policy sets and bindings.
4. Select a binding. You must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Caller** link in the Main message security policy bindings section.
7. Click **New**.

   Note: When you create a new caller it will automatically be assigned the next available order. You can change the order of preference, as described in the Order section below.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Name

Specifies the name of the caller to use for authentication. Enter a caller name in this required field. This arbitrary name identifies this caller setting.

Order

Specifies the order of preference for the callers. The order determines which caller will be utilized when multiple authentication tokens are received.

You can change the order of preference by moving a caller up or down in the list. Click the checkbox next to a caller name to select the caller, then click the **Move up** button to move the caller higher in the list, or click the **Move down** button to move the caller to a lower position in the preference order.

Button	Resulting Action
Move up	Moves the order of the selected caller up in the caller list.
Move down	Moves the order of the selected caller down in the caller list.

Note: The order column displays only for bindings using the new namespace. If a binding with multiple callers was migrated to the new namespace, then the callers do not have an order. In that case, an error message is displayed. When this occurs, select a caller in the table and then click either **Move up** or **Move down** to assign an order to each caller. Callers must have orders assigned before you save the bindings or use the bindings with an application.

Caller identity local part

Specifies the local name of the caller to use for authentication. Enter a caller identity local name in this required field.

When specifying an LTPA caller, use LTPA as the local name for a caller that uses an older binding, prior to IBM WebSphere Application Server, Version 7.0. Newer bindings for IBM WebSphere Application Server, Version 7.0 and later should use LTPAv2 as the local name. Specifying LTPAv2 allows both LTPA and LTPAv2 tokens to be consumed, unless the Enforce token version option is selected on the token consumer.

Table 18. Caller identity namespace URI field description. The table lists the possible values for the Caller identity namespace URI field description.

Information	Value
Default	String

Caller identity namespace URI

Specifies the uniform resource identifier (URI) of the caller to use for authentication. Enter a caller URI in this field.

When specifying an LTPA caller, use http://www.ibm.com/websphere/appserver/tokentype/5.0.2 as the URI for a caller that uses an older binding, prior to IBM WebSphere Application Server, Version 7.0. Newer bindings for IBM WebSphere Application Server, Version 7.0 and later should use the http://www.ibm.com/websphere/appserver/tokentype URI.

Table 19. Possible values for the caller identity. The table provides a list of the Caller identity local part and the Caller identity namespace URI field values as applicable. A Caller identity namespace URI value is not needed unless it is otherwise specified in the table. The caller identity is used for message authentication.

Token type	Caller identity local part	Caller identity namespace URI
Username token 1.0	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken	

Table 19. Possible values for the caller identity (continued). The table provides a list of the Caller identity local part and the Caller identity namespace URI field values as applicable. A Caller identity namespace URI value is not needed unless it is otherwise specified in the table. The caller identity is used for message authentication.

Token type	Caller identity local part	Caller identity namespace URI
Username token 1.1	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken	
X509 certificate token	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3	
X509 certificates in a PKIPath	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1	
A list of X509 certificates and CRLs in a PKCS#7	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7	
LTPA token	LTPA	http://www.ibm.com/websphere/appserver/tokentype/5.0.2
LTPA token	LTPAv2	http://www.ibm.com/websphere/appserver/tokentype
LTPA propagation token	LTPA_PROPAGATION	http://www.ibm.com/websphere/appserver/tokentype
SAML 1.1 token	http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1	
SAML 2.0 token	http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0	
Kerberos token	http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ	

Note: If you specify a custom value type for a custom token, you must specify the Caller identity local part and Caller identity namespace URI values. For example, you might enter Custom in the Caller identity local part value field and http://www.ibm.com/custom in the Caller identity namespace URI field.

Signing part reference

When the trusted identity is based on a signing token, select the signing part reference that represents the message parts signed by that token.

If you select the Signing part reference option, you must specify a callback handler for the bindings to work properly.

Use identity assertion

Specifies whether identity assertion is used when authenticating.

Select this check box if you want to use identity assertion. When you select this checkbox, the **Trusted identity local name** and **Trusted identity namespace URI** fields are enabled.

Trusted identity local name

Specifies the trusted identity local name when the identity assertion is used.

If you select the **Use identity assertion** option and a trust token exists in the WS-Security policy, you must provide a value for the **Trusted identity local name** field for the bindings to work properly.

Trusted identity URI

Specifies the trusted identity uniform resource identifier (URI).

Callback handler

Specifies the class name of the callback handler. Enter the class name of the callback handler in this field.

If you provide a value for the **Trusted identity local name** field and you do not set the token consumer for the trust token to **Trust any certificate**, then you must set the value in this **Callback handler** field to `com.ibm.ws.wssecurity.impl.auth.callback.TrustedIdentityCallbackHandler`.

When you provide a callback handler name, you must specify the trusted identities as callback handler custom properties. For example:

```
property name="trustedId_0", value="CN=Bob,O=ACME,C=US"
property name="trustedId_1", value="user1"
```

JAAS login

Specifies the Java Authentication and Authorization Service (JAAS) application login. You can enter a JAAS login, select one from the menu, or click **New** to add a new one.

For information on updating the Kerberos system JAAS login module for JAX-WS applications, read the topic Updating the system JAAS login with the Kerberos login module.

Custom properties – Name

Specifies the name of the custom property.

Custom properties are not initially displayed in this column. Select one of the following actions for custom properties:

Button	Resulting Action
New	Creates a new custom property entry. To add a custom property, enter the name and value.
Edit	Specifies that you can edit the custom property value. At least one custom property must exist before this option is displayed.
Delete	Removes the selected custom property.

Custom properties – Value

Specifies the value of the custom property that you want to use. Use the Value field to add, edit, or delete the value for a custom property.

Caller collection

The caller specifies the token or message part that you want to use for authentication. Use this administrative console page to access, view and configure the caller settings for message parts.

To configure general bindings for tokens and message parts that are required by the policy set, complete the following steps.

1. To access and configure the general bindings, click **Services > Policy sets > General provider policy set bindings**. The caller settings are available only for the service provider policy sets and bindings. The caller settings are not available for service client policy sets and bindings.
2. Click the **WS-Security** policy in the Policies table.
3. Click the **Caller** link in the Main message security policy bindings section.

To view and configure application specific bindings for tokens and message parts that are required by a policy set, complete the following steps:

1. Click **Applications > Application Types > WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider.
3. Click the **Service provider policy sets and bindings** link in the Web Services Properties section. The caller settings are available only for the service provider policy sets and bindings. The caller settings are not available for service client policy sets and bindings.
4. Select a binding. You must have previously attached a policy set and assigned an application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Caller** link in the Main message security policy bindings section.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Name

Specifies the name of the caller to use for authentication. Select a caller name from this field.

The following actions are available to work with callers.

Button	Resulting Action
New	Opens the Caller settings page, which you use to add a caller.
Delete	Removes the selected caller.

Order

This number specifies the order of preference for the configured callers. If multiple caller tokens are found in an incoming message, the caller used for authentication will be the one with highest priority, based on decreasing order of preference.

You can change the order of preference using the **Move Up** and **Move Down** buttons.

Button	Resulting Action
Move Up	Moves the selected caller up in the order of preference, switching positions with the immediately preceding caller. The selected caller is now preferred over the caller that you demoted in the list.
Move Down	Moves the selected caller lower in the order of preference, switching positions with the caller following it. The demoted caller is now lower in preference than the caller that was previously below it.

Caller Identity Local Part

Specifies the local identity part of the caller to use for authentication.

Caller Identity URI

Specifies the uniform resource identifier (URI) of the caller to use for authentication.

Message expiration settings

Use this page to define settings for message expiration, if and when messages expire. When you specify message expiration, the message expires after the specified interval of time passes.

You can define message expiration settings for tokens and message parts when you are editing a default cell or server binding. You can also configure application specific bindings for tokens and message parts that are required by the policy set.

To view this administrative console page when you are editing a default cell binding, complete the following actions:

1. Click **Services > Policy sets > Default policy set bindings**.
2. Click the **WS-Security** policy in the Policies table.
3. Click the **Message expiration** link in the Main message security policy bindings section.

To view this administrative console page when you are configuring application specific bindings for tokens and message parts that are required by the policy set, complete the following actions:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.

2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Message expiration** link in the Main message security policy bindings section.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Message expiration

Specifies whether message expiration is enabled. To enable message expiration, select this check box. Leave it unchecked to disable message expiration.

Message timeout interval

Specifies the time, in minutes, for the message to time out if message expiration is enabled. This field is enabled only when you select the **Enable message expiration** check box.

Actor roles settings

Use this page to define settings for SOAP actor roles. The SOAP actor, also known as the SOAP role, defines the intermediary or ultimate recipient of a message.

To view this administrative console page use one of the following options:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** in the Web Services Properties section.
4. Select a binding. The binding must have previously attached a policy set and assigned a application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Actor roles** link in the Main message security policy bindings section.

This administrative console page applies only to Java API for XML Web Services (JAX-WS) applications.

Inbound actor role URI

Specifies the name of the uniform resource identifier (URI) for the inbound actor role.

Outbound actor role URI

Specifies the name of the uniform resource identifier (URI) for the outbound actor role.

Securing web services

The Web Services Security specification defines core facilities for protecting the integrity and confidentiality of a message, and provides mechanisms for associating security-related claims with a message. Web Services Security, an extension of the IBM web services engine, provides a quality of service.

Securing web services applications at the transport level

Transport-level security is a well-known and often used mechanism to secure HTTP Internet and intranet communications. Transport level security can be used to secure web services messages. Transport-level security functionality is independent from functionality that is provided by message-level security (WS-Security) or HTTP basic authentication.

Before you begin

You can use either message-level security (WS-Security) or transport-level security, or a combination of both. The following examples are common usage scenarios, but are not an exhaustive list of all possible scenarios:

- Use message-level security when security is essential to the Web service application. HTTP basic authentication uses a user name and password to authenticate a service client to a secure endpoint. The basic authentication is encoded in the HTTP request that carries the SOAP message. When the application server receives the HTTP request, the user name and password are retrieved and verified using the authentication mechanism specific to the server.

  Important: With message-level security, if you are not using the default outbound secure sockets layer (SSL) port of 443, ensure that the dynamic outbound endpoint for SSL is configured properly for your configuration.
- Use transport-level security to enable basic authentication. Transport-level security can be enabled or disabled independently from message-level security. Transport-level security provides minimal security. You can use this configuration when a web service is a client to another web service.
- Use SSL for confidentiality and integrity and HTTP Basic Authentication for authentication.
- Use SSL for confidentiality and integrity and WS-Security for authentication. For example, a Username token or LTPA token can be used for authentication.
- Use WS-Security for both confidentiality and integrity, and authentication.

About this task

Transport-level security is based on Secure Sockets Layer (SSL) or Transport Layer Security (TLS) that runs beneath HTTP. HTTP, the most used Internet communication protocol, is currently also the most popular protocol for web services. HTTP is an inherently insecure protocol because all information is sent in clear text between unauthenticated peers over an insecure network. To secure HTTP, transport-level security can be applied.

Transport level security can be used to secure web services messages. However, transport-level security functionality is independent from functionality that is provided by WS-Security or HTTP Basic Authentication.

SSL and TLS provide security features including authentication, data protection, and cryptographic token support for secure HTTP connections. To run with HTTPS, the service port address must be in the form `https://`. The integrity and confidentiality of transport data, including SOAP messages and HTTP basic authentication, is confirmed when you use SSL and TLS.

WebSphere Application Server uses the Java Secure Sockets Extension (JSSE) package to support SSL and TLS.

This task is one of several ways that you can configure the HTTP outbound transport level security for a web service acting as a client to another Web service server. You can also configure the HTTP outbound transport level security with an assembly tool or by using the Java properties. If you do not configure the HTTP outbound transport level security, the web services runtime defers to the Java Platform, Enterprise Edition (Java EE) security runtime in the WebSphere product for an effective Secure Sockets Layer (SSL) configuration. If there is no SSL configuration with the Java EE security runtime in the WebSphere product, the Java Secure Socket Extension (JSSE) system properties are used.

You can define additional HTTP transport properties for web services applications. Use the additional properties to manage the connection pool for HTTP outbound connections, configure the content encoding of the HTTP message, enable HTTP persistent connection, and resend the HTTP request when a timeout occurs.

Procedure

1. Develop and assemble a web services application. You can configure and assemble the HTTP outbound transport level security for the application with an assembly tool.
2. Deploy the application. For more information about deploying web services applications, read about deploying Web services.
3. Configure transport level security for the application. You can use one of the following methods to configure HTTP outbound transport level security.
   - Configure HTTP outbound transport level security using the administrative console.
   - Configure HTTP outbound transport-level security using Java properties.
4. Define additional HTTP transport properties for the Web services application. Use one of the following methods to define additional HTTP transport properties:
   - Configure additional HTTP transport properties using the JVM custom property panel in the administrative console.
   - Configure additional HTTP transport properties using an assembly tool.

Results

By completing these steps, you have secured web services applications at the transport level.

Authenticating web services clients using HTTP basic authentication

A simple way to provide authentication data for the service client is to authenticate to the protected service endpoint by using HTTP basic authentication. *HTTP basic authentication* uses a user name and password to authenticate a service client to a secure endpoint.

Before you begin

You can use either message-level security (WS-Security) or transport-level security:

- Use *message-level security* when security is essential to the web service application. HTTP basic authentication uses a user name and password to authenticate a service client to a secure endpoint. The basic authentication is encoded in the HTTP request that carries the SOAP message. When the application server receives the HTTP request, the user name and password are retrieved and verified using the authentication mechanism specific to the server.
- Use *transport-level security* to enable basic authentication. Transport-level security can be enabled or disabled independently from message-level security. Transport-level security provides minimal security. You can use this configuration when a web service is a client to another web service.

About this task

WebSphere Application Server can have several resources, including web services, protected by a Java Platform, Enterprise Edition (Java EE) security model.

HTTP basic authentication is orthogonal to the security support provided by WS-Security or HTTP Secure Sockets Layer (SSL) configuration.

A simple way to provide authentication data for the service client is to authenticate to the protected service endpoint using HTTP basic authentication. The basic authentication is encoded in the HTTP request that carries the SOAP message. When the application server receives the HTTP request, the user name and password are retrieved and verified using the authentication mechanism specific to the server.

Although the basic authentication data is base64-encoded, sending data over HTTPS is recommended. The integrity and confidentiality of the data can be protected by the SSL protocol.

In some cases, a firewall is present using a pass-through HTTP proxy server. The HTTP proxy server forwards the basic authentication data into the Java EE application server. The proxy server can also be protected. Applications can specify the proxy data by setting properties in a stub object.

Procedure

1. Develop and assemble a web services application. You can configure and assemble HTTP authentication for the application using an assembly tool, or programmatically. Modify the HTTP properties programmatically if you want the values that are set programmatically to take precedence over the values that are defined in the binding. If you configure HTTP basic authentication programmatically, the properties are configured in the Stub or Call instance. However, you only can programmatically configure HTTP proxy authentication.
2. Deploy the application. For more information about deploying web services applications, read about deploying Web services.
3. Configure HTTP authentication for the application. If you choose to configure HTTP basic authentication with the administrative console, the Web Services Security binding information is modified.

Securing JAX-WS web services using message-level security

Web Services Security standards and profiles address how to provide message-level protection for messages that are exchanged in a web service environment.

Before you begin

Before you begin this task, you must develop and deploy a JAX-WS application. See the topic "JAX-WS" for more information.

About this task

Java API for XML-Based Web Services (JAX-WS) is the next generation web services programming model complimenting the foundation provided by the Java API for XML-based RPC (JAX-RPC) programming model. Using JAX-WS, development of web services and clients is simplified with greater platform independence for Java applications through the use of dynamic proxies and Java annotations. JAX-WS simplifies application development through support of a standard, annotation-based model to develop web service applications and clients. A required part of the Java Platform, Enterprise Edition 5 (Java EE 5), JAX-WS is also known as JSR 224.

JAX-WS applications can be secured with Web Services Security in one of two ways. The application can be secured using policy sets, or through the use of the Web Services Security API (WSS API). The WSS API can only be used to secure a JAX-WS client application. The following sections describe both methods.

Procedure

1. Learn about Web Services Security.
2. Decide which programming model, JAX-WS or JAX-RPC, works best for securing your web services applications. This procedure uses the JAX-WS programming model.
3. Configure the security bindings, or migrate an application and associated bindings. For more information about bindings, read about defining and managing policy set bindings.
4. Develop and assemble a JAX-WS application.
5. Deploy the JAX-WS application.
6. Configure and administer the Web Services Security runtime environment. Read about signing and encrypting message parts using policy sets to find out how to specify the required message-level protection. The policy specifies what protection will be applied, including which message parts to sign

or encrypt, and the token types and algorithms to use. For complete information about policy sets, read about managing policy sets using the administrative console.

7. Configure policy sets through metadata exchange (WS-MetadataExchange). In WebSphere Application Server Version 7.0 and later, using JAX-WS, you can enable the Web Services Metadata Exchange (WS-MetadataExchange) protocol so that the policy configuration of the service provider is included in the WSDL and is available to a WS-MetadataExchange GetMetadata request. One advantage of using the WS-MetadataExhange protocol is that you can apply message-level security to WS-MetadataExchange GetMetadata requests by using a suitable system policy set. Another advantage is that the client does not have to match the provider configuration, or have a policy set attached. The client only needs the binding information, and then the client can operate based on the provider policy, or based on the intersection of the client and provider policies. You can configure a service provider to share its policy configuration using the administrative console. For more information, read the following topics:
   - Configuring security for a WS-MetadataExchange request
   - Configuring a service provider to share its policy configuration
   - Transformation of policy and binding assertions for WSDL

Securing JAX-RPC web services using message-level security

Standards and profiles address how to provide protection for messages that are exchanged in a web service environment.

Before you begin

best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

About this task

To secure web services with WebSphere Application Server, you must specify several different configurations. Although there is not a specific sequence in which you must specify these different configurations, some configurations reference other configurations. See “Web Services Security configuration considerations” on page 222.

Web service security is supported in the managed web service container. To establish a managed environment and to enforce constraints for Web Services Security, you must perform a Java Naming and Directory Interface (JNDI) lookup on the client to resolve the service reference.

Because of the relationship between the different Web Services Security configurations, it is recommended that you specify the configurations on each level of the configuration in the following order. You can choose to configure Web Services Security for the application level, the server level or the cell level as it depends upon your environment and security needs.

Procedure

1. Learn about Web Services Security.
2. Decide which programming model, JAX-WS or JAX-RPC, works best for securing your web services applications. This procedure uses the JAX-RPC programming model.

3. Configure Web Services Security. You can choose to configure Web Services Security for the application level, the server level, the cell level, or the platform level, depending on your environment and security needs. Cell-level configuration is supported only in a network deployment environment.
4. Specify the application-level configuration.
5. Specify the server-level configuration.
6. Specify the cell-level configuration. Cell-level configuration is supported only in a network deployment environment.
7. Specify the platform-level configuration.
8. Develop and assemble a JAX-RPC application, or migrate an existing application. Assemble your Web Services Security-enabled application using an assembly tool. For more information, read about assembly tools. Prior to modifying a Web Services Security-enabled application in the WebSphere Application Server administrative console, you must assemble your application using an assembly tool. Although you can modify some of the application settings using the administrative console, you must configure the generator and the consumer security constraints using an assembly tool.
9. Deploy the JAX-RPC application.

Results

After completing these steps for WebSphere Application Server, you have secured web services.

Securing web services using Security Markup Assertion Language (SAML)

The Security Assertion Markup Language (SAML) is an XML-based OASIS standard for exchanging user identity and security attributes information. Using SAML, a client can communicate assertions regarding the identity, attributes, and entitlements of a SOAP message. You can apply policy sets to JAX-WS applications to use SAML assertions in web services messages and in web services usage scenarios. Use SAML assertions to represent user identity and user security attributes, and optionally, to sign and to encrypt SOAP message elements.

Procedure

1. Learn about SAML.
2. Configure SAML application support.

   Security Assertion Markup Language (SAML) is an XML-based, OASIS standard for exchanging user identity and security attributes information. You can use the SAML function to apply a default policy to use SAML assertions in web services messages and in web services usage scenarios. In a typical SAML usage scenario, you authenticate to a security domain and request an identity provider to issue SAML assertions.

   In WebSphere Application Server Version 7.0.0.7 and later, to use the SAML default policy sets, sample SAML general bindings, and JAAS login configuration settings for SAML, you were required to set up the SAML configuration, which is stored in a profile. In WebSphere Application Server Version 8.5, the SAML feature is available in all profiles by default.
3. Develop and assemble a SAML application.
4. Deploy the SAML application.

Authenticating web services using generic security token login modules

You can use the generic security token login modules to issue, validate, and exchange security tokens using an external Security Token Service (STS).

Procedure

1. Learn about generic security token login modules.

The generic security token login modules generate and consume tokens using WS-Trust Issue and WS-Trust Validate requests. As a result of these requests, the login module issues, validates, or exchanges tokens with a WS-Trust Security Token Service, such as the service that is provided with the IBM Tivoli Federated Identity Manager.

2. Administering a generic security token login module.

   To use the generic security login module features, you must configure the login module and the callback handler for both the token generator and the token consumer.

Web Services Security concepts

The Web Services Security specification defines core facilities for protecting the integrity and confidentiality of a message, and provides mechanisms for associating security-related claims with a message.

Web Services Security concepts

The Web Services Security specification defines core facilities for protecting the integrity and confidentiality of a message, and provides mechanisms for associating security-related claims with a message.

What is new for securing web services:

In WebSphere Application Server, there are many security enhancements for web services. The enhancements include supporting sections of the Web Services Security (WS-Security) specifications and providing architectural support for plugging in and extending the capabilities of security tokens.

Enhancements from the supported Web Services Security specifications

Since September 2002, the Organization for the Advancement of Structured Information Standards (OASIS) has been developing the Web Services Security (WS-Security) for SOAP message standard.

In April 2004, OASIS released the Web Services Security Version 1.0 specification, which is a major milestone for securing web services. In Feburary 2006, the specification was updated to Version 1.1. This specification is the foundation for other Web Services Security specifications and is also the basis for the Basic Security Profile (WS-I BSP) Version 1.0 specification, which was approved in March 2007.See the Basic Security Profile web page for more information.

Web Services Security Version 1.1 is a strategic move towards Web Services Security interoperability, and an important part of the Web Services Security roadmap. For more information on the Web Services Security roadmap, see Security in a Web Services World: A Proposed Architecture and Roadmap.

WebSphere Application Server supports the following OASIS specifications and WS-I profiles:

- OASIS: Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)
- OASIS: Web Services Security: UsernameToken Profile 1.1
- OASIS: Web Services Security: Kerberos Token Profile 1.1
- OASIS: WS-SecurityPolicy 1.2
- OASIS: WS-SecureConversation 1.3
- OASIS: WS-Trust 1.3
- Basic Security Profile (WS-I BSP) 1.0
- OASIS: Web Services Security: SAML Token Profile 1.1

The Security Assertion Markup Language (SAML) is an XML-based OASIS standard for exchanging user identity and security attributes information. Using SAML, a client can communicate assertions regarding the identity, attributes, and entitlements of a SOAP message. Using the SAML function in WebSphere Application Server, you can apply policy sets to JAX-WS applications to use SAML assertions in web services messages and in web services usage scenarios. Use SAML assertions to represent user identity and user security attributes, and optionally, to sign and to encrypt SOAP message elements.

For details on what parts of the previous specifications are supported in WebSphere Application Server, see "Supported functionality from OASIS specifications" on page 201.

High level features overview in WebSphere Application Server

In WebSphere Application Server, the Web Services Security for SOAP Message Version 1.1 specification is designed to be flexible and accommodate the requirements of Web services. For example, the specification does not have a mandatory security token definition. Instead, the specification defines a generic mechanism to associate the security token with a SOAP message. The use of security tokens is defined in the various Version 1.0 and 1.1 security token profiles, such as:

- The Username Token Profile
- The X.509 Token Profile
- The Kerberos Token Profile

For more information on security token profile development at OASIS, see Organization for the Advancement of Structured Information Standards.

The Web Services Security for SOAP Message Version 1.1 updates the Web Services Security for SOAP Message core specification and the various security token profiles. For this release, WebSphere Application Server implements the Username Token Profile 1.1 and the X.509 Token Profile 1.1, which includes support for the Thumbprint type of security token reference. In addition, it supports the signature confirmation and encrypted header portions of the Web Services Security Version 1.1 standard.

Important: The wire format (such as namespaces) in the WS-SecureConversation and WS-Trust 1.3 specification has changed. WebSphere Application Server tolerates requests formatted according to both the Submission Drafts and version 1.3 specifications, but you must ensure that the correct version is used when clients are communicating with a Web Services Feature Pack service provider. You can disable tolerance of the older format for WS-SecureConversation and WS-Trust 1.3 endpoints. Submission Drafts requests are not interoperable with version 1.3 standards.

WebSphere Application Server supports pluggable security tokens. The pluggable architecture is enhanced to support the Web Services Security specifications, other profiles, and other Web Services Security specifications. You can learn more about the pluggable security token framework for JAX-RPC web services, and associating custom security tokens with SOAP messages, by reading these articles on the IBM developerWorks® website:

- Security for JAX-RPC Web services, Part 1: Generating custom tokens
- Security for JAX-RPC Web services, Part 2: Consuming custom tokens

WebSphere Application Server includes the following key enhancements:

- Support for the LTPA version 2 token
- Support for configuration of multiple callers, and an order attribute on the caller to determine which caller is used for the WebSphere credential
- Support for the published WS-SecurityPolicy version 1.2 specification embedded in WSDL
- Support for the WS-SecureConversation version 1.3 specification and the WS-Trust version 1.3 specification (used by WS-SecureConversation)
- Support for Kerberos token as defined in the WS-Kerberos Token Profile version 1.1 specification

For more information on some of these enhancements, see "Web Services Security enhancements" on page 197.

Configuration of Web Services Security

WebSphere Application Server uses the policy set model for implementing the Web Services Security Version 1.1 specification, including the Username token Version 1.1 profile, support for the Kerberos and LTPA v2 tokens, and the X.509 token version 1.1 profile. Policy sets combine configuration settings, including those for transport and message level configuration, such as WS-Addressing, WS-ReliableMessaging, WS-SecureConversation, and WS-Security. For more information on policy sets, refer to the topic Managing policy sets using the administrative console.

You can use the administrative console to configure the Web Services Security binding of a deployed application with Web Services Security constraints that are defined in the policy set.

For the X.509 Certificate Token Profile, one new type of security token reference is the Thumbprint reference, which is specified in the binding. WebSphere Application Server now supports creating and authenticating a security token by using a security token reference (STR) with a key identifier and a Thumbprint in the <KeyInfo> element. The Thumbprint key information type requires that there be a keystore with the public and private key pair instead of a shared key. To use the Thumbprint of the specified certificate, specify the keyInfo type THUMBPRINT in the bindings.

For example, a decryption key is referenced by means of the thumbprint of an associated certificate. The certificate is not included in the message. Instead, the <ds:KeyInfo> element contains a <wsse:SecurityTokenReference> element that specified the thumbprint of the specified certificate by means of the http://docs.oasis-open.org/wss/oasis-wss-soap-message-security-1.1#ThumbprintSHA1 attribute of the <wsse:KeyIdentifier> element.

To take advantage of implementations associated with the Web Services Security Version 1.1 specification, you must:

- Ensure that your applications use the Java API for XML Web Services (JAX-WS) programming model.
- Re-configure the Web Services Security constraints in the new policy set and binding format.

WebSphere Application Server provides the following tools that you can use to edit the policy set file and the binding file:

IBM assembly tools
: You can use IBM assembly tools to develop web services and configure the policy set and the binding file for Web Services Security. The tools enable you to assemble both web and Enterprise JavaBeans (EJB) modules. The assembly tools do not support direct editing of policy sets, but can import policy sets from the application server, and then attach the modified policy sets to the service. For more information, read about assembly tools.

  Note: You can use policy sets only with Java API for XML-Based Web Services (JAX-WS) applications. You cannot use policy sets with Java API for XML-based RPC (JAX-RPC) applications.

WebSphere Application Server administrative console
: You can use the administrative console to configure the Web Services Security binding of a deployed application with Web Services Security constraints that are defined in the policy set.

What is not supported

Web service security is still fairly new and some of the standards are still being defined or standardized. The following functionality is not supported in WebSphere Application Server:

- JSR-183 (Java API for Web Services Security: SOAP Message Security 1.0 specification). See the standard documentation for more information: JSR-183 (Java API for Web Services Security: SOAP Message Security 1.0 specification).

- Application programming interfaces (API) do not exist for Web Services Security in WebSphere Application Server Versions 6.0.x and later.
- SAML token profile is not supported out of the box.
- REL token profile is not supported.
- SwA profile is not supported

What is supported by the IBM Software Development Kit (SDK)

The following standards exist for the Java application programming interface for XML security and Web Services Security:

- JSR-105 (Java API for XML-Signature XPath Filter Version 2.0

  W3C Recommendation, November 2002
- JSR-106 (Java API for XML Encryption Syntax and Processing)

  W3C Recommendation, December 2002

For more information on the IBM SDK for Java Version 6, see the security information documentation.

For information on what is supported for Web Services Security in WebSphere Application Server, see "Supported functionality from OASIS specifications" on page 201.

Web Services Security enhancements:

WebSphere Application Server includes a number of enhancements for securing web services. For example, policy sets are supported in WebSphere Application Server Version 6.1 Feature Pack for Web Services, and later, to simplify security configuration for web services.

Building your applications

The Web Services Security runtime implementation used by WebSphere Application Server Version 8 is based on the Java API for XML Web Services (JAX-WS) programming model. The JAX-WS runtime environment is based on Apache Open Source Axis2, and the data model is AXIOM. Instead of deployment descriptor and bindings, a policy set is used for configuration. You can use the WebSphere Application Server administrative console to edit the application binding files associated with the policy sets. The JAX-WS runtime environment is supported for the WebSphere Application Server V6.1 Feature Pack for Web Services, and later.

The JAX-RPC programming model, which uses deployment descriptors and bindings, is still supported. Read the topic Securing JAX-RPC Web services using message level security for more information.

Using policy sets

Use policy sets to simplify your web service Quality of Service configuration.

Note: Policy sets can only be used with JAX-WS applications, in WebSphere Application Server V6.1 Feature Pack for Web Services, and later. Policy sets cannot be used for JAX-RPC applications.

Policy sets combine configuration settings, including those for transport and message level configuration, such as Web Services Addressing (WS-Addressing), Web Services Reliable Messaging (WS-ReliableMessaging), and Web Services Security (WS-Security), which includes Secure Conversation (WS-SecureConversation).

Managing trust policies

Web Services Security Trust (WS-Trust) provides the ability for an endpoint to issue a security context token for Web Services Secure Conversation (WS-SecureConversation). The token issuing support is limited to the security context token. Trust policy management defines a policy for each of the trust service operations, such as issuing, cancelling, validating, and renewing a token. A client's bootstrap policies must correspond to the WebSphere Application Server trust service policies.

Securing session-based messages

Web Services Secure Conversation provides a secured session for long running message exchanges and leveraging symmetric cryptographic algorithm. WS-SecureConversation provides the basic security for securing session-based messages exchange patterns, such as Web Services Security Reliable Messaging (WS-ReliableMessaging).

Updating message-level security

Web Services Security (WS-Security) Version 1.1 supports the following functions that update the message-level security.

- Signature confirmation
- Encrypted headers

Signature confirmation enhances the protection of XML digital signature security. The <SignatureConfirmation> element indicates that the responder has processed the signature in the request, and the signature confirmation ensures that the signature is indeed processed by the intended recipient. To process signature confirmation correctly, the initiator must preserve the signatures during the request generation processing and later must retrieve the signatures for confirmation checks even with the stateless nature of web services and the different message exchange patterns. You enable signature confirmation by configuring the policy.

The encrypted header element provides a standard way of encrypting SOAP headers, which helps inter-operability. As defined in the SOAP message security specification, the <EncryptedHeader> element indicates that a specific SOAP header (or set of headers) must be protected. Encrypting SOAP headers and parts helps to provide more secure message-level security. The EncryptedHeader element ensures compliance with the SOAP mustUnderstand processing guidelines and prevents disclosure of information contained in attributes on a SOAP header block.

Using identity assertion

In a secured environment such as an intranet, a secure sockets layer (SSL) connection or through a Virtual Private Network (VPN), it is useful to send the requester identity only without credentials, such as password, with other trusted credentials, such as the server identity. WebSphere Application Server supports the following types of identity assertions:

- A username token without a password
- An X.509 Token for a X.509 certificate

For more information about identity assertion, read the topic Trusted ID evaluator.

Signing or encrypting data with a custom token

For the JAX-RPC programming model, the key locator, or the com.ibm.wsspi.wssecurity.keyinfo.KeyLocator Java interface, is enhanced to support the flexibility of the specification. The key locator is responsible for locating the key. The local JAAS Subject is passed into the KeyLocator.getKey() method in the context. The key locator implementation can derive the key from the token, which is created by the token generator or the token consumer, to sign a message, to verify the

signature within a message, to encrypt a message, or to decrypt a message. The com.ibm.wsspi.wssecurity.keyinfo.KeyLocator Java interface is different from the version in WebSphere Application Server Version 5.x. The com.ibm.wsspi.wssecurity.config.KeyLocator interface from Version 5.x is deprecated. There is no automatic migration for the key locator from Version 5.x to Versions 6 and later. You must migrate the source code for the Version 5.x key locator implementation to the key locator programming model for Version 6 and later.

For the JAX-WS programming model, the pluggable token framework reuses the same framework from the WSS API. The same implementation for creating and validating a security token can be used in both the Web Services Security run time and the WSS API application. This simplifies the SPI programming model and makes it easier to add new or custom security token types. The redesigned SPI consists of the following interfaces:

- The JAAS CallbackHandler and JAAS Login Module create security tokens on the generator side and validate, or authenticate, security tokens on the consumer side.
- The Security Token interface, com.ibm.websphere.wssecurity.wssapi.token.SecurityToken, represents the security token that has methods to get the identity, XML format and cryptographic keys.

When using JAX-WS, the following interfaces are no longer required:

- Token Generator (com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent)
- Token Consumer (com.ibm.wsspi.wssecurity.token.TokenConsumerComponent)
- Key Locator (com.ibm.wsspi.wssecurity.keyinfo.KeyLocator)

You can learn more about custom security tokens by reading these articles on the IBM developerWorks website:

- Security for JAX-RPC Web services, Part 1: Generating custom tokens
- Security for JAX-RPC Web services, Part 2: Consuming custom tokens

Signing or encrypting any XML element

An XPath expression is used for selecting which XML element to sign or encrypt. However, an envelope signature is used when you sign the SOAP envelope, SOAP header, or Web Services Security header. In JAX-RPC web services, the XPath expression is specified in the application deployment descriptor. In JAX-WS web services, the XPath expression is specified in the WS-Security policy of the policy set.

The JAX-WS programming model uses policy sets to indicate the message parts where security should be applied. For example, the <Body> assertion is used to indicate that the body of the SOAP message is signed or encrypted. Another example is the <Header> assertion, where the QName of the SOAP header to be signed or encrypted is specified.

Signing or encrypting SOAP headers

The OASIS Web Services Security (WS-Security) Version 1.1 support provides for a standard way of encrypting and signing SOAP headers. To sign or encrypt SOAP messages, specify the QName to select header elements in the SOAP header of the SOAP message.

You can configure policy sets for signing or encrypting either by using the administrative console or by using Web Services Security APIs (WSS APIs). For more details, see the topic Securing message parts using the administrative console.

For signing, specify the following:

Name This optional attribute indicates the local name of the SOAP header to be integrity protected. If this attribute is not specified, all SOAP headers whose namespace matches the Namespace attribute are to be protected.

Namespace
This required attribute indicates the namespace of the SOAP headers to be integrity protected.

For encrypting, specify the following:

Name This optional attribute indicates the local name of the SOAP header to be confidentiality protected. If this attribute is not specified, all SOAP headers whose namespace matches the Namespace attribute are to be protected.

Namespace
This required attribute indicates the namespace of the SOAP header(s) to be confidentiality protected.

This results in an <EncryptedHeader> element that contains the <EncryptedData> element.

For Web Services Security Version 1.0 behavior, specify the com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.0 property with a value of `true` in EncryptionInfo in the bindings. Specifying this property results in an <EncryptedData> element.

For Web Services Security Version 1.1 behavior that is equivalent to WebSphere Application Server versions prior to version 7.0, specify the com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.1.pre.V7 property with a value of `true` on the <encryptionInfo> element in the binding. When this property is specified, the <EncryptedHeader> element includes a wsu:Id parameter and the <EncryptedData> element omits the Id parameter. This property should only be used if compliance with Basic Security Profile 1.1 is not required and it is necessary to send <EncryptedHeader> elements to a client or server that uses the WebSphere Application Server Version 5.1 Feature Pack for Web Services.

Supporting LTPA

Lightweight Third Party Authentication (LTPA) is supported as a binary security token in Web Services Security. Web Services Security supports both LTPA (version 1) and LTPA version 2 tokens. The LTPA version 2 token, which is more secure than version 1, is supported in WebSphere Application Server version 7.0 and later.

Extending the support for timestamps

You can insert a timestamp in other elements during the signing process besides the Web Services Security header. This timestamp provides a mechanism for adding a time limit to an element. This support is an extension for WebSphere Application Server. Other vendor implementations might not have the ability to consume a message that is generated with an additional timestamp that is inserted in the message.

Extending the support for nonce

You can insert a nonce, which is a randomly generated value, in other elements beside the Username token. The nonce is used to reduce the chance of a replay attack. This support is an extension for WebSphere Application Server. Other vendor implementations might not have the ability to consume messages with a nonce that is inserted into elements other than a Username token.

Supporting distributed nonce caching

Distributed nonce caching is a new feature for web services in WebSphere Application Server Versions 6 and later that enables you to replicate nonce data between servers in a cluster. For example, you might have application server A and application server B in cluster C. If application server A accepts a nonce with a value of `X`, then application server B creates a SoapSecurityException if it receives the nonce with the same value within a specified period of time.

Important: The distributed nonce caching feature uses the WebSphere Application Server data replication service (DRS). The data in the local cache is pushed to the cache in other servers in the same replication domain. The replication is an out-of-process call and, in some cases, is a remote call. Therefore, there is a possible delay in replication while the content of the cache in each application server within the cluster is updated. The delay might be due to network traffic, network workload, machine workload, and so on. For adequate security protection, you must enable appropriate security for the DRS cache. See the topic Multi-broker replication domains for more information.

Caching the X.509 certificate

WebSphere Application Server caches the X.509 certificates it receives, by default, to avoid certificate path validation and improve its performance. However, this change might lead to security exposure. You can disable X.509 certificate caching by using the following steps:

On the cell level:
- Click **Security > Web services**.
- Under Additional properties, click **Properties > New**.
- In the Property name field, type `com.ibm.ws.wssecurity.config.token.certificate.useCache`.
- In the Property value field, type `false`.

On the server level:
- Click **Servers > Application servers > *server_name*** .
- Under Security, click **Web services: Default bindings for Web Services Security**.
- Under Additional properties, click **Properties > New**.
- In the Property name field, type `com.ibm.ws.wssecurity.config.token.certificate.useCache`.
- In the Property value field, type `false`.

Providing support for a certificate revocation list

The certificate revocation list (CRL) in WebSphere Application Server is used to enhance certificate path validation. You can specify a CRL in the collection certificate store for validation. You can also encode a CRL in an X.509 token using PKCS#7 encoding. However, WebSphere Application Server Version 6 and later do not support X509PKIPathv1 CRL encoding in a X.509 token.

Important: The PKCS#7 encoding was tested with the IBM certificate path (IBM CertPath) provider only. The encoding is not supported for other certificate path providers.

Supported functionality from OASIS specifications:

The application server supports the Organization for the Advancement of Structured Information (OASIS) Web Services Security (WS-Security) specifications.

WebSphere Application Server supports these OASIS Web Services Security Version 1.0 specifications.
- OASIS: Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)
- OASIS: Web Services Security: UsernameToken Profile 1.0
- OASIS: Web Services Security X.509 Certificate Token Profile 1.0

In WebSphere Application Server Version 6.1 Feature Pack for Web Services, and later, support for the OASIS standards has been updated to the latest versions of Web Services Security (WS-Security) specifications and tokens. Web Services Security Version 1.1 provides better security verification for signature, a standard way of encrypting SOAP headers, and meets the requirement from some of the inter-operability scenarios that use features from Web Services Security Version 1.1.

- OASIS: Web Services Security: SOAP Message Security 1.1 (WS-Security 2004) OASIS Standard Specification, 1 February 2006
- OASIS: Web Services Security UsernameToken Profile 1.1 (Standard Specification, 1 February 2006)
- OASIS: Web Services Security X.509 Certificate Token Profile 1.1 (Standard Specification, 1 February 2006)

The following standards are supported only in WebSphere Application Server Version 7.0 and later.

- WS-Security Kerberos Token Profile 1.1
- WS-SecureConversation Version 1.3
- WS-Trust Version 1.3
- WS-SecurityPolicy Version 1.2

WS-SecurityPolicy support is only available for Web Services Metadata Exchange (WS-MetadataExchange) scenarios where the assertions are embedded in the WSDL file. For more information, read the WS-MetadataExchange requests topic.

In 2007, the OASIS Web Services Secure Exchange Technical Committee (WS-SX) produced and approved the following specifications. Portions of these specifications are supported by WebSphere Application Server Version 7 and later.

- WS-SecureConversation
- WS-Trust
- WS-SecurityPolicy

OASIS: Web Services Security SOAP Message Security 1.0 and 1.1

The following table shows the aspects of the OASIS: Web Services Security: SOAP Message Security 1.0 and 1.1 specifications that are supported in WebSphere Application Server Versions 6 and later.

Table 20. Aspects of OASIS SOAP Message Security standard supported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Security header	• @S11:actor (for an intermediary) • @S11:mustUnderstand • @S12:mustUnderstand • @S12:role (S12 is the namespace prefix for http://www.w3.org/2003/05/soap-envelope when using SOAP Version 1.2)
Security tokens	• Username token (user name and password) • Binary security token (X.509 and Lightweight Third Party Authentication (LTPA) • Custom token – Other binary security token – XML token **Note:** WebSphere Application Server does not provide an implementation, but you can use an XML token with plug-in point.
Token references	• Direct reference • Key identifier • Key name • Embedded reference
Signature	Signature confirmation

Table 20. Aspects of OASIS SOAP Message Security standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Signature algorithms	• **Digest** **SHA1** http://www.w3.org/2000/09/xmldsig#sha1 **SHA256** http://www.w3.org/2001/04/xmlenc#sha256 **SHA512** http://www.w3.org/2001/04/xmlenc#sha512 • **MAC** **HMAC-SHA1** http://www.w3.org/2000/09/xmldsig#hmac-sha1 • **Signature** **DSA with SHA1** http://www.w3.org/2000/09/xmldsig#dsa-sha1 Do not use this algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP) **RSA with SHA1** http://www.w3.org/2000/09/xmldsig#rsa-sha1 • **Canonicalization** **Canonical XML (with comments)** http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments **Canonical XML (without comments)** http://www.w3.org/TR/2001/REC-xml-c14n-20010315 **Exclusive XML canonicalization (with comments)** http://www.w3.org/2001/10/xml-exc-c14n#WithComments **Exclusive XML canonicalization (without comments)** http://www.w3.org/2001/10/xml-exc-c14n# • **Transform** **STR transform** http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soapmessage- security-1.0#STR-Transform **XPath** http://www.w3.org/TR/1999/REC-xpath-19991116 Do not use the original XPATH transform if you want your configured application to be in compliance with the Basic Security Profile (BSP). **Note:** When referring to an element in a SECURE_ENVELOPE that does not carry an attribute of type ID from a ds:Reference in a SIGNATURE, you must use the XPATH Filter 2.0 Transform, http://www.w3.org/2002/06/xmldsig-filter2 **Enveloped signature** http://www.w3.org/2000/09/xmldsig#enveloped-signature **XPath Filter2** http://www.w3.org/2002/06/xmldsig-filter2 **Note:** When referring to an element in a SECURE_ENVELOPE that does not carry an ID attribute type from a ds:Reference in a SIGNATURE, you must use the XPATH Filter 2.0 Transform, http://www.w3.org/2002/06/xmldsig-filter2 **Decryption transform** http://www.w3.org/2002/07/decrypt#XML

Table 20. Aspects of OASIS SOAP Message Security standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Signature signed parts for JAX-RPC only	• WebSphere Application Server key words: – body, which signs the SOAP message body – timestamp, which signs all of the time stamps – securitytoken, which signs all of the security tokens – dsigkey, which signs the signing key – enckey, which signs the encryption key – messageid, which signs the `wsa :MessageID` element in WS-Addressing. – to, which signs the `wsa:To` element in WS-Addressing – action, which signs the `wsa:Action` element in WS-Addressing – relatesto, which signs the `wsa:RelatesTo` element in WS-Addressing wsa is the namespace prefix of http://schemas.xmlsoap.org/ws/2004/08/addressing – wscontext, which specifies the WS-Context header for the SOAP header. – wsafrom, which specifies the `<wsa:From>` WS-Addressing From element in the SOAP header. – wsareplyto, which specifies the `<wsa:ReplyTo>` WS-Addressing ReplyTo element in the SOAP header. – wsafaultto, which specifies the `<wsa:FaultTo>` WS-Addressing FaultTo element in the SOAP header. – wsaall, which specifies all of the WS-Addressing elements in the SOAP header. • XPath expression to select an XML element in a SOAP message. For more information, see http://www.w3.org/TR/1999/REC-xpath-19991116.
Signature message parts for JAX-WS only	• Body (which signs the SOAP message body) • Header (which signs one or more SOAP headers within the main SOAP header) • XPath expression to select an XML element in a SOAP message. – For more information, see http://www.w3.org/TR/1999/REC-xpath-19991116.
Encryption	EncryptedHeader element

Table 20. Aspects of OASIS SOAP Message Security standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Encryption algorithms	**Important:** Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted. • Data encryption – Triple DES in CBC: http://www.w3.org/2001/04/xmlenc#tripledes-cbc – AES128 in CBC: http://www.w3.org/2001/04/xmlenc#aes128-cbc – AES192 in CBC: http://www.w3.org/2001/04/xmlenc#aes192-cbc This algorithm requires the unrestricted JCE policy file. For more information, see the Key encryption algorithm description in the “Encryption information configuration settings: Message parts” on page 891. Do not use the 192-bit data encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP). – AES256 in CBC: http://www.w3.org/2001/04/xmlenc#aes256-cbc This algorithm requires the unrestricted JCE policy file. For more information, see the Key encryption algorithm description in the “Encryption information configuration settings: Message parts” on page 891. • Key encryption – Key transport (public key cryptography) - http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p. **Note:** • When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this one. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5. • Use of the Federal Information Processing Standard (FIPS)-compliant Java cryptography engine does not support this transport algorithm. - RSA Version 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5 – Symmetric key wrap (private key cryptography) - Triple DES key wrap: http://www.w3.org/2001/04/xmlenc#kw-tripledes - AES key wrap (aes128): http://www.w3.org/2001/04/xmlenc#kw-aes128 - AES key wrap (aes192): http://www.w3.org/2001/04/xmlenc#kw-aes192 This algorithm requires the unrestricted JCE policy file. For more information, see the Key encryption algorithm description in the “Encryption information configuration settings: Message parts” on page 891. Do not use the 192-bit data encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP). - AES key wrap (aes256): http://www.w3.org/2001/04/xmlenc#kw-aes256 This algorithm requires the unrestricted JCE policy file. For more information, see the Key encryption algorithm description in the “Encryption information configuration settings: Message parts” on page 891. • Manifests-xenc is the namespace prefix of http://www.w3.org/TR/xmlenc-core – xenc:ReferenceList – xenc:EncryptedKey Advanced Encryption Standard (AES) is designed to provide stronger and better performance for symmetric key encryption over Triple-DES (data encryption standard). Therefore, it is recommended that you use AES, if possible, for symmetric key encryption.
Encryption message parts for JAX-RPC only	• WebSphere Application Server keywords – bodycontent, which is used to encrypt the SOAP body content – usernametoken, which is used to encrypt the username token – digestvalue, which is used to encrypt the digest value of the digital signature – signature, which is used to encrypt the entire digital signature – wscontextcontent, which encrypts the content in the WS-Context header for the SOAP header. • XPath expression to select the XML element in the SOAP message – XML elements – XML element contents

Table 20. Aspects of OASIS SOAP Message Security standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Encryption message parts for JAX-WS only	• Body (which encrypts the SOAP message body content) • Header (which encrypts one or more SOAP headers within the main SOAP header, resulting in the EncryptedHeader element) • XPath expression to select an XML element in a SOAP message – For more information, see http://www.w3.org/TR/1999/REC-xpath-19991116.
Time stamp	• Within Web Services Security header • WebSphere Application Server is extended to allow you to insert time stamps into other elements so that the age of those elements can be determined.
Error handling	SOAP faults • New `failure` SOAP fault with faultcode • `The message has expired` text has been added

OASIS: Web Services Security UsernameToken Profile 1.0

The following table shows the aspects of the OASIS: Web Services Security Username Token Profile 1.0 specification that is supported in WebSphere Application Server.

Table 21. Aspects of OASIS Username Token Profile V1.0 standard supported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Password types	Text
Token references	Direct reference

OASIS: Web Services Security UsernameToken Profile 1.1

The following table shows the aspects of the OASIS: Web Services Security Username Token Profile 1.1 specification that is supported in WebSphere Application Server. Items that were previously supported for Web Services Security UsernameToken Profile 1.0 are not listed but are still supported, unless noted otherwise.

Table 22. Aspects of OASIS Username Token Profile V1.1 standard supported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Password types	Text
Token references	Direct reference

OASIS: Web Services Security X.509 Certificate Token Profile 1.0

The following table shows the aspects of the OASIS: Web Services Security X.509 Certificate Token Profile specification that are supported in WebSphere Application Server Versions 6 and later.

Table 23. Aspects of OASIS X.509 Certificate Token V1.0 standard supported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Token types	• X.509 Version 3: Single certificate http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509- token-profile-1.0#X509v3 • X.509 Version 3: X509PKIPathv1 without certificate revocation lists (CRL) http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509- token-profile-1.0#X509PKIPathv1 • X.509 Version 3: PKCS7 with or without CRLs. The IBM software development kit (SDK) supports both. The Sun Java SE Development Kit 6 (JDK 6) supports PKCS7 without CRL only.

Table 23. Aspects of OASIS X.509 Certificate Token V1.0 standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Token references	• Key identifier – subject key identifier • Direct reference • Custom reference – issuer name and serial number

OASIS: Web Services Security X.509 Certificate Token Profile 1.1

The following table shows the aspects of the OASIS: Web Services Security X.509 Certificate Token Profile 1.1 specification that are supported in WebSphere Application Server. Items that were previously supported for Web Services Security X.509 Certificate Token Profile 1.0 are not listed but are still supported, unless noted otherwise.

Table 24. Aspects of OASIS X.509 Certificate Token V1.1 standard supported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Token types	X.509 Version 1: Single certificate
Token references	Key identifier – subject key identifier • Can only reference an X.509v3 certificate • Can specify the thumbprint of the specified certificate by using the http://docs.oasis-open.org/wss/oasis-wss-soap-message-security-1.1#ThumbprintSHA1 attribute of the <wsse:KeyIdentifier> element.

OASIS: Web Services Security Kerberos Token Profile 1.1

The following table shows the aspects of the OASIS: Web Services Security Kerberos Token Profile 1.1 specification that are supported in WebSphere Application Server.

Table 25. Aspects of OASIS Kerberos Token Profile standard supported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Token types	• GSS_API Kerberos v5 token http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ • GSS_API Kerberos v5 token per RFC1510 http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510 • GSS_API Kerberos v5 token per RFC4120 http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120 • Kerberos v5 token http://docs.oasis-open.org/wss/oasiswss- kerberos-token-profile-1.1#Kerberosv5_AP_REQ • Kerberos v5 token per RFC1510 http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510 • Kerberos v5 token per RFC4120 http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ412
Token references	• Security token reference • Key identifier, which is used after the initial Kerberos v5 token is consumed • Derived key token based on the Kerberos key

OASIS: Web Services Security WS-Secure Conversation Draft and Version 1.3

The following table shows the aspects of the OASIS: WS-SecureConversation specification that are supported in WebSphere Application Server Version 6.1 Feature Pack for Web Services, and later. Support for Version 1.3 of the specification is provided in WebSphere Application Server Version 7.0 and later.

Table 26. Aspects of OASIS SecureConversation standard supported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Token types	• Security Context Token draft version: http://schemas.xmlsoap.org/ws/2005/02/sc/sct • Security Context Token Version 1.3: http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
Token references	Direct reference
Security context establishment	Security context token created by a security token service that is embedded in the WebSphere Application Server.
Renewing context	Automatic renewal of the token when its about to expire.
Cancelling context	Explicit cancel request support.
Derived keys	The following information is used to derive the keys using a shared secret from a security context: • /wsc:DerivedKeyToken/wsse:SecurityTokenReference • /wsc:DerivedKeyToken/wsc:Label • /wsc:DerivedKeyToken/wsc:Nonce • /wsc:DerivedKeyToken/wsc:Length
Error handling	SOAP faults, including: • wsc:BadContextToken • wsc:UnsupportedContextToken • wsc:RenewNeeded • wsc:UnableToRenew

OASIS: Web Services Security WS-Trust Version 1.0 Draft and Version 1.3

The following tables show the aspects of the OASIS: Web Services Security: WS-Trust Version 1.0 Draft and Version 1.3 specifications that are supported in WebSphere Application Server Version 6.1 Feature Pack for Web Services, and later.

Table 27. Aspects of OASIS Trust V1.0 and V1.3 standard supported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Namespace	http://schemas.xmlsoap.org/ws/2005/02/trust
Request header	/wsa:Action Valid options include: • http://schemas.xmlsoap.org/ws/2005/02/trust/RST/Issue • http://schemas.xmlsoap.org/ws/2005/02/trust/RST/Renew • http://schemas.xmlsoap.org/ws/2005/02/trust/RST/Cancel • http://schemas.xmlsoap.org/ws/2005/02/trust/RST/Validate

Table 27. Aspects of OASIS Trust V1.0 and V1.3 standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Request elements and attributes	/wst:RequestSecurityToken /wst:RequestSecurityToken/@Context /wst:RequestSecurityToken/wst:RequestType • Valid options include: – http://schemas.xmlsoap.org/ws/2005/02/trust/Issue – http://schemas.xmlsoap.org/ws/2005/02/trust/Renew – http://schemas.xmlsoap.org/ws/2005/02/trust/Cancel – http://schemas.xmlsoap.org/ws/2005/02/trust/Validate /wst:RequestSecurityToken/wst:TokenType • Valid options include: – for http://schemas.xmlsoap.org/ws/2005/02/sc/sct - /wst:RequestSecurityToken/wsp:AppliesTo - /wst:RequestSecurityToken/wst:Entropy - /wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret - /wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret/@Type – for http://schemas.xmlsoap.org/ws/2005/02/trust/Nonce - /wst:RequestSecurityToken/wst:Lifetime - /wst:RequestSecurityToken/wst:Lifetime/wsu:Created - /wst:RequestSecurityToken/wst:Lifetime/wsu:Expires - /wst:RequestSecurityToken/wst:KeySize - /wst:RequestSecurityToken/wst:KeyType – for http://schemas.xmlsoap.org/ws/2005/02/trust/SymmetricKey - /wst:RequestSecurityToken/wst:RenewTarget - /wst:RequestSecurityToken/wst:Renewing - /wst:RequestSecurityToken/wst:Renewing/@Allow - /wst:RequestSecurityToken/wst:Renewing/@OK - /wst:RequestSecurityToken/wst:CancelTarget - /wst:RequestSecurityToken/wst:ValidateTarget - /wst:RequestSecurityToken/wst:Issuer
Response header	/wsa:Action Valid options include: • http://schemas.xmlsoap.org/ws/2005/02/trust/RSTR/Issue • http://schemas.xmlsoap.org/ws/2005/02/trust/RSTR/Renew • http://schemas.xmlsoap.org/ws/2005/02/trust/RSTR/Cancel • http://schemas.xmlsoap.org/ws/2005/02/trust/RSTR/Validate

Table 27. Aspects of OASIS Trust V1.0 and V1.3 standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Response elements and attributes	/wst:RequestSecurityTokenResponse /wst:RequestSecurityTokenResponse/@Context /wst:RequestSecurityTokenResponse/wst:TokenType /wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken /wst:RequestSecurityTokenResponse/wsp:AppliesTo /wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken /wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference /wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference /wst:RequestSecurityTokenResponse/wst:RequestedProofToken /wst:RequestSecurityTokenResponse/wst:Entropy /wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret /wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret/@Type /wst:RequestSecurityTokenResponse/wst:Lifetime /wst:RequestSecurityTokenResponse/wst:Lifetime/wsu:Created /wst:RequestSecurityTokenResponse/wst:Lifetime/wsu:Expires /wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey /wst:RequestSecurityTokenResponse/wst:KeySize /wst:RequestSecurityTokenResponse/wst:Renewing /wst:RequestSecurityTokenResponse/wst:Renewing/@Allow /wst:RequestSecurityTokenResponse/wst:Renewing/@OK /wst:RequestSecurityTokenResponse/wst:RequestedTokenCancelled /wst:RequestSecurityTokenResponse/wst:Status /wst:RequestSecurityTokenResponse/wst:Status /wst:RequestSecurityTokenResponse/wst:Status/wst:Code • Valid responses include: – http://schemas.xmlsoap.org/ws/2005/02/trust/status/valid – http://schemas.xmlsoap.org/ws/2005/02/trust/status/invalid /wst:RequestSecurityTokenResponse/wst:Status/wst:Reason

Table 27. Aspects of OASIS Trust V1.0 and V1.3 standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Error handling	wst:InvalidRequest wst:FailedAuthentication wst:RequestFailed wst:InvalidSecurityToken wst:AuthenticationBadElements wst:BadRequest wst:ExpiredData wst:InvalidTimeRange wst:InvalidScope wst:RenewNeeded wst:UnableToRenew

Table 28. Aspects of OASIS Trust V1.3 standard supported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Namespace	http://docs.oasis-open.org/ws-sx/ws-trust/200512
Request header	/wsa:Action Valid options include: • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate • http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchIssue • http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchCancel • http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchRenew • http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchValidate

Table 28. Aspects of OASIS Trust V1.3 standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Request elements and attributes	/wst:RequestSecurityToken /wst:RequestSecurityToken/@Context /wst:RequestSecurityToken/wst:RequestType • Valid options include: – http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue – http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew – http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel – http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate – http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchIssue – http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchRenew – http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchCancel – http://docs.oasis-open.org/ws-sx/ws-trust/200512/BatchValidate /wst:RequestSecurityToken/wst:TokenType • Valid options include: – for http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct - /wst:RequestSecurityToken/wsp:AppliesTo - /wst:RequestSecurityToken/wst:Entropy - /wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret - /wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret/@Type – for http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce - /wst:RequestSecurityToken/wst:Lifetime - /wst:RequestSecurityToken/wst:Lifetime/wsu:Created - /wst:RequestSecurityToken/wst:Lifetime/wsu:Expires - /wst:RequestSecurityToken/wst:KeySize - /wst:RequestSecurityToken/wst:KeyType – for http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey - /wst:RequestSecurityToken/wst:RenewTarget - /wst:RequestSecurityToken/wst:Renewing - /wst:RequestSecurityToken/wst:Renewing/@Allow - /wst:RequestSecurityToken/wst:Renewing/@OK - /wst:RequestSecurityToken/wst:CancelTarget - /wst:RequestSecurityToken/wst:ValidateTarget - /wst:RequestSecurityToken/wst:Issuer
Response header	/wsa:Action Valid options include: • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/CancelFinal • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/RenewFinal • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/ValidateFinal

Table 28. Aspects of OASIS Trust V1.3 standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Response elements and attributes	/wst:RequestSecurityTokenResponse /wst:RequestSecurityTokenResponse/@Context /wst:RequestSecurityTokenResponse/wst:TokenType /wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken /wst:RequestSecurityTokenResponse/wsp:AppliesTo /wst:RequestSecurityTokenResponse/wst:RequestedSecurityToken /wst:RequestSecurityTokenResponse/wst:RequestedAttachedReference /wst:RequestSecurityTokenResponse/wst:RequestedUnattachedReference /wst:RequestSecurityTokenResponse/wst:RequestedProofToken /wst:RequestSecurityTokenResponse/wst:Entropy /wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret /wst:RequestSecurityTokenResponse/wst:Entropy/wst:BinarySecret/@Type /wst:RequestSecurityTokenResponse/wst:Lifetime /wst:RequestSecurityTokenResponse/wst:Lifetime/wsu:Created /wst:RequestSecurityTokenResponse/wst:Lifetime/wsu:Expires /wst:RequestSecurityTokenResponse/wst:RequestedProofToken/wst:ComputedKey /wst:RequestSecurityTokenResponse/wst:KeySize /wst:RequestSecurityTokenResponse/wst:Renewing /wst:RequestSecurityTokenResponse/wst:Renewing/@Allow /wst:RequestSecurityTokenResponse/wst:Renewing/@OK /wst:RequestSecurityTokenResponse/wst:RequestedTokenCancelled /wst:RequestSecurityTokenResponse/wst:Status /wst:RequestSecurityTokenResponse/wst:Status/wst:Code • Valid responses include: – http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid – http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/invalid /wst:RequestSecurityTokenResponse/wst:Status/wst:Reason

Table 28. Aspects of OASIS Trust V1.3 standard supported in WebSphere Application Server (continued). Use the table to determine which aspects of the OASIS standard are supported.

Supported topic	Specific aspect that is supported
Error handling	wst:InvalidRequest wst:FailedAuthentication wst:RequestFailed wst:InvalidSecurityToken wst:AuthenticationBadElements wst:BadRequest wst:ExpiredData wst:InvalidTimeRange wst:InvalidScope wst:RenewNeeded wst:UnableToRenew

Functionality that is not supported by WebSphere Application Server

The following list shows the functionality that is supported in the OASIS specifications, OASIS drafts, and other recommendations but is not supported by WebSphere Application Server Version 6 and later:

- Web Services Security SOAP Messages with Attachments (SwA) profile 1.0

  Note: When using the JAX-WS programming model, securing the SOAP Message Transmission Optimization Mechanism (MTOM) attachment is supported. See the topic Enabling MTOM for JAX-WS web services for more information.
- XrML token profile
- XML enveloping digital signature
- XML enveloping digital encryption
- The following WS-SecureConversation functionality is not supported by WebSphere Application Server:
  - Two methods for establishing security context are not supported: 1) security context token created by one of the communicating parties and propagated with a message; and 2) security context token created through negotiation or exchanges.
  - SCT propagation
  - Amending security contexts
- The following transform algorithms for digital signatures are not supported:
  - XSLT: http://www.w3.org/TR/1999/REC-xslt-19991116
  - SOAP Message Normalization

    See SOAP Version 1.2 Message Normalization for information, such as an empty header or header entry with `mustUnderstand=false` is removed, and so forth.
  - Decryption transform
- The following key agreement algorithm for encryption is not supported:
  - Diffie-Hellman: http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/Overview.html#sec-DHKeyValue
- The following canonicalization algorithm for encryption, which is optional in the XML encryption specification, is not supported:
  - Canonical XML with or without comments
  - Exclusive XML Canonicalization with or without comments

- DSA digital signature is not supported.
- Pre-agreed symmetric key data encryption is not supported.
- Auditing for nonrepudiation for digital signatures is not supported.
- In both versions of the Username Token Profile specification, the digest password type is not supported.
- In the Username Token Version 1.1 Profile specification, the key derivation based on a password is not supported.

Unsupported function for WS-Trust Version 1.0 Draft and Version 1.3

The following tables show the aspects of the OASIS: Web Services Security: WS-Trust Version 1.0 Draft and Version 1.3 specifications that are **not** supported in WebSphere Application Server Version 6.1 Feature Pack for Web Services, and later.

Table 29. Aspects of OASIS Trust V1.0 and V1.3 standard that are unsupported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are not supported.

Unsupported topic	Specific aspect that is not supported
Elements and attributes	/wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret/@Type Unsupported request options: • for http://schemas.xmlsoap.org/ws/2005/02/trust/AsymmetricKey and http://schemas.xmlsoap.org/ws/2005/02/trust/SymmetricKey – /wst:RequestSecurityToken/wst:Claims – /wst:RequestSecurityToken/wst:AllowPostdating – /wst:RequestSecurityToken/wst:OnBehalfOf – /wst:RequestSecurityToken/wst:AuthenticationType – /wst:RequestSecurityToken/wst:KeyType • for http://schemas.xmlsoap.org/ws/2005/02/trust/PublicKey – /wst:RequestSecurityToken/wst:SignatureAlgorithm – /wst:RequestSecurityToken/wst:EncryptionAlgorithm – /wst:RequestSecurityToken/wst:CanonicalizationAlgorithm – /wst:RequestSecurityToken/wst:ComputedKeyAlgorithm – /wst:RequestSecurityToken/wst:Encryption – /wst:RequestSecurityToken/wst:ProofEncryption – /wst:RequestSecurityToken/wst:UseKey – /wst:RequestSecurityToken/wst:UseKey/@Sig – /wst:RequestSecurityToken/wst:SignWith – /wst:RequestSecurityToken/wst:EncryptWith – /wst:RequestSecurityToken/wst:DelegateTo – /wst:RequestSecurityToken/wst:Forwardable – /wst:RequestSecurityToken/wst:Delegatable – /wst:RequestSecurityToken/wsp:Policy – /wst:RequestSecurityToken/wsp:PolicyReference
Response elements and attributes	/wst:RequestSecurityTokenResponseCollection /wst:RequestSecurityTokenResponseCollection/wst:RequestSecurityTokenResponse

Table 30. Aspects of OASIS Trust V1.3 standard that are unsupported in WebSphere Application Server. Use the table to determine which aspects of the OASIS standard are not supported.

Unsupported topic	Specific aspect that is not supported
Elements and attributes	/wst:RequestSecurityToken/wst:Entropy/wst:BinarySecret/@Type Unsupported request options: • for http://docs.oasis-open.org/ws-sx/ws-trust/200512/AsymmetricKey and http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey – /wst:RequestSecurityToken/wst:Claims – /wst:RequestSecurityToken/wst:AllowPostdating – /wst:RequestSecurityToken/wst:OnBehalfOf – /wst:RequestSecurityToken/wst:AuthenticationType – /wst:RequestSecurityToken/wst:KeyType • for http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey and http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer – /wst:RequestSecurityToken/wst:SignatureAlgorithm – /wst:RequestSecurityToken/wst:EncryptionAlgorithm – /wst:RequestSecurityToken/wst:CanonicalizationAlgorithm – /wst:RequestSecurityToken/wst:ComputedKeyAlgorithm – /wst:RequestSecurityToken/wst:Encryption – /wst:RequestSecurityToken/wst:ProofEncryption – /wst:RequestSecurityToken/wst:UseKey – /wst:RequestSecurityToken/wst:UseKey/@Sig – /wst:RequestSecurityToken/wst:SignWith – /wst:RequestSecurityToken/wst:EncryptWith – /wst:RequestSecurityToken/wst:DelegateTo – /wst:RequestSecurityToken/wst:Forwardable – /wst:RequestSecurityToken/wst:Delegatable – /wst:RequestSecurityToken/wsp:Policy – /wst:RequestSecurityToken/wsp:PolicyReference
Response header	/wsa:Action Unsupported Responses: • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Issue • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Renew • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Cancel • http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Validate

Web Services Security specification - a chronology:

The development of the Web Services Security specification includes information on the Organization for the Advancement of Structured Information Standards (OASIS) Web Services Security specification. The OASIS Web Services Security specification serves as a basis for securing web services in WebSphere Application Server.

best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

Advantages of using the JAX-WS programming model in WebSphere Application Server include:

- The configuration of qualities of service (QoS) is simplified when using policy sets. Policy sets combine configuration settings, including those for transport and message-level configuration. Policy sets and general bindings can be reused across multiple applications, making web services QoS more consumable.
- WS-Security for JAX-WS is supported in both a managed environment, such as a Java EE container, and unmanaged environments, such as Java Platform, Standard Edition (Java SE 6). In addition, there is an API for enabling WS-Security in the JAX-WS client.

Non-OASIS activities

Web services is gaining rapid acceptance as a viable technology for interoperability and integration. However, securing web services is one of the paramount quality of services that makes the adoption of web services a viable industry and commercial solution for businesses. IBM and Microsoft jointly published a security white paper on web services entitled Security in a Web Services World: A Proposed Architecture and Roadmap. The white paper discusses the following initial and subsequent specifications in the proposed Web Services Security roadmap:

Web service security
: This specification defines how to attach a digital signature, use encryption, and use security tokens in SOAP messages.

WS-Policy
: This specification defines the language that is used to describe security constraints and the policy of intermediaries or endpoints.

WS-Trust
: This specification defines a framework for trust models to establish trust between web services.

WS-Privacy
: This specification defines a model of how to express a privacy policy for a web service and a requester.

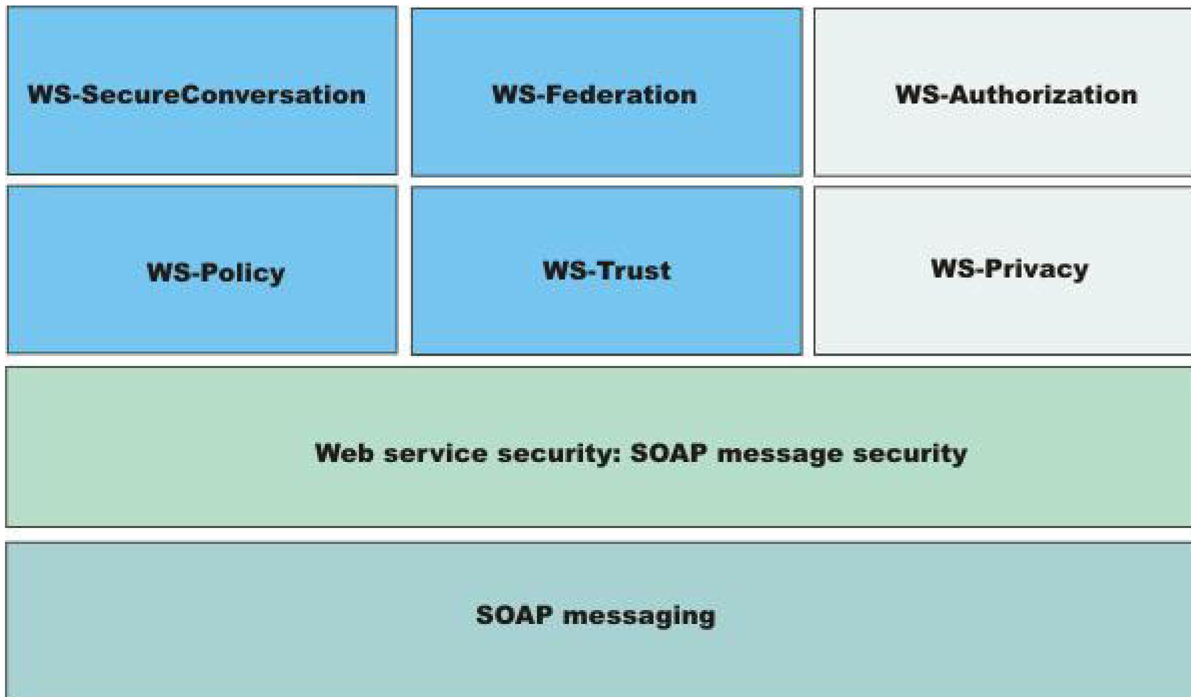
WS-SecureConversation
: This specification defines how to exchange and establish a secured context, which derives session keys between web services.

WS-Authorization

This specification defines the authorization policy for a Web service. However, the WS-Authorization specification has not been published. The existing implementation of Web Services Security is based upon the Web Services for Java Platform, Enterprise Edition (Java EE) or Java Specification Requirements (JSR) 109 specification. The implementation of Web Services Security leverages the Java EE role-based authorization checks. For conceptual information, read about role-based authorization. If you develop a web service that requires method-level authorization checks, then you must use stateless session beans to implement your web service. For more information, read about securing enterprise bean applications.

If you develop a web service that is implemented as a servlet, you can use coarse-grained or URL-based authorization in the web container. However, in this situation, you cannot use the identity from Web Services Security for authorization checks. Instead, you can use the identity from the transport. If you use SOAP over HTTP, then the identity is in the HTTP transport.

This following figure shows the relationship between these specifications:



In April 2002, IBM, Microsoft, and VeriSign proposed the Web Services Security (WS-Security) specification on their websites as depicted by the green box in the previous figure. This specification included the basic ideas of a security token, XML digital signature, and XML encryption. The specification also defined the format for user name tokens and encoded binary security tokens. After some discussion and an interoperability test based on the specification, the following issues were noted:

- The specification requires that the Web Services Security processors understand the schema correctly so that the processor distinguishes between the ID attribute for XML digital signature and XML encryption.
- The freshness of the message, which indicates whether the message complies with predefined time constraints, cannot be determined.
- Digested password strings do not strengthen security.

In August 2002, IBM, Microsoft, and VeriSign published the *Web Services Security Addendum*, which attempted to address the previously listed issues. The following solutions were addressed in the addendum:

- Require a global ID attribute for XML signature and XML encryption.
- Use time stamp header elements that indicate the time of the creation, receipt, or expiration of the message.
- Use password strings that are digested with a time stamp and nonce, which is a randomly generated token.

The specifications for the blue boxes in the previous figure have been proposed by various industry vendors and various interoperability events have been organized by the vendors to verify and refine the proposed specifications.

OASIS activities

In June 2002, OASIS received a proposed Web Services Security specification from IBM, Microsoft, and VeriSign. The Web Services Security Technical Committee (WSS TC) was organized at OASIS soon after the submission. The technical committee included many companies including IBM, Microsoft, VeriSign, Sun Microsystems, and BEA Systems.

In September 2002, WSS TC published its first specification, Web Services Security Core Specification, Working Draft 01. This specification included the contents of both the original Web Services Security specification and its addendum.

The coverage of the technical committee became larger as the discussion proceeded. Because the Web Services Security Core Specification allows arbitrary types of security tokens, proposals were published as profiles. The profiles described the method for embedding tokens, including Security Assertion Markup Language (SAML) tokens and Kerberos tokens embedded into the Web Services Security messages. Subsequently, the definitions of the usage for user name tokens and X.509 binary security tokens, which were defined in the original Web Services Security Specification, were divided into the profiles.

WebSphere Application Server Versions 5.0.2, 5.1, and 5.1.1 support the following specifications:

- Web Services Security: SOAP Message Security Draft 13 (formerly Web Services Security Core Specification)
- Web Services Security: Username Token Profile Draft 2

In April 2004, the Web Service Security specification (officially called Web Services Security: SOAP Message Security Version 1.0) became the Version 1.0 OASIS standard. Also, the Username token and X.509 token profiles are Version 1.0 specifications. WebSphere Application Server 6 and later support the following Web Services Security specifications from OASIS:

- Web Services Security: SOAP Message Security 1.0 specification
- Web Services Security: Username Token 1.0 Profile
- Web Services Security: X.509 Token 1.0 Profile

In February 2006, the core Web Service Security specification was updated and became the Version 1.1 OASIS standard. Also, the Username token, X.509 token profile, and Kerberos token profile were updated to the Version 1.1 specifications. Portions of the following Web Services Security specifications from OASIS are supported in WebSphere Application Server, specifically signature confirmation, encrypted header, and thumbprint references:

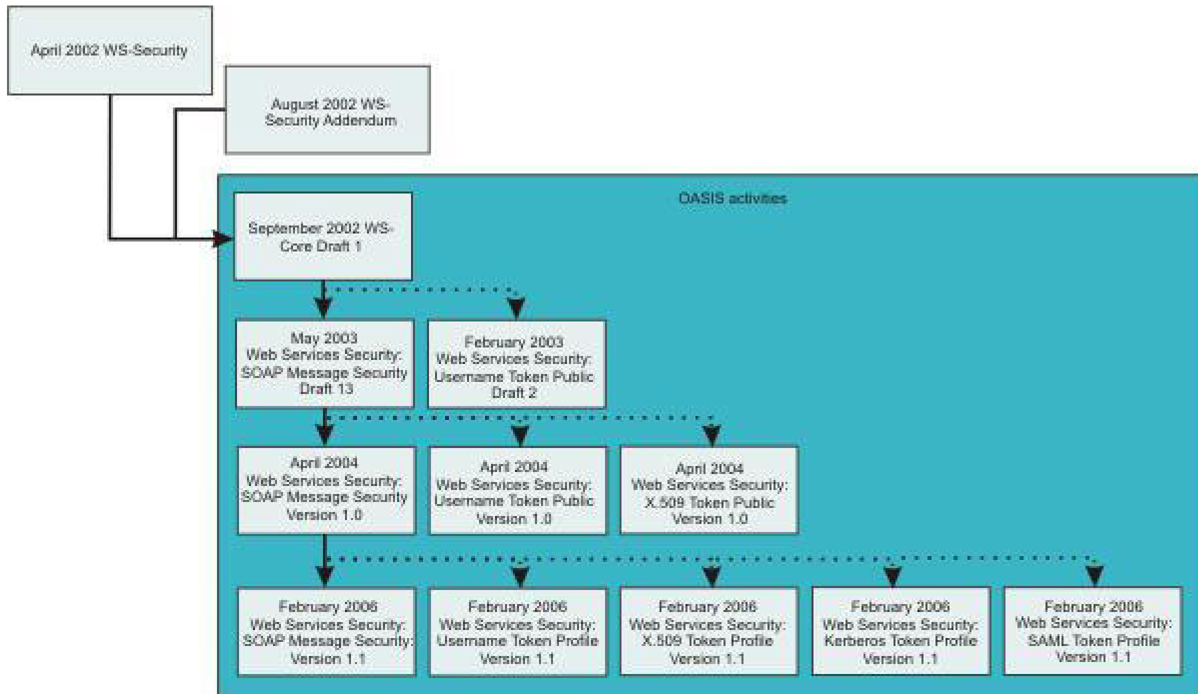
- OASIS: Web Services Security: SOAP Message Security 1.1 (WS-Security 2004) OASIS Standard Specification, 1 February 2006
- OASIS: Web Services Security UsernameToken Profile 1.1 OASIS Standard Specification, 1 February 2006
- OASIS: Web Services Security X.509 Certificate Token Profile 1.1 OASIS Standard Specification, 1 February 2006

The following specification describes the use of Kerberos tokens with respect to the Web Services Security message security specifications. The specification defines how to use a Kerberos token to support authentication and message protection: OASIS: Web Services Security Kerberos Token Profile 1.1 OASIS Standard Specification, 1 February 2006.

In 2007, the OASIS Web Services Secure Exchange Technical Committee (WS-SX) produced and approved the following specifications. Portions of these specifications are supported by WebSphere Application Server Version 7 and later.

- WS-SecureConversation
- WS-Trust
- WS-SecurityPolicy

The following figure shows the various Web Services Security-related specifications.



WebSphere Application Server also provides plug-in capability to enable security providers to extend the runtime capability and implement some of the higher level specifications in the Web Service Security stack. The plug-in points are exposed as Service Provider Programming Interfaces (SPI). For more information on these SPIs, see "Default implementations of the Web Services Security service provider programming interfaces" on page 263.

Web Services Security specification 1.0 development

The OASIS Web Services Security specification is based upon the following World Wide Web Consortium (W3C) specifications. Most of the W3C specifications are in the standard body recommended status.

- XML-Signature Syntax and Processing

  W3C recommendation, February 2002 (Also, IETF RFC 3275, March 2002)
- Canonical XML Version 1.0

  W3C recommendation, March 2001
- Exclusive XML Canonicalization Version 1.0

  W3C recommendation, July 2002
- XML-Signature XPath Filter Version 2.0

  W3C Recommendation, November 2002
- XML Encryption Syntax and Processing

  W3C Recommendation, December 2002
- Decryption Transform for XML Signature

  W3C Recommendation, December 2002

These specifications are supported in WebSphere Application Server in the context of Web Services Security. For example, you can sign a SOAP message by specifying the integrity option in the deployment descriptors. There is a client side application programming interface (API) that an application can use to enable Web Services Security for securing a SOAP message.

The OASIS Web Services Security Version 1.0 specification defines the enhancements that are used to provide message integrity and confidentiality. It also provides a general framework for associating the

security tokens with a SOAP message. The specification is designed to be extensible to support multiple security token formats. The particular security token usage is addressed with the security token profile.

Specification and profile support in WebSphere Application Server

OASIS is working on various profiles. For more information, see Organization for the Advancement of Structured Information Standards Committees.

The following list includes of the published draft profiles and OASIS Web Services Security technical committee work in progress.

WebSphere Application Server does not support these profiles:
- Web Services Security: SAML token profile 1.0
- Web Services Security: Rights Expression Language (REL) token profile 1.0
- Web Services Security: SOAP Messages with Attachments (SwA) profile 1.0

Note: Support for Web Services Security draft 13 and Username token profile draft 2 is deprecated in WebSphere Application Server 5.0.2, 5.1.0 and 5.1.1. For migration information, see "Migrating JAX-RPC Web Services Security applications to Version 8.5 applications" on page 363.

The wire format of the SOAP message with Web Services Security in Web Services Security Version 1.0 has changed and is not compatible with previous drafts of the OASIS Web Services Security specification. Interoperability between OASIS Web Services Security Version 1.0 and previous Web Services Security drafts is not supported. However, it is possible to run an application that is based on Web Services Security draft 13 on WebSphere Application Server Version 6 and later. The application can interoperate with an application that is based on Web Services Security draft 13 on WebSphere Application Server Version 5.0.2, 5.1 or 5.1.1.

WebSphere Application Server supports both the OASIS Web Services Security draft 13 and the OASIS Web Services Security 1.0 specification. But in WebSphere Application Server Version 6 and later, the support of OASIS Web Services Security draft 13 is deprecated. However, applications that were developed using OASIS Web Services Security draft 13 on WebSphere Application Server 5.0.2, 5.1.0 and 5.1.1 can run on WebSphere Application Server Version 6 and later. OASIS Web Services Security Version 1.0 support is available only for Java Platform, Enterprise Edition (Java EE) Version 1.4 and later applications. The configuration format for the deployment descriptor and the binding is different from previous versions of WebSphere Application Server. You must migrate the existing applications to Java EE 1.4 and migrate the Web Services Security configuration to the WebSphere Application Server Version 6 format.

Other Web Services Security specifications development

The most recently updated versions of the following OASIS Web Services Security specifications are supported in WebSphere Application Server in the context of Web Services Security:
- WS-Trust Version 1.3

  The Web Services Trust Language (WS-Trust) uses the secure messaging mechanisms of Web Services Security to define additional primitives and extensions for the issuance, exchange and validation of security tokens. WS-Trust enables the issuance and dissemination of credentials within different trust domains. This specification defines ways to establish, assess the presence of, and broker trust relationships.
- WS-SecureConversation Version 1.3

  The Web Services Secure Conversation Language (WS-SecureConversation) is built on the WS-Security and WS-Policy models to provide secure communication between services. WS-Security focuses on the message authentication model but not a security context, and thus is subject several forms of security attacks. This specification defines mechanisms for establishing and sharing security

contexts, and deriving keys from security contexts, to enable a secure conversation. By using the SOAP extensibility model, modular SOAP-based specifications are designed to be composed with each other to provide a rich messaging environment.

- WS-SecurityPolicy Version 1.2

  Web Services Security Policy (WS-Policy) provides a general purpose model and syntax to describe and communicate the policies of a web service. WS-Policy assertions express the capabilities and constraints of a particular web service. WS-PolicyAttachments defines several methods for associating the WS-Policy expressions with web services (such as WSDL). The Web Services Security specifications have been updated following the re-publication of WS-Security Policy in July 2005, to reflect the constraints and capabilities of web services that are using WS-Security, WSTrust and WS-SecureConversation. WS-ReliableMessaging Policy has also been re-published in 2005 to express the capabilities and constraints of web services implementing WS-ReliableMessaging.

Web Services Interoperability Organization (WS-I) activities

Web Services Interoperability Organization (WS-I) is an open industry effort to promote web services interoperability across vendors, platforms, programming languages and applications. The organization is a consortium of companies across many industries including IBM, Microsoft, Oracle, Sun, Novell, VeriSign, and Daimler Chrysler. WS-I began working on the basic security profile (BSP) in the spring of 2003. BSP consists of a set of non-proprietary web services specifications that clarifies and amplifies those specifications to promote Web Services Security interoperability across different vendor implementations. As of June 2004, BSP is a public draft. For more information, see the Web Services Interoperability Organization web page.

Specifically, see Basic Security Profile Version 1.0 for details about the BSP. WebSphere Application Server supports compliance with the BSP draft, but Web Services Security does not support the BSP Version 1.1 draft. See "Basic Security Profile compliance tips" on page 295 for the details to configure your application in compliance with the BSP draft.

Web Services Security configuration considerations:

To secure web services for WebSphere Application Server, you must specify several different configurations. Although there is not a specific sequence in which you must specify these different configurations, some configurations reference other configurations.

best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

You can configure Web Services Security on the application level, server level, and the cell level. The following table shows an example of the relationships between each of the configurations that apply to just the application, to an entire server, or to the entire cell. However, the requirements for the bindings depend upon the deployment descriptor. Some binding information depends upon other information in the binding or server and cell-level configuration. Within the table, the configurations in the Referenced configurations column are referenced by the configuration listed in the Configuration name column. For example, the token generator on the application-level for the request generator references the collection certificate store, the nonce, time stamp, and callback handler configurations.

Table 31. The relationship between the configurations.. Use the table to determine the mapping between the configurations and the level of Web Services Security.

Configuration level	Configuration name	Referenced configurations
Application-level request generator	Token generator	• Collection certificate store • Nonce • Timestamp • Callback handler
Application-level request generator	Key information	• Key locator • Key name • Token
Application-level request generator	Signing information	• Key information
Application-level request generator	Encryption information	• Key information
Application-level request consumer	Token consumer	• Trust anchor • Collection certificate store • Trusted ID evaluators • Java Authentication and Authorization Service (JAAS) configuration
Application-level request consumer	Key information	• Key locator • Token
Application-level request consumer	Signing information	• Key information
Application-level request consumer	Encryption information	• Key information
Application-level response generator	Token generator	• Collection certificate store • Callback handler
Application-level response generator	Key information	• Key locator • Token
Application-level response generator	Signing information	• Key information
Application-level response generator	Encryption information	• Key information
Application-level response consumer	Token consumer	• Trust anchor • Collection certificate store • JAAS configuration
Application-level response consumer	Key information	• Key locator • Key name • Token
Application-level response consumer	Signing information	• Key information
Application-level response consumer	Encryption information	• Key information
Server-level default generator bindings	Token generator	• Collection certificate store • Callback handler
Server-level default generator bindings	Key information	• Key locator • Token
Server-level default generator bindings	Signing information	• Key information
Server-level default generator bindings	Encryption information	• Key information
Server-level default consumer bindings	Token consumer	• Trust anchor • Collection certificate store • Trusted ID evaluator • JAAS configuration
Server-level default consumer bindings	Key information	• Key locator • Token
Server-level default consumer bindings	Signing information	• Key information

Table 31. The relationship between the configurations. (continued). Use the table to determine the mapping between the configurations and the level of Web Services Security.

Configuration level	Configuration name	Referenced configurations
Server-level default consumer bindings	Encryption information	• Key information
Cell-level default generator bindings	Token generator	• Collection certificate store • Callback handler
Cell-level default generator bindings	Key information	• Key locator • Token
Cell-level default generator bindings	Signing information	• Key information
Cell-level default generator bindings	Encryption information	• Key information
Cell-level default consumer bindings	Token consumer	• Trust anchor • Collection certificate store • Trusted ID evaluator • JAAS configuration
Cell-level default consumer bindings	Key information	• Key locator • Token
Cell-level default consumer bindings	Signing information	• Key information
Cell-level default consumer bindings	Encryption information	• Key information

When multiple applications will use the same binding information, consider configuring the binding information on the server or cell level. For example, you might have a global key locator configuration that is used by multiple applications. Configuration information for the application-level precedes similar configuration information on the server-level and the cell level.

Default bindings and runtime properties for Web Services Security:

Use this page to configure the settings for nonce on the server level and to manage the default bindings for the signing information, encryption information, key information, token generators, token consumers, key locators, collection certificate store, trust anchors, trusted ID evaluators, algorithm mappings, and login mappings.

Displayed options and the panel title depend on your server configuration and version.

To view this administrative console page for the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

Read the web services documentation before you begin defining the default bindings for Web Services Security.

Nonce is a unique cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens.

In WebSphere Application Server and WebSphere Application Server, Express, you must specify values for the Nonce cache timeout, Nonce maximum age, and Nonce clock skew fields for the server level.

Nonce cache timeout:

Specifies the timeout value, in seconds, for the nonce cached on the server. Nonce is a randomly generated value.

The Nonce cache timeout field is not required on the server level, but it is required on the cell level. To specify a value for the field on the cell level, click **Security** > **JAX-WS and JAX-RPC security runtime**.

If you make changes to the value for the Nonce cache timeout field, you must restart the application server for the changes to take effect.

Information	Value
Default	600 seconds
Minimum	300 seconds

Nonce maximum age:

Specifies the default time, in seconds, before the nonce timestamp expires. Nonce is a randomly generated value.

The maximum value cannot exceed the number of seconds that is specified in the Nonce cache timeout field for the server level.

The Nonce maximum age field is not required on the server level, but it is required on the cell level. The value set for this Nonce maximum age field on the server level must not exceed the value for the Nonce maximum age field on the cell level. To specify a value for the Nonce maximum age field on the cell level, click **Security** > **JAX-WS and JAX-RPC security runtime**.

Information	Value
Default	300 seconds
Range	300 to the value that is specified, in seconds, in the Nonce cache timeout field.

Nonce clock skew:

Specifies the default clock skew value, in seconds, to consider when the application server checks the timeliness of the message. Nonce is a randomly generated value.

The maximum value cannot exceed the number of seconds that is specified in the Nonce maximum age field.

The Nonce clock skew field is not required on the server level, but it is required on the cell level. To specify a value for the Nonce clock skew field on the cell level, click **Security** > **JAX-WS and JAX-RPC security runtime**.

Information	Value
Default	0 seconds
Range	0 to the value that is specified, in seconds, in the Nonce maximum age field.

Enable cryptographic operations on hardware device:

Enables cryptographic operations on hardware devices. Enabling this feature might improve the performance, depending on the hardware device.

Cryptographic hardware configuration name:

Specifies the name of the hardware device configuration name that is defined in the keystore settings in the secure communications.

This value is necessary only if **Hardware acceleration** has been selected.

Custom properties:

The linked Properties panel specifies additional properties for the security runtime configuration.

Web Services Security provides message integrity, confidentiality, and authentication:

OASIS Web Services Security (WS-Security) is a flexible standard that is used to secure web services at the message level within multiple security models. You can secure SOAP messages through XML digital signature, confidentiality through XML encryption, and credential propagation through security tokens.

The WS-Security specification defines the core facilities for protecting the integrity and confidentiality of a message and provides mechanisms for associating security-related claims with the message. Message-level security, or securing web services at the message level, addresses the same security requirements as for traditional web security. These security requirements include: identity, authentication, authorization, integrity, confidentiality, nonrepudiation, basic message exchange, and so forth. Both traditional web and message-level security share many of the same mechanisms for handling security, including digital certificates, encryption, and digital signatures. While HTTPS and Secure Sockets Layer (SSL) transport-level technology may be used for securing web services, some security scenarios are addressed more effectively by message-level security.

Traditional web security mechanisms, such as HTTPS, might be insufficient to manage the security requirements of all web service scenarios. For example, when an application sends a document with JAX-RPC using HTTPS, the message is secured only for the HTTPS connection, meaning during the transport of the document between the service requester (the client) and the service. However, the application might require that the document data be secured beyond the HTTPS connection, or even beyond the transport layer. By securing web services at the message level, message-level security is capable of meeting these expanded requirements.

Message-level security applies to XML documents that are sent as SOAP messages. Message-level security makes security part of the message itself by embedding all required security information in the SOAP header of a message. In addition, message-level security can apply security mechanisms, such as encryption and digital signature, to the data in the message itself.

With message-level security, the SOAP message itself either contains the information needed to secure the message or it contains information about where to get that information to handle security needs. The SOAP message also contains information relevant to the protocols and procedures for processing the specified message-level security. However, message-level security is not tied to any particular transport mechanism. Because the security information is part of the message, it is independent of a transport protocol, such as HTTPS.

The client adds to the SOAP message header security information that applies to that particular message. When the message is received, the web service endpoint, using the security information in the header, verifies the secured message and validates it against the policy. For example, the service endpoint might verify the message signature and check that the message has not been tampered with. It is possible to add signature and encryption information to the SOAP message headers, as well as other information such as security tokens for identity (for example, an X.509 certificate) that are bound to the SOAP message content.

For WebSphere Application Server Versions 6 and later, Web Services Security can be applied as transport-level security and as message-level security. You can architect highly secure client and server

designs by using these security mechanisms. Transport-level security refers to securing the connection between a client application and a web service with Secure Sockets Layer (SSL).

You can apply various scenarios of Web Services Security according to the characteristics of each web service application. You have choices of how to protect your information when using Web Services Security. The authentication mechanism, integrity, and confidentiality can be applied at the message level and at the transport level. When message-level security is applied, you can protect the SOAP message with a security token, digital signature, and encryption.

Without Web Services Security, the SOAP message is sent in clear text, and personal information such as a user ID or an account number is not protected. Without applying Web Services Security, there is only a SOAP body under the SOAP envelope in the SOAP message. By applying features from the WS-Security specification, the SOAP security header is inserted under the SOAP envelope in the SOAP message when the SOAP body is signed and encrypted.

To maintain the integrity or confidentiality of the message, digital signatures and encryption are typically applied.

- *Confidentiality* specifies the confidentiality constraints that are applied to generated messages. This includes specifying which message parts within the generated message must be encrypted, and the message parts to attach encrypted Nonce and time stamp elements to.
- *Integrity* is provided by applying a digital signature to a SOAP message. Confidentiality is applied by SOAP message encryption. Multiple signatures and encryptions are supported. In addition, both signing and encryption can be applied to the same parts, such as the SOAP body.

You can add an authentication mechanism by inserting various types of security tokens, such as the Username token (<UsernameToken> element). When the Username token is received by the web service server, the user name and password are extracted and verified. Only when the user name and password combination is valid, will the message be accepted and processed at the server. Using the Username token is just one of the ways of implementing authentication. This mechanism is also known as *basic authentication*.

In addition to digital signatures, encryption, and basic authentication, other forms of authentication include identity assertion, LTPA tokens, Kerberos tokens, and custom tokens. These other forms of authentication are also extensions of WebSphere Application Server. You can configure these authentication mechanisms using the assembly tools to implement authentication.

With updates to Web Services Security in the Version 1.1 specification, it is possible to layer additional functionality in addition to these basic mechanisms. Some Version 1.1 mechanisms are extensions of WebSphere Application Server, such as signature confirmation and the encrypted header. The security token profiles that are supported by WebSphere Application Server include the Username token profile, the X.509 token profile, and the Kerberos profile. In this case, when the message is received, the web service endpoint, using the security information in the header, applies the appropriate security mechanisms to the message. For example, the service endpoint might add signature and encryption information to the SOAP message headers, as well as other information, such as security tokens, that are bound to the SOAP message content. You can implement these new mechanisms by using a policy set.

WS-SecureConversation was introduced in WebSphere Application Server Version 6.1 with the Feature Pack for Web Services. Secure Conversation uses a session key to protect SOAP messages more efficiently, particularly when multiple SOAP messages are transmitted in a session.

Other enhancements include:

- The Kerberos token, which is used for both authentication and for subsequent message protection.
- Dynamic policy, which allows the client to retrieve the provider policy through a WSDL request, or using Web Services MetadataExhange (WS-MEX), to simplify web services client deployment.

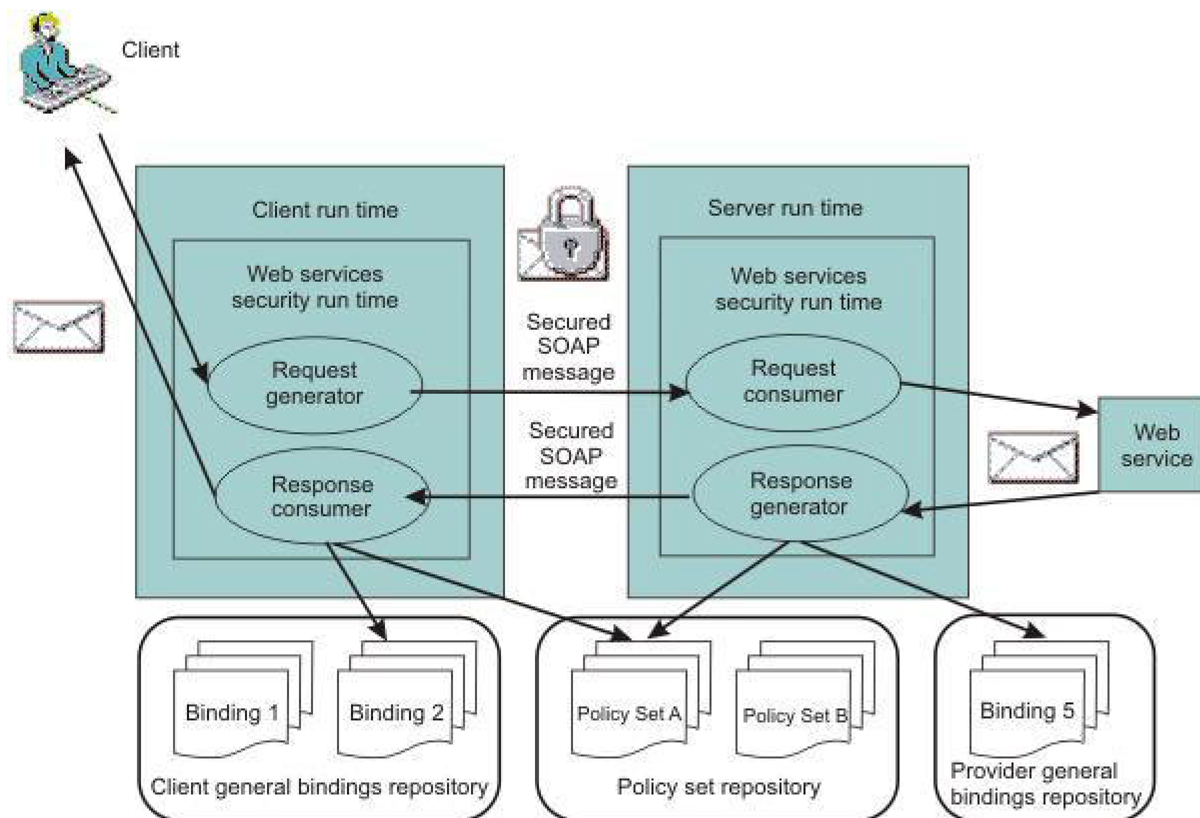
High-level architecture for Web Services Security:

The Web Services Security policy is specified in the IBM extension of the web services deployment descriptors when using the JAX-RPC programming model, and in policy sets when using the JAX-WS programming model. A stand-alone JAX-WS client application may specify Web Services Security policy programmatically. Binding data that supports the Web Services Security policy are stored in the IBM extension of the web services deployment descriptors for both the JAX-RPC and JAX-WS programming models. The Web Services Security run time enforces the security assertions that are specified in the policy document, or in the application program, in that order.

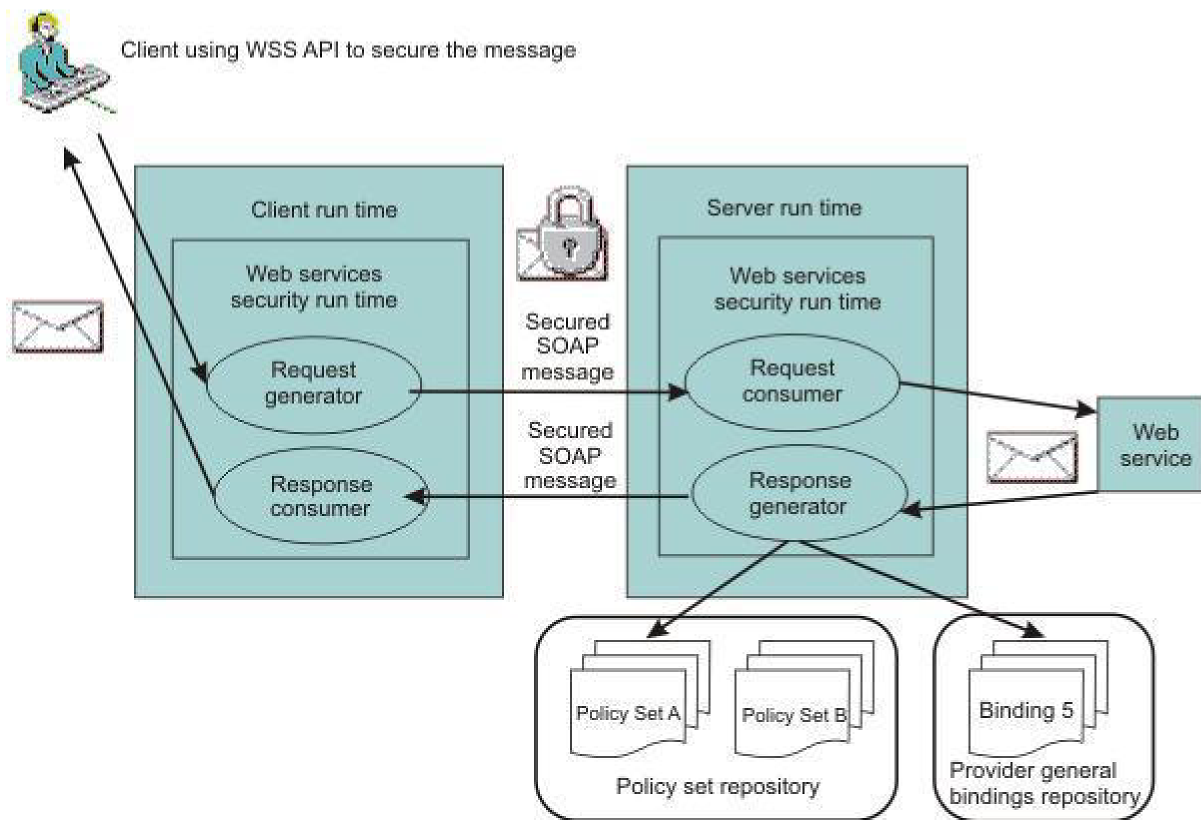
best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

WebSphere Application Server uses the Java Platform, Enterprise Edition (Java EE) Version 1.4 or later web services deployment model to implement Web Services Security. One of the advantages of deployment model is that you can define the Web Services Security requirements outside of the application business logic. With the separation of roles, the application developer can focus on the business logic and the security expert can specify the security requirement.

The following figure shows the high-level architecture model that is used to secure web services in WebSphere Application Server:



The WSS API can also be used to secure the message, as illustrated later in this section:



There are two sets of configurations on both the client side and the server side:

Request generator
This client-side configuration defines the Web Services Security requirements for the outgoing SOAP message request. These requirements might involve generating a SOAP message request that uses a digital signature, incorporates encryption, and attaches security tokens. In WebSphere Application Server Versions 5.0.2, 5.1, and 5.1.1, the request generator was known as the *request sender*.

Request consumer
This server-side configuration defines the Web Services Security requirements for the incoming SOAP message request. These requirements might involve verifying that the required integrity parts are digitally signed; verifying the digital signature; verifying that the required confidential parts were encrypted by the request generator; decrypting the required confidential parts; validating the security tokens, and verifying that the security context is set up with the appropriate identity. In WebSphere Application Server Versions 5.0.2, 5.1, and 5.1.1, the request consumer was known as the *request receiver*.

Response generator
This server-side configuration defines the Web Services Security requirements for the outgoing SOAP message response. These requirements might involve generating the SOAP message response with Web Services Security; including digital signature; and encrypting and attaching the security tokens, if necessary. In WebSphere Application Server Versions 5.0.2, 5.1, and 5.1.1, the response generator was known as the *response sender*.

Response consumer
This client-side configuration defines the Web Services Security requirements for the incoming SOAP response. The requirements might involve verifying that the integrity parts are signed and

the signature is verified; verifying that the required confidential parts are encrypted and that the parts are decrypted; and validating the security tokens. In WebSphere Application Server Versions 5.0.2, 5.1, and 5.1.1, the response consumer was known as the *response receiver*.

WebSphere Application Server does not include security policy negotiation or exchange between the client and server. This security policy negotiation, as defined by the WS-Policy, WS-PolicyAssertion, and WS-SecurityPolicy specifications, are not supported in WebSphere Application Server.

Note: The Web Services Security requirements that are defined in the request generator must match the request consumer. The requirements that are defined in the response generator must match the response consumer. Otherwise, the request or response is rejected because the Web Services Security constraints cannot be met by the request consumer and response consumer.

The format of the Web Services Security deployment descriptors and bindings are IBM proprietary. However, the following tools are available to edit the deployment descriptors and bindings:

IBM assembly tools
Use IBM assembly tools to edit the Web Services Security deployment descriptor and binding. Use the tools to assemble both web and Enterprise JavaBeans (EJB) modules. For more information, read about assembly tools.

WebSphere Application Server Administrative Console
Use this tool to edit the Web Services Security binding of a deployed application.

Security model mixture:

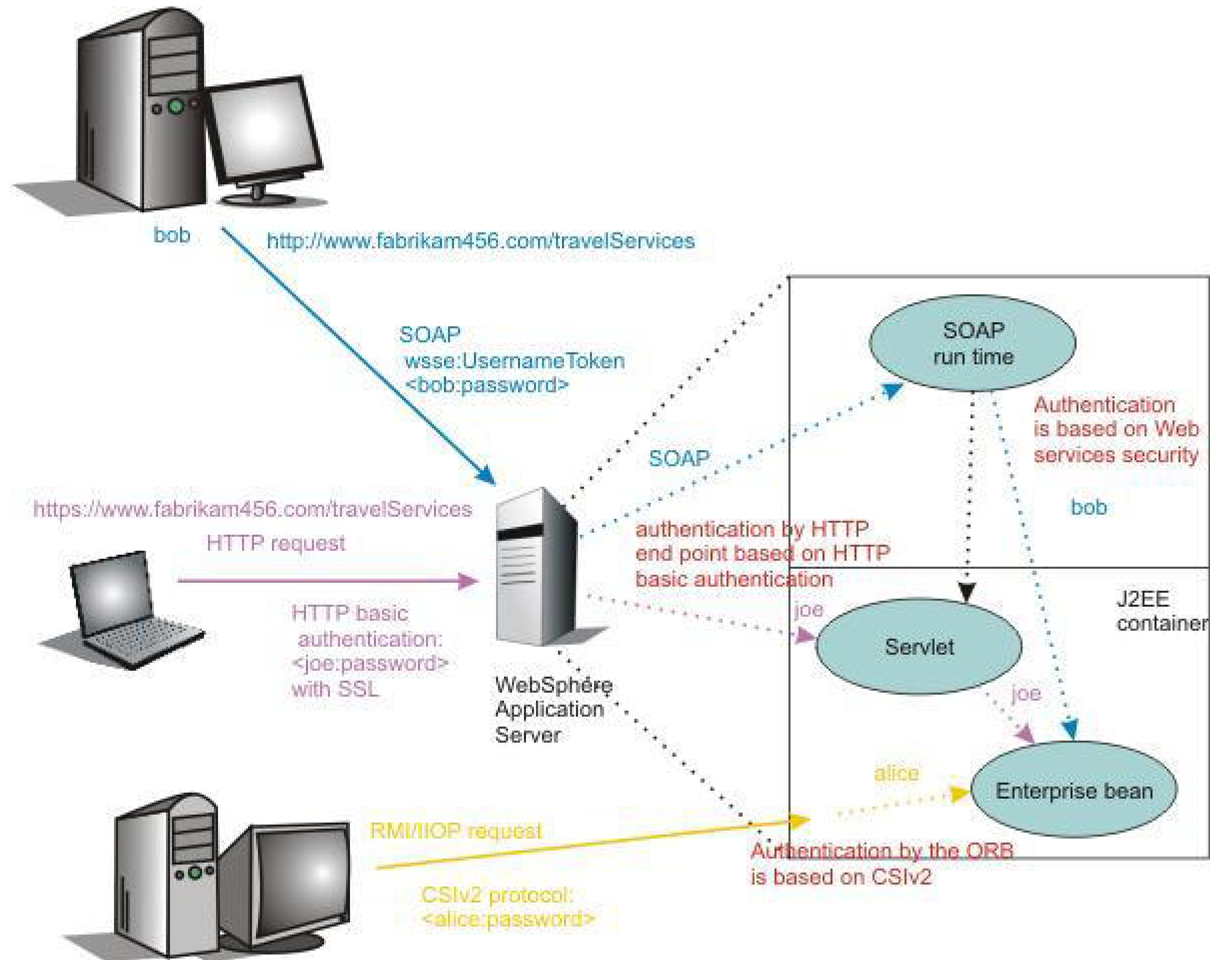
There can be multiple protocols and channels in the WebSphere Application Server Version 6 and later programming environments. Each of these applications serve different business needs.

For example, you might access:
- A Web-based application through the HTTP transport such as a servlet, JavaServer Pages (JSP) file, HTML and so on.
- An enterprise application through the Remote Method Invocation (RMI) over the Internet Inter-ORB (RMI/IIOP) protocol.
- A web service application through the SOAP over HTTP, SOAP over the Java Message Service (JMS), or SOAP over the RMI/IIOP protocol.

More importantly, web services are often implemented as servlets with a Enterprise JavaBeans (EJB) file. Therefore, you can mix and match the Web Services Security model with the Java Platform, Enterprise Edition (Java EE) security model for web and EJB components. It is intended that web service security complement the Java EE role-based security and the security run time for WebSphere Application Server Version 6 and later.

Web Services Security also can take advantage of the security features in Java EE and the security run time for WebSphere Application Server Version 6 and later. For example, Web Services Security can use the following security features to provide an end-to-end security deployment:
- Use the local OS, Lightweight Directory Access Protocol (LDAP), and custom user registries for authenticating the username token
- Propagate the Lightweight Third Party Authentication (LTPA) security token in the SOAP message
- Use identity assertion
- Use a trust association interceptor (TAI)
- Enable security attribute propagation
- Use Java EE role-based authorization
- Use a Java Authorization Contract for Containers (JACC) authorization provider, such as Tivoli Access Manager

The following figure shows that different security protocols are used to send authentication information to the application server. For a web service, you might use either HTTP basic authentication with Secure Sockets Layer (SSL) or a Web Services Security username token with signing and encryption. In the following figure, when identity *bob* from Web Services Security is authenticated and set as the caller identity of the SOAP message request, the Java EE Enterprise JavaBeans container performs authorization using *bob* before the call is dispatched to the service implementation, which, in this case, is the enterprise bean.



You can secure a web service using the transport layer security. For example, when you are using SOAP over HTTP, HTTPS can be used to secure the web service. However, transport layer security provides point-to-point security only. This layer of security might be adequate for certain scenarios. However, when the SOAP message must travel through intermediary servers (multi-hop) before it is consumed by the target endpoint, you might use SOAP over the Java Message Service (JMS). The usage scenarios and security requirements dictate how to secure web services. The requirements depend upon the operating environment and the business needs. However, one key advantage of using Web Services Security is that it is transport layer independent; the same Web Services Security constraints can be used for SOAP over HTTP, SOAP over JMS, or SOAP over RMI/IIOP.

Overview of platform configuration and bindings:

The Web Services Security policy is specified in the IBM extension of the web services deployment descriptors when using the JAX-RPC programming model, and in policy sets when using the JAX-WS programming model. Binding information to support the Web Services Security policy is stored in the IBM extension of the web services deployment descriptors for both the JAX-RPC and JAX-WS programming models.

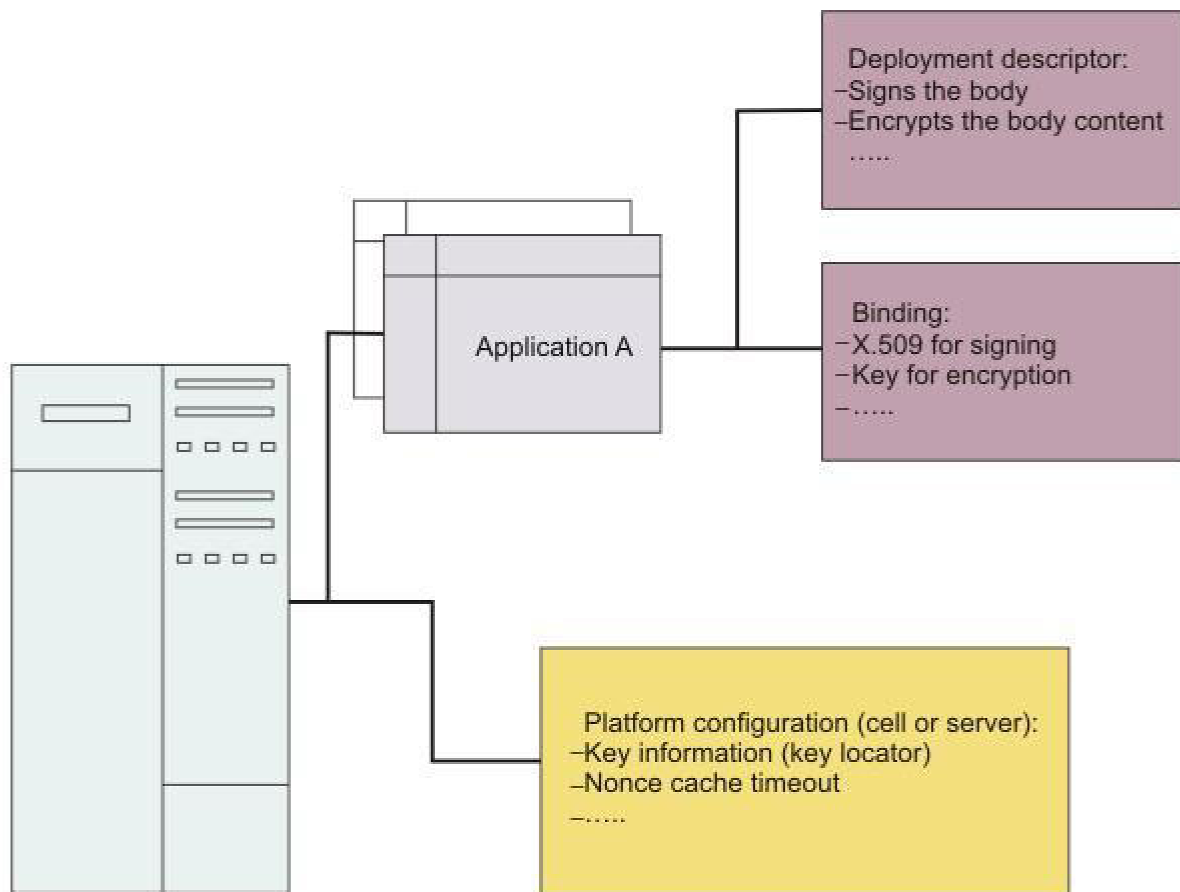
best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

Due to the complexity of these files, it is not recommended that you edit the deployment descriptor and binding files manually with a text editor because they might cause errors. It is recommended, however, that you use the tools provided by IBM to configure the Web Services Security constraints for an application. These tools are the WebSphere Application Server administrative console, or an assembly tool. For more information about IBM assembly tools, see the assembly tools information.

You can use the policy set function of the WebSphere Application Server to simplify your web services configuration because policy sets group security and other web services settings into reusable units. Policy sets are assertions about how quality of services is defined. A policy set incorporates policy types, and their settings.

In addition to the application deployment descriptor and binding files, WebSphere Application Server Versions 6 and later have a cell level and a server level configuration. These configurations are global for all applications. Because WebSphere Application Server Version 6 and later support 5.x applications, some of the configurations are valid for Version 5.x applications only and some are valid for Version 6 and later applications only.

The following figure represents the relationship of the application deployment descriptor and binding files to the cell (WebSphere Application Server, Network Deployment only) or server level configuration.



Platform configuration

The following options are available in the administrative console:

Nonce cache timeout
This option, which is found on the cell level (WebSphere Application Server, Network Deployment only) and server level, specifies the cache timeout value for a nonce in seconds.

Nonce maximum age
This option, which is found on the cell level (WebSphere Application Server, Network Deployment only) and server level, specifies the default life span for the nonce in seconds.

Nonce clock skew
This option, which is found on the cell level (WebSphere Application Server, Network Deployment only) and server level, specifies the default clock skew to account for network delay, processing delay, and so on. It is used to calculate when the nonce expires. Its unit of measurement is seconds.

Distribute nonce caching
This feature enables you to distribute the cache for the nonce to different servers in a cluster. It is available for WebSphere Application Server Version 6.0.x and later.

The following features can be referenced in the application binding:

Key locator
: This feature specifies how the keys are retrieved for signing, encryption, and decryption. The implementation classes for the key locator are different in WebSphere Application Server Versions 6 and later and Version 5.*x*.

Collection certificate store
: This feature specifies the certificate store for certificate path validation. It is typically used for validating X.509 tokens during signature verification or constructing the X.509 token with a certificate revocation list that is encoded in the PKCS#7 format. The certificate revocation list is supported for WebSphere Application Server Version 6.x and later applications only.

Trust anchors
: This feature specifies the trust level for the signer certificate and is typically used in the X.509 token validation during signature verification.

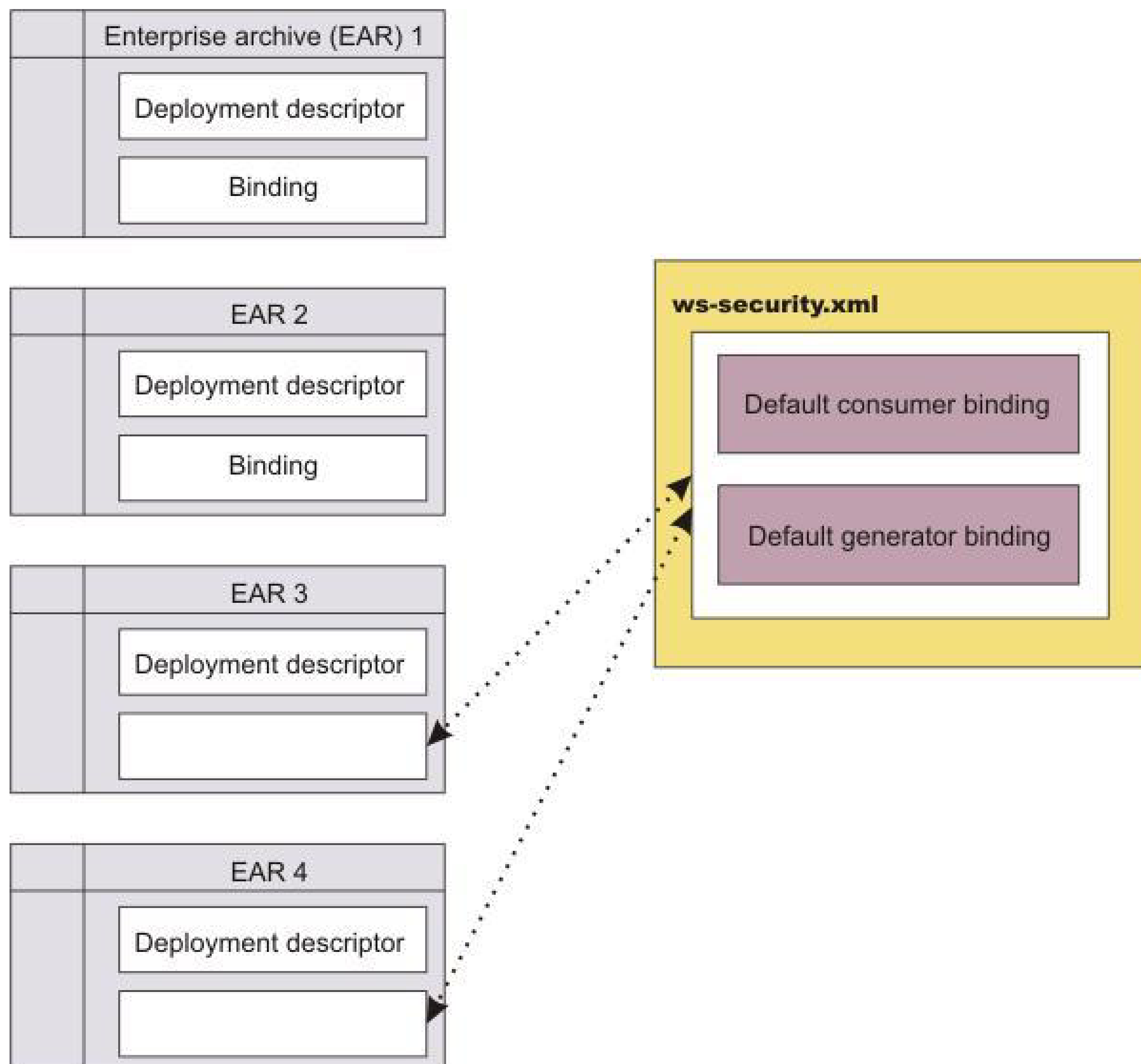
Trusted ID evaluators
: This feature specifies how to verify the trust level for the identity. The feature is used with identity assertion.

Login mappings
: This feature specifies the login configuration binding to the authentication methods. This feature is used by WebSphere Application Server Version 5.*x* applications only and it is deprecated.

Default bindings

The configuration of the default cell level and default server level bindings has changed in WebSphere Application Server. Previously, you could configure only one set of default bindings for the cell, and optionally configure one set of default bindings for each server. In version 7.0 and later, you can configure one or more general provider bindings and one or more general client bindings. However, only one general provider binding and one general client binding can be designated as the default.

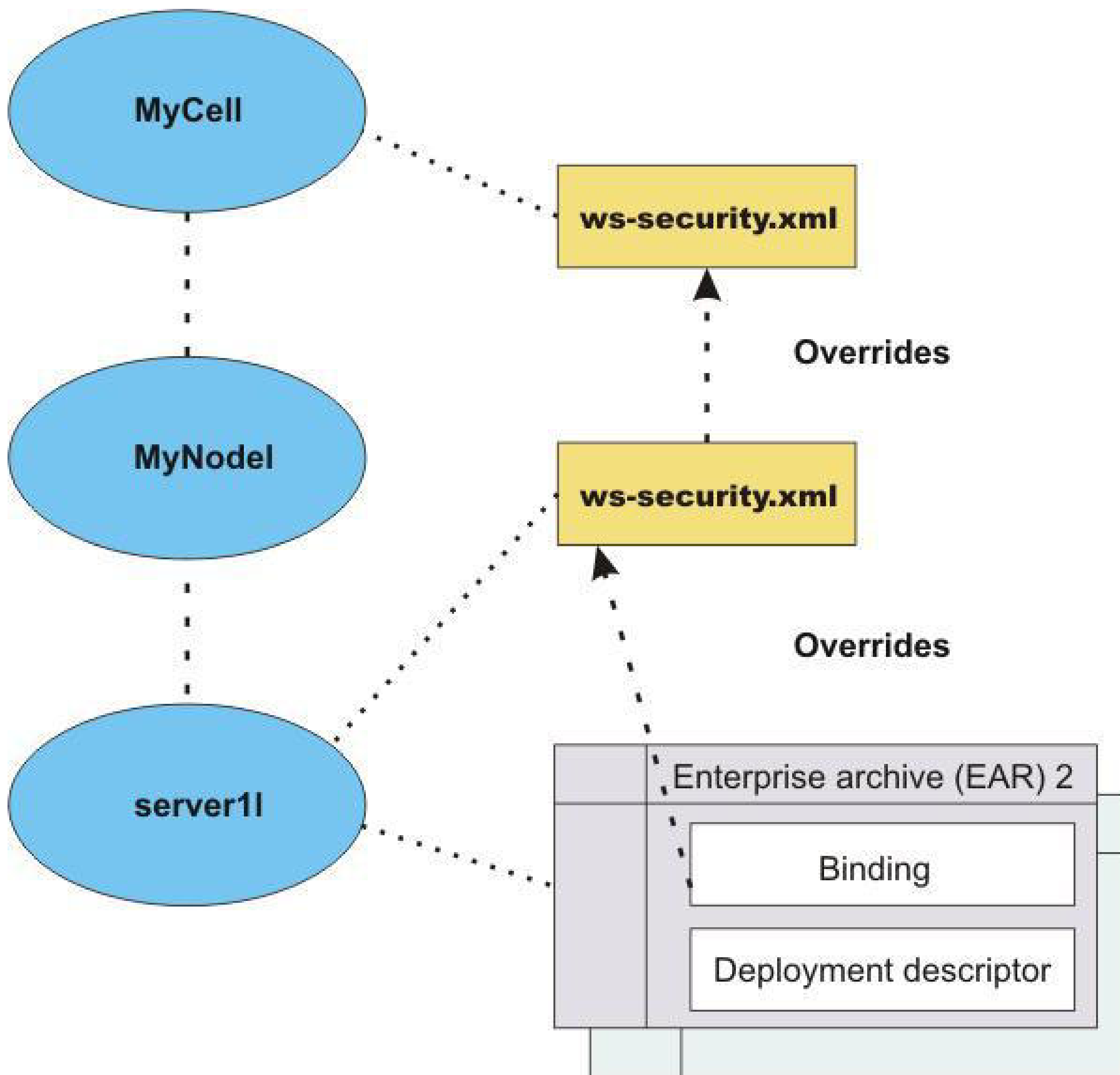
The following figure shows the relationship between the application enterprise archive (EAR) file and the `ws-security.xml` file.



Applications EAR 1 and EAR 2 have specific bindings in the application binding file. However, applications EAR 3 and EAR 4 do not have a binding in the application binding file; it must be referenced to use the default bindings defined in the `ws-security.xml` file. The configuration is resolved by nearest configuration in the hierarchy. For example, there might be three key locators named `mykeylocator` that is defined in the application binding file, the server level, and the cell level.

If `mykeylocator` is referenced in the application binding, then the key locator that is defined in the application binding is used. The visibility scope of the data depends upon where the data is defined. If the data is defined in the application binding, then its visibility is scoped to that particular application. If the data is defined on the server level, then the visibility scope is all of the applications deployed on that server. If the data is defined on the cell level, then the visibility scope is all of the applications deployed on servers in the cell. In general, if data is not meant to be shared by other applications, define the configuration in the application binding level.

The following figure shows the relationship of the bindings on the application, server, and cell (WebSphere Application Server, Network Deployment only) levels.



General bindings

General bindings are used as the default bindings at the cell level or server level. The general bindings that are shipped with WebSphere Application Server are initially set as the default bindings, but you can choose a different binding as the default, or change the level of binding that should be used as the default, for example, from cell level binding to server level binding.

In version 7.0 and later, there are two types of bindings: application specific bindings, and general bindings. Both types of bindings are supported for WS-Security policy sets. General bindings can be shared across multiple applications and for trust service attachments. There are two types of general bindings: one for service providers and one for service clients. Multiple general bindings can be defined for the provider and also for the client.

Keys:

Use keys for XML digital signature and encryption.

There are two predominant kinds of keys used in the current Web Services Security implementation:

- Public key - such as Rivest Shamir Adleman (RSA) encryption and Digital Signature Algorithm (DSA) encryption
- Secret key - such as triple-strength DES (3DES) encryption

In public key-based signature, a message is signed using the sender private key and is verified using the sender public key. In public key-based encryption, a message is encrypted using the receiver public key and is decrypted using the receiver private key. In secret key-based signature and encryption, the same key is used by both parties.

While the current implementation of Web Services Security can support both kinds of keys, the format of the message differs slightly between public key-based encryption and secret key-based encryption.

Key locator:

A key locator is an abstraction of the mechanism that retrieves the key for digital signature and encryption. The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side and to validate (authenticate) the security token on the consumer side.

Retrieve keys from one of the following sources, depending upon your implementation:
- Java keystore file
- Database
- Kerberos KDC server (WebSphere Application Server using JAX-WS only)
- Trust service can provide a security context token and key (WebSphere Application Server using JAX-WS only)

Key locators search for the key using some type of a clue. The following types of clues are supported:
- A string label of the key, which is explicitly passed through the application programming interface (API). The relationship between each key and its name (string label) is maintained inside the key locator.
- The implementation context of the key locator; explicit information is not passed to the key locator. A key locator determines the appropriate key according to the implementation context.

WebSphere Application Server Versions 6 and later support a secret key-based signature called HMAC-SHA1. If you use HMAC-SHA1, the SOAP message does not contain a binary security token. In this case, it is assumed that the key information within the message contains the key name that is used to specify the secret key within the keystore.

Because the key locators support the public key-based signature, the key for verification is embedded in the X.509 certificate as a `<BinarySecurityToken>` element in the incoming message. For example, key locators can obtain the identity of the caller from the context and can retrieve the public key of the caller for response encryption.

This section describes the usage scenarios for key locators.

Signing:

The name of the signing key is specified in the Web Services Security configuration. This value is passed to the key locator and the actual key is returned. The corresponding X.509 certificate also can be returned.

Verification:

By default, WebSphere Application Server Versions 6 and later supports the following types of key locators:

KeyStoreKeyLocator
Uses the keystore to retrieve the key that is used for digital signature and verification or encryption and decryption.

X509CertKeyLocator
Uses an X.509 certificate within a message to retrieve the key for verification or decryption.

SignerCertKeyLocator
Uses the X.509 certificate within the request message to retrieve the key that is used for encryption in the response message.

Encryption:

The name of the encryption key is specified in the Web Services Security configuration. This value is passed to the key locator and the actual key is returned. On the server side, you can use the SignerCertKeyLocator to retrieve the key for encryption in the response message from the X.509 certificate in the request message.

Decryption:

The Web Services Security specification recommends using the key identifier instead of the key name. However, while the algorithm for computing the identifier for the public keys is defined in Internet Engineering Task Force (IETF) Request for Comment (RFC) 3280, there is no agreed-upon algorithm for the secret keys. Therefore, the current implementation of Web Services Security uses the identifier only when public key-based encryption is performed. Otherwise, the ordinal key name is used.

When you use public key-based encryption, the value of the key identifier is embedded in the incoming encrypted message. Then, the Web Services Security implementation searches for all of the keys managed by the key locator and decrypts the message using the key whose identifier value matches the one in the message.

When you use secret key-based encryption, the value of the key name is embedded in the incoming encrypted message. The Web Services Security implementation asks the key locator for the key with the name that matches the name in the message and decrypts the message using the key.

Trust anchor:

A trust anchor specifies the key stores that contain trusted root certificates. These certificates are used to validate the X.509 certificate that is embedded in the SOAP message.

When using WebSphere Application Server with the JAX-RPC programming model, key stores are implemented with the following message points to validate the X.509 certificate that is used for digital signature or XML encryption:

- Request consumer, as defined in the `ibm-webservices-bnd.xmi` file.
- Response consumer, as defined in the `ibm-webservicesclient-bnd.xmi` file when a web service is acting as a client to another web service.

For WebSphere Application Server Version 7.0 and later, using JAX-WS, key stores are used by the following message points to validate the X.509 certificate that is used for digital signature or XML encryption:

- Request consumer, as defined in the inbound keys and certificates of the WS-Security bindings.
- Response consumer, as defined in the inbound keys and certificates of the WS-Security bindings when a web service is acting as a client to another web service.

Key stores are critical to the integrity of the digital signature validation. If the key stores are tampered with, the result of the digital signature verification is doubtful and compromised. Therefore, it is recommended that you secure the key stores. The binding configuration specified for the consumer must match the binding configuration for the generator.

The trust anchor is defined as java.security.cert.TrustAnchor in the Java CertPath application programming interface (API). The Java CertPath API uses the trust anchor and the certificate store to validate the incoming X.509 certificate that is embedded in the SOAP message. The Web Services Security implementation in WebSphere Application Server supports this trust anchor. In WebSphere Application Server, the trust anchor is represented as a Java key store object. The type, path, and password of the key store are passed to the implementation through the administrative console or by scripting.

Trusted ID evaluator:

A trusted ID evaluator is the mechanism that evaluates whether a given ID name is trusted.

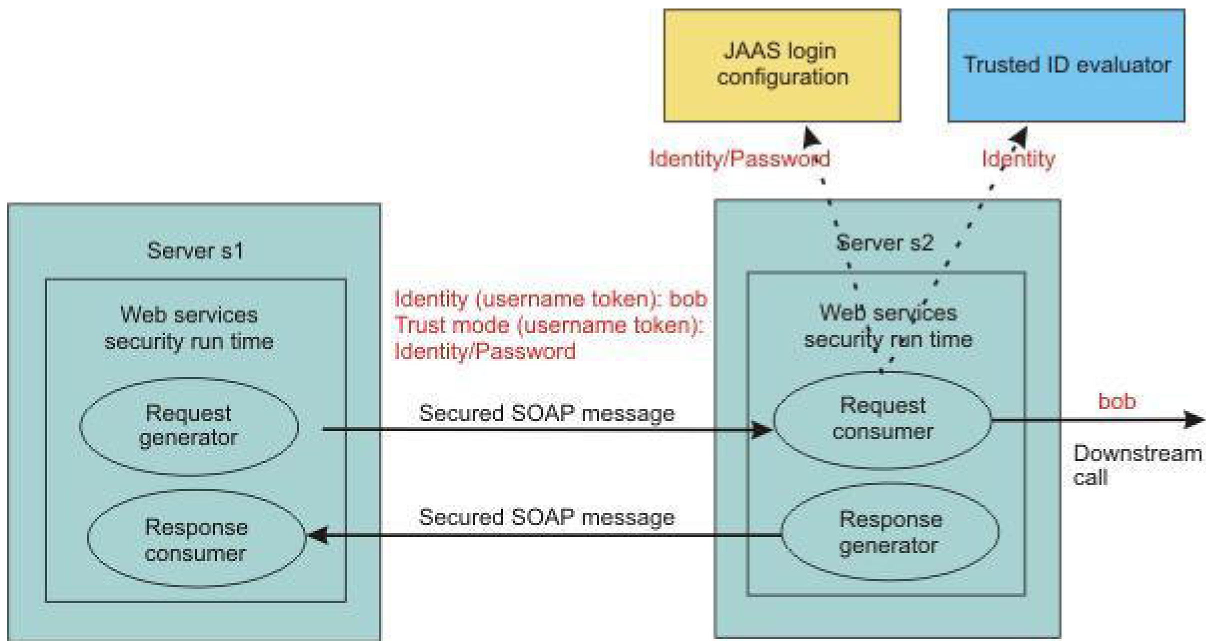
Using the trusted ID evaluator with the JAX-RPC programming model

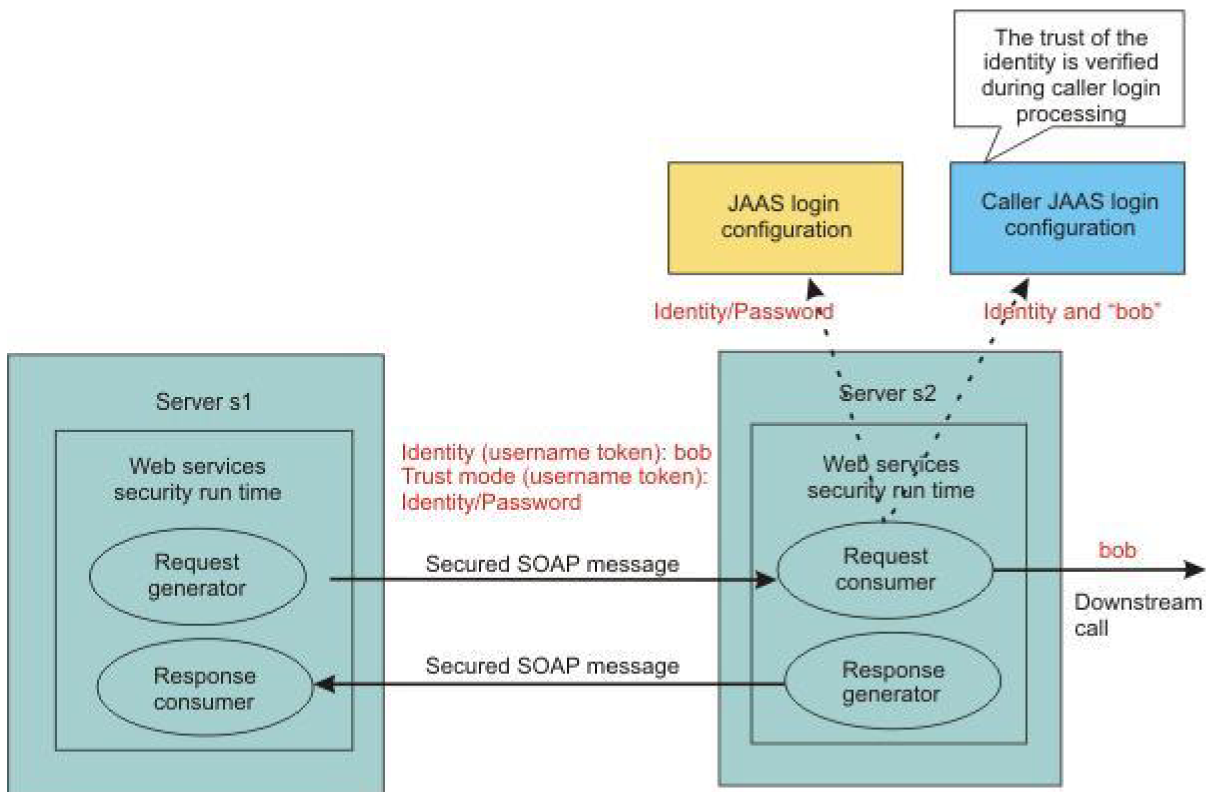
In the JAX-RPC programming model, the trusted ID evaluator, com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl, is an abstraction of the mechanism that evaluates whether a given ID name is trusted. There are two trust modes for validating the trust of the upstream server when using JAX-RPC:

Basic authentication (username token)
: The upstream server sends a username token with a user name and password to a downstream server. The consumer or receiver of the message authenticates the username token and validates the trust based upon the TrustedIDEvaluator implementation. The TrustedIDEvaluator implementation must implement the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator Java interface.

Signature
: The upstream server signs the message, which can be any message part such as the SOAP body. The upstream server sends the X.509 token to a downstream server. The consumer or receiver of the message verifies the signature and validates the X.509 token. The identity or the distinguished name from the X.509 token that is used in the digital signature is validated based on the TrustedIDEvaluator implementation. The TrustedIDEvaluator implementation must implement the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator Java interface. For the X.509 certificate, WebSphere Application Server uses the distinguished name in the certificate as a requester identity.

The following figures demonstrate the identity assertion trust process for both programming models:





In this figure, server `s1` is the upstream server and identity assertion is set up between server `s1` and server `s2`. The `s1` server authenticates the identity called *bob*. Server `s1` wants to send *bob* to the `s2` server with a password. The trust mode is an `s1` credential that contains the identity and a password. Server `s2` receives the request, authenticates the user using a Java Authentication and Authorization Service (JAAS) login module, and uses the trusted ID evaluator to determine whether to trust the identity. If the identity is trusted, *bob* is used as the caller that invokes the service. If authorization is required, *bob* is the identity that is used for authorization verification.

The identity can be asserted as the RunAs (invocation) identity of the current security context. For example, the web services gateway authenticates a requester using a secure method such as password authentication and then sends the requester identity only to a back-end server. You might also use identity assertion for interoperability with another Web Services Security implementation.

Depending upon the implementation of JAX-RPC, you can use various types of infrastructure to store a list of the trusted IDs, such as:
- Plain text file
- Database
- Lightweight Directory Access Protocol (LDAP) server

The trusted ID evaluator is typically used by the eventual receiver in a multi-hop environment. The Web Services Security implementation invokes the trusted ID evaluator and passes the identity name of the intermediary as a parameter. If the identity is evaluated and deemed trustworthy, the procedure continues. Otherwise, an exception is created and the procedure is stopped.

Using the trusted ID evaluator with the JAX-WS programming model

In the JAX-WS programming model, the same concepts are supported for the trusted ID evaluator, although the implementation is different. For the JAX-WS run time, use the administrative console to select the **Use identity assertion** option on the caller binding panel. This defines the trusted identity token type, and then defines a list of one or more trusted identities. The trusted ID evaluator validates the trusted identity token against the list of trusted identities. For more information about the list of trusted identities, read the topic Changing the order of the callers for a token or message part.

For WebSphere Application Server Version 6.1 and later, the Caller and TrustMethod elements are used to support the requestor login. The requestor sends a message to an intermediary, and the message is dispatched to the service. Based on the security information, the service performs a login for the requestor. In some cases, there are multiple security tokens, so the service has to decide which one to use. When the requestor ID is included as an ID assertion, the service can specify how to trust the intermediary. The following intermediary scenarios are supported:

<BasicAuth, null, null>
: The requestor username and password is used for authentication. In this case, authentication is performed with requestor properties, therefore a password is required for authentication.

<Signature, null, null>
: The requestor signature is used for authentication.

<IDAssertion, Username, null>
: The requestor username (without a password) is used to identify the requestor. The UsernameToken token is used as the ID assertion, therefore no password is required to accompany the username. In this case, the service trusts the intermediary unconditionally.

<IDAssertion, Username, Username>
: The requestor username (without a password) is used to identify the requestor, and the username and password of the intermediary is used to authenticate the intermediary. The UsernameToken token, when used to establish trusted identity, always requires a password because the purpose of the token is to establish trust between the intermediary and the service.

<IDAssertion, Username, X509>
: The requestor username (without a password) is used to identify the requestor, and the signature of the intermediary is used to authenticate the intermediary. In this case, the trusted identity for the signature of the intermediary must be established using an X.509 certificate.

<IDAssertion, X509, null>
: The identity of the requestor is established using an X.509 certificate. In this case, the X.509

certificate from the requestor does not provide a signature to prove possession of the certificate, and therefore the service trusts the intermediary unconditionally.

<IDAssertion, X509, Username>
The identity of the requestor is established using an X.509 certificate, and the username and password of the intermediary is used to authenticate the intermediary. The UsernameToken token, when used to establish trusted identity, always requires a password because the purpose of the token is to establish trust between the intermediary and the service.

<IDAssertion, X509, X509>
The identity of the requestor is established using an X.509 certificate, and the signature of intermediary is used to authenticate the intermediary.

Hardware cryptographic device support for Web Services Security:

In IBM WebSphere Application Server Version 6.1 or later, Web Services Security supports the use of cryptographic hardware devices. There are two ways in which to use hardware cryptographic devices with Web Services Security.

Enabling cryptographic operations on hardware devices

You can enable cryptographic operations on hardware devices. The keys that are used can be stored in a Java keystore file; it is not necessary to store them on the hardware device. The decision to use enable cryptographic operations on hardware devices is made at the server level only, not at the application level.

If cryptographic operations on hardware device is enabled, the Web Service Security run time first attempts to use the hardware device for cryptographic operations. If the attempt to use the hardware device fails or if the algorithm is not supported by the hardware device, the run time uses a software provider from the security providers list.

Enabling this feature might improve the performance, depending on the hardware device. For more information on how to enable cryptographic operations on hardware devices, see “Configuring hardware cryptographic devices for Web Services Security” on page 980.

Secure keys

Cryptographic keys can be stored on the hardware cryptographic device and never leave the device. These secure keys are confined to the hardware cryptographic device for security considerations rather than performance considerations. The option to select whether to use keys that are stored in a hardware cryptographic device or a Java keystore file can be made at the application level.

If the keystore reference is specified to be a hardware device configuration, the Web Services Security run time first attempts to obtain the cryptographic algorithm from the hardware device. If the algorithm is not supported or fails, the run time uses a software provider from the security providers list.

See further information about how to enable secure keys, see “Enabling cryptographic keys stored in hardware devices in Web Services Security” on page 982.

Limitations

The hardware cryptographic device support for Web Services Security currently has the following limitations:

- There is no support for a web services client running as a Java Platform, Enterprise Edition (Java EE) Application Client.
- There is no support for hardware cryptographic devices on iSeries.

- Only Version 6.1 and later, Web Services Security applications can take advantage of the hardware cryptographic support.

  Note: Versions 5.*x* and 6.0.*x* Web Services Security applications can run in a Version 6.1 WebSphere Application Server, but these versions cannot take advantage of the hardware cryptographic support.

Long-term usage of session keys

You can configure WebSphere Application Server to use the hardware keystore, or you can configure the hardware acceleration card to allow the long-term usage of session keys. Session keys might be insecure.

If you are concerned about insecure session keys, configure WebSphere Application Server to use the hardware keystore. See the information about how to enable cryptographic keys that are stored in hardware devices in Web Services Security.

To configure the hardware acceleration card to allow the long-term usage of session keys, see the manufacturer's documentation for the specific hardware acceleration card. For example:

1. For the nCipher nforce 1600 server Version 2.23.6, follow the nCipher documentation instructions.
2. You can set the `CKNFAST_SECURITY_ASSURANCES_OVERRIDE=longterm` parameter in the `cknfastrc` configuration file. This configuration change eliminates the time limit that is associated with session keys.
3. Follow the documentation for Cipher to restart the nCipher server.
4. Restart WebSphere Application Server.

Default configuration:

You can use sample configurations with the administrative console for testing purposes. The configurations that you specify are reflected on the cell or server level.

The information in the following sections describes sample default bindings, sample general bindings, and samples for key stores, key locators, collection certificate store, trust anchors, and trusted ID evaluators. You can develop web services using the Java API for XML-based RPC (JAX-RPC) programming model, or for WebSphere Application Server Version 7, using the Java API for XML-Based Web Services (JAX-WS) programming model. Samples that are provided with WebSphere Application Server differ depending on which programming model you use.

best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

Do not use these sample configurations in a production environment as they are for sample and testing purposes only. To make modifications to these sample configurations, it is recommended that you use the administrative console provided by WebSphere Application Server.

Detailed information on the sample general bindings for the JAX-WS programming model is available in the topic General sample bindings for JAX-WS applications.

Information on configuring default bindings, key stores, key locators, collection certificate store, trust anchors, and trusted ID evaluators for the JAX-RPC programming model is available in the topic Default sample configurations for JAX-RPC.

General sample bindings for JAX-WS applications:

You can use sample bindings with the administrative console for testing purposes. The configurations that you specify are reflected on the cell or server level.

WebSphere Application Server Version 7.0 and later includes provider and client sample bindings for testing purposes. In the bindings, the product provides sample values for supporting tokens for different token types, such as the X.509 token, the username token, the LTPA token, and the Kerberos token. The bindings also include sample values for message protection information for token types such as X.509 and secure conversation. Both provider and client sample bindings can be applied to the applications attached with a system policy set, or application policy set, from the default local repository.

This information describes the general sample bindings for the Java API for XML-Based Web Services (JAX-WS) programming model. You can develop web services using the Java API for XML-based RPC (JAX-RPC) programming model, or for WebSphere Application Server Version 7.0 and later, using the Java API for XML-Based Web Services (JAX-WS) programming model. Sample general bindings may differ depending on which programming model you use. The following sections, describing various general sample bindings, are provided:
- "General client sample bindings"
- "Client sample bindings V2" on page 249
- "General provider sample bindings" on page 251
- "Provider sample bindings V2" on page 255

best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

Do not use these provider and client sample bindings in their default state in a production environment. You must modify the bindings to meet your security needs before using them in a production environment by making a copy of the bindings and then modifying the copy. For example, you must change the key and keystore settings to ensure security, and modify the binding settings to match your environment.

One set of general default bindings is shared by the applications to make application deployment easier. You can specify default bindings for your service provider or client that are used at the global security (cell) level, for a security domain, or for a particular server. The default bindings are used in the absence of an overriding binding specified at a lower scope. The order of precedence from lowest to highest that the application server uses to determine which default bindings to use is as follows:
1. Server level default
2. Security domain level default
3. Global security (cell) default

General client sample bindings
- The sample configuration for signing information generation, called `asymmetric-signingInfoRequest`, contains the following configuration:

- References the gen_signkeyinfo signing key information.
- The part reference configuration, which contains the transform configuration using the http://www.w3.org/2001/10/xml-exc-c14n# algorithm.
- The signing key information, gen_signkeyinfo, which contains this configuration:
  - The security token reference.
  - The gen_signx509token protection token asymmetric signature generator, as follows:
    - Contains the X.509 V3 Token v1.0 token type.
    - Contains the http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3 value type for the local part value.
    - Contains the wss.generate.x509 JAAS login
  - The X.509 Callback Handler. The callback handler calls the custom keystore in ${USER_INSTALL_ROOT}/etc/ws-security/samples/dsigsender.ks, with these characteristics:
    - The keystore type is JKS.
    - The keystore password is client.
    - The alias name of the trusted certificate is soapca.
    - The alias name of the personal certificate is soaprequester.
    - The key password client issued by the intermediary certificate authority Int CA2, which is in turn issued by soapca.
- The signature method http://www.w3.org/2000/09/xmldsig#rsa-sha1.
- The canonicalization method http://www.w3.org/2001/10/xml-exc-c14n#.

- The sample configuration for signing information generation called symmetric-signingInfoRequest contains the following configuration:
  - References the gen_signsctkeyinfo signing key information.
  - The part reference configuration, which contains the transform configuration using the http://www.w3.org/2001/10/xml-exc-c14n# algorithm.
  - The signing key information, gen_signsctkeyinfo, which contains this configuration:
    - The security token reference.
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The gen_scttoken protection token generator, as follows:
      - Contains the Secure Conversation Token Version 1.3 token type.
      - Contains the http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct value type as the local part value.
      - Contains wss.generate.sct JAAS login
    - The WS-Trust Callback Handler.
  - The signature method http://www.w3.org/2000/09/xmldsig#hmac-sha1.
  - The canonicalization method http://www.w3.org/2001/10/xml-exc-c14n#.
- The sample configuration for encryption information generation, called asymmetric-encryptionInfoRequest, contains the following configuration:
  - References the gen_enckeyinfo encryption key information.
  - Encryption key information, named gen_enckeyinfo, which contains this configuration:
    - The key identifier.

- The gen_encx509token protection token asymmetric encryption generator, as follows:
  - Keystore type is JCEKS.
  - Keystore password is client.
  - Alias name of the trusted certificate is soapca.
  - Alias name of the personal certificate is bob.
  - Key password client issued by intermediary certificate authority Int CA2, which is in turn issued by soapca.
- The X.509 Callback Handler. The callback handler calls the custom keystore in ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks.
- The key encryption method http://www.w3.org/2001/04/xmlenc#rsa-1_5.

- The sample configuration for encryption information generation, called symmetric-encryptionInfoRequest, contains the following configuration:
  - References the gen_encsctkeyinfo encryption key information.
  - The encryption key information, gen_encsctkeyinfo, which contains this configuration:
    - The security token reference.
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The gen_scttoken protection token generator, which contains the following configuration:
      - Contains the Secure Conversation Token v1.3 token type.
      - Contains the http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct value type for the local part value.
      - Contains wss.generate.sct JAAS login.
    - The WS-Trust Callback Handler.
  - The data encryption method http://www.w3.org/2001/04/xmlenc#aes128-cbc.
- The sample configuration for signing information consumption, called asymmetric-signingInfoResponse, contains the following configuration:
  - References the con_signkeyinfo signing key information.
  - The part reference configuration, which uses the transform configuration http://www.w3.org/2001/10/xml-exc-c14n# algorithm.
  - The signing key information, named con_signkeyinfo, which contains the following configuration:
    - The con_signx509token protection token asymmetric signature consumer, as follows:
      - Contains the X.509 V3 Token v1.0 token type.
      - Contains the http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3 value type for the local part value.
      - Contains the wss.consume.x509 JAAS login.
    - The X.509 Callback Handler, as follows:
      - References a certificate store named DigSigCertStore.
      - References a trusted anchor store named DigSigTrustAnchor.
  - The signature method http://www.w3.org/2000/09/xmldsig#rsa-sha1.
  - The canonicalization method http://www.w3.org/2001/10/xml-exc-c14n#.
- The sample configuration for signing information consumption, called symmetric-signingInfoResponse, contains the following configuration:

– References the con_sctsignkeyinfo signing key information.
– The part reference configuration, which uses the transform configuration http://www.w3.org/2001/10/xml-exc-c14n# algorithm.
– The signing key information, named con_sctsignkeyinfo, which contains the following configuration:
  - The derived key, as follows:
    • Requires explicit derived key token.
    • WS-SecureConversation as the client label.
    • WS-SecureConversation as the service label.
    • Key length of 16 bytes.
    • Nonce length of 16 bytes.
  - The con_scttoken protection token consumer, as follows:
    • Contains the Secure Conversation Token v1.3 token type.
    • Contains the http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct value type for the local part value.
    • Contains the wss.consume.sct JAAS login.
  - The WS-SecureConversation Callback Handler.
– The signature method http://www.w3.org/2000/09/xmldsig#hmac-sha1.
– The canonicalization method http://www.w3.org/2001/10/xml-exc-c14n#.

• The sample configuration for encryption information consumption, called asymmetric-encryptionInfoResponse, which contains the following configuration:
  – References the dec_keyinfo encryption key information.
  – The encryption key information, named dec_keyinfo, which contains the following configuration:
    - The con_encx509token protection token asymmetric encryption consumer, as follows:
      • Contains the X.509 V3 Token v1.0 token type.
      • Contains the http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3 value type for the local part value.
      • Contains the wss.consume.x509 JAAS login.
    - The X.509 Callback Handler. The callback handler calls the custom keystore in ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks, with the follow characteristics:
      • The keystore type is JCEKS.
      • The keystore password is client.
      • The alias name of the trusted certificate is soapca.
      • The alias name of the personal certificate is alice.
      • The key password client issued by intermediary certificate authority Int CA2, which is in turn issued by soapca.
  – The key encryption method http://www.w3.org/2001/04/xmlenc#rsa-1_5.

• The sample configuration for encryption information consumption, called symmetric-encryptionInfoResponse, contains the following configuration:
  – References the dec_sctkeyinfo encryption key information.
  – The encryption key information, named dec_sctkeyinfo, contains the following configuration:
    - The derived key, as follows:
      • Requires explicit derived key token.
      • WS-SecureConversation as the client label.
      • WS-SecureConversation as the service label.
      • Key length of 16 bytes.

- Nonce length of 16 bytes.
- The con_scttoken protection token consumer, as follows:
  - Contains the Secure Conversation Token v1.3 token type.
  - Contains the http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct value type for the local part value.
  - Contains the wss.consume.sct JAAS login.
- The WS-SecureConversation Callback Handler.

– The data encryption method http://www.w3.org/2001/04/xmlenc#aes128-cbc.

- The sample configuration for authentication token generation, called gen_signkrb5token, contains the following configuration:
  – The custom token type for the Kerberos v5 token, which uses http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ for the local part value.
  – The wss.generate.KRB5BST JAAS login.
  – The following custom properties:
    - com.ibm.wsspi.wssecurity.krbtoken.targetServiceName, the target Kerberos service name.
    - com.ibm.wsspi.wssecurity.krbtoken.targetServiceHost, the host name associated with the target Kerberos service name,

      You must provide the correct values for your environment before using this configuration.
  – The custom Kerberos token callback handler. You must provide the correct values for the Kerberos client principal and password.
- The sample configuration for authentication token generation, called gen_signltpaproptoken, contains the following configuration:
  – The token type LTPA propagation token, as follows:
    - Contains LTPA_PROPAGATION for the local part value.
    - Contains http://www.ibm.com/websphere/appserver/tokentype for the Namespace URI value.
  – Contains the wss.generate.ltpaProp JAAS login.
  – Uses the LTPA token callback handler.
- The sample configuration for authentication token generation, called gen_signltpatoken, contains the following configuration:
  – The token type of LTPA Token v2.0, as follows:
    - Contains LTPA_PROPAGATION for the local part value.
    - Contains http://www.ibm.com/websphere/appserver/tokentype for the Namespace URI value.
  – The wss.generate.ltpa JAAS login.
  – The LTPA token callback handler.
- The sample configuration for authentication token generation, called gen_signunametoken, contains the following configuration:
  – The token type of Username Token v1.0, which uses http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken for the local part value.
  – The wss.generate.unt JAAS login.
  – The Username token callback handler, as follows:
    - Contains basic authentication fields. You must provide the correct values for your environment for client principal and password.
    - Contains the following custom properties:
      - com.ibm.wsspi.wssecurity.token.username.addNonce for adding the nonce value.
      - com.ibm.wsspi.wssecurity.token.username.addTimestamp for adding the time stamp value.

Client sample bindings V2

Two new general sample bindings, Client sample V2, and Provider sample V2, have been added to the product. While many of the configurations are the same as previous versions of the client sample and provider sample bindings, there are several additional, new sample configurations. To use these new bindings, create a new profile after installing the product. For more information, read the topic Configuring Kerberos policy sets and V2 general sample bindings.

- The sample configuration for signing information generation, called `symmetric-KrbsignInfoRequest`, contains the following configuration:
  - References the `gen_reqKRBsignkeyinfo` signing key information.
  - The part reference configuration, which contains the transform configuration using the `http://www.w3.org/2001/10/xml-exc-c14n#` algorithm.
  - The signing key information, `gen_reqKRBsignkeyinfo`, which contains this configuration:
    - The security token reference.
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The `gen_krb5token` protection token generator, as follows:
      - Contains the Kerberos V5 GSS AP_REQ binary security token type.
      - Contains the `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ` value type as the local part value.
      - Contains `wss.generate.KRB5BST JAAS` login
    - The com.ibm.websphere.wssecurity.callbackhandler.KRBTokenGenerateCallbackHandler.
  - The signature method `http://www.w3.org/2000/09/xmldsig#hmac-sha1`.
  - The canonicalization method `http://www.w3.org/2001/10/xml-exc-c14n#`.
- The sample configuration for encryption information generation, called `symmetric-KrbEncInfoRequest`, contains the following configuration:
  - References the `gen_reqKRBenckeyinfo` encryption key information.
  - The encryption key information, `gen_reqKRBenckeyinfo`, which contains this configuration:
    - The security token reference.
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The `gen_krb5token` protection token generator, which contains the following configuration:
      - Contains the Kerberos V5 GSS AP_REQ binary security token type.
      - Contains the `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ` value type for the local part value.
      - Contains `wss.generate.KRB5BST` JAAS login.
    - The com.ibm.websphere.wssecurity.callbackhandler.KRBTokenGenerateCallbackHandler.
  - The data encryption method `http://www.w3.org/2001/04/xmlenc#aes128-cbc`.

- The sample configuration for signing information consumption, called symmetric-KrbsignInfoResponse, contains the following configuration:
  - References the con_respKRBsignkeyinfo signing key information.
  - The part reference configuration, which uses the transform configuration http://www.w3.org/2001/10/xml-exc-c14n# algorithm.
  - The signing key information, named con_respKRBsignkeyinfo, which contains the following configuration:
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The con_krb5token protection token consumer, as follows:
      - Contains the Kerberos V5 GSS AP_REQ binary security token type.
      - Contains the http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ value type for the local part value.
      - Contains the wss.consume.KRB5BST JAAS login.
    - The com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler
  - The signature method http://www.w3.org/2000/09/xmldsig#hmac-sha1.
  - The canonicalization method http://www.w3.org/2001/10/xml-exc-c14n#.
- The sample configuration for encryption information consumption, called symmetric-KrbEncInfoResponse, contains the following configuration:
  - References the con_respKRBenckeyinfo encryption key information.
  - The encryption key information, named con_respKRBenckeyinfo, contains the following configuration:
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The con_krb5token protection token consumer, as follows:
      - Contains the Kerberos V5 GSS AP_REQ binary security token type.
      - Contains the http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ value type for the local part value.
      - Contains the wss.consume.KRB5BST JAAS login.
    - The com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler.
  - The data encryption method http://www.w3.org/2001/04/xmlenc#aes128-cbc.
- The sample configuration for authentication token generation, called gen_krb5token, contains the following configuration:
  - The custom token type for the Kerberos V5 token, which uses http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ for the local part value.
  - The wss.generate.KRB5BST JAAS login.
  - The following custom properties:
    - com.ibm.wsspi.wssecurity.krbtoken.targetServiceName, the target Kerberos service name.
    - com.ibm.wsspi.wssecurity.krbtoken.targetServiceHost, the host name associated with the target Kerberos service name.

Note: You must provide the correct values for your environment before using this configuration.

– The custom Kerberos token callback handler.

Note: You must provide the correct values for the Kerberos client principal and password.

- The sample configuration for authentication token generation, called `con_krb5token`, contains the following configuration:
  – The custom token type for the Kerberos V5 token, which uses `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ` for the local part value.
  – The `wss.consume.KRB5BST` JAAS login.
  – The custom Kerberos token callback handler.

General provider sample bindings

- The sample configuration for signing information consumption, called `asymmetric-signingInfoRequest`, contains the following configuration:
  – References the `con_signkeyinfo` signing key information.
  – The part reference configuration, which uses the transform configuration `http://www.w3.org/2001/10/xml-exc-c14n#` algorithm.
  – The signing key information, named `con_signkeyinfo`, which contains the following configuration:
    - The `con_signx509token` protection token asymmetric signature consumer, as follows:
      - Contains the X.509 V3 Token v1.0 token type.
      - Contains the `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3` value type for the local part value.
      - Contains the `wss.consume.x509` JAAS login.
    - The X.509 Callback Handler, as follows:
      - References a certificate store named DigSigCertStore.
      - References a trusted anchor store named DigSigTrustAnchor.
  – The signature method `http://www.w3.org/2000/09/xmldsig#rsa-sha1`.
  – The canonicalization method `http://www.w3.org/2001/10/xml-exc-c14n#`.
- The sample configuration for signing information consumption, called `symmetric-signingInfoRequest`, contains the following configuration:
  – References the `con_sctsignkeyinfo` signing key information.
  – The part reference configuration, which uses the transform configuration `http://www.w3.org/2001/10/xml-exc-c14n#` algorithm.
  – The signing key information, named `con_sctsignkeyinfo`, which contains the following configuration:
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The `con_scttoken` protection token generator, as follows:
      - Contains the Secure Conversation Token v1.3 token type.
      - Contains the `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct` value type for the local part value.
      - Contains the `wss.consume.sct` JAAS login.
    - The WS-SecureConversation Callback Handler.
  – The signature method `http://www.w3.org/2000/09/xmldsig#hmac-sha1`.

- The canonicalization method `http://www.w3.org/2001/10/xml-exc-c14n#`.

- The sample configuration for encryption information consumption, called `asymmetric-encryptionInfoRequest`, contains the following configurations:
  - References the `dec_keyinfo` encryption key information.
  - The encryption key information, named `dec_keyinfo`, which contains the following configuration:
    - The `con_encx509token` protection token asymmetric encryption consumer, as follows:
      - Contains the X.509 V3 Token v1.0 token type.
      - Contains the `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3` value type for the local part value.
      - Contains the `wss.consume.x509` JAAS login.
    - The X.509 Callback Handler. The callback handler calls the custom keystore in `${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks`, with the following characteristics:
      - The keystore type is `JCEKS`.
      - The keystore password is `client`.
      - The alias name of the trusted certificate is `soapca`.
      - The alias name of the personal certificate is `bob`.
      - The key password client issued by intermediary certificate authority `Int CA2`, which is in turn issued by `soapca`.
  - The key encryption method `http://www.w3.org/2001/04/xmlenc#rsa-1_5`.

- The sample configuration for encryption information consumption, called `symmetric-encryptionInfoRequest`, contains the following configuration:
  - References the `dec_sctkeyinfo` encryption key information.
  - The encryption key information, named `dec_sctkeyinfo`, which contains the following configuration:
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The `con_scttoken` protection token consumer, as follows:
      - Contains the Secure Conversation Token v1.3 token type.
      - Contains the `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct` value type for the local part value.
      - Contains the `wss.consume.sct` JAAS login.
    - The WS-SecureConversation Callback Handler.
  - The data encryption method `http://www.w3.org/2001/04/xmlenc#aes128-cbc`.

- The sample configuration for signing information generation, called `asymmetric-signingInfoResponse`, contains the following configuration:
  - References the `gen_signkeyinfo` signing key information.
  - The part reference configuration, which uses the transform configuration `http://www.w3.org/2001/10/xml-exc-c14n#` algorithm.
  - The signing key information, named `gen_signkeyinfo`, which contains the following configuration:
    - The security token reference.
    - The `gen_signx509token` protection token asymmetric signature generator, as follows:
      - Contains the X.509 V3 Token v1.0 token type.

- Contains the http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3 value type for the local part value.
- Contains the wss.generate.x509 JAAS login.
- The X.509 Callback Handler. The callback handler calls the custom keystore in ${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks, with the following characteristics:
  - The keystore type is JKS.
  - The keystore password is client.
  - The alias name of the trusted certificate is soapca.
  - The alias name of the personal certificate is soapprovider.
  - The key password client issued by intermediary certificate authority Int CA2, which is in turn issued by soapca.
- The signature method http://www.w3.org/2000/09/xmldsig#rsa-sha1.
- The canonicalization method http://www.w3.org/2001/10/xml-exc-c14n#.

- The sample configuration for signing information generation, called symmetric-signingInfoResponse, contains the following configuration:
  - References the gen_signsctkeyinfo signing key information.
  - The part reference configuration, which uses the transform configuration http://www.w3.org/2001/10/xml-exc-c14n# algorithm.
  - The signing key information, named gen_signsctkeyinfo, which contains the following configuration:
    - The security token reference.
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The gen_scttoken protection token generator, as follows:
      - Contains the Secure Conversation Token v1.3 token type.
      - Contains the http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct value type for the local part value.
      - Contains the wss.generate.sct JAAS login.
    - The WS-Trust Callback Handler.
  - The signature method http://www.w3.org/2000/09/xmldsig#hmac-sha1.
  - The canonicalization method http://www.w3.org/2001/10/xml-exc-c14n#.
- The sample configuration for encryption information generation, called asymmetric-encryptionInfoResponse, contains the following configuration:
  - References the gen_enckeyinfo encryption key information.
  - The encryption key information, named gen_enckeyinfo, contains the following configuration
    - The key identifier.
    - The gen_encx509token protection token asymmetric encryption generator, as follows:
      - Contains the X.509 V3 Token v1.0 token type.
      - Contains the http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3 value type for the local part value.
      - Contains the wss.generate.x509 JAAS login.

- Uses X.509 Callback Handler. The callback handler calls the custom keystore in ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks, with the following characteristics:
  - The keystore type is JCEKS.
  - The keystore password is client.
  - The alias name of the trusted certificate is soapca.
  - The alias name of the personal certificate is alice.
  - The key password client issued by intermediary certificate authority Int CA2, which is in turn issued by soapca.
- The key encryption method http://www.w3.org/2001/04/xmlenc#rsa-1_5.

- The sample configuration for encryption information generation, called symmetric-encryptionInfoResponse, contains the following configuration:
  - References the gen_encsctkeyinfo encryption key information.
  - The encryption key information, named gen_encsctkeyinfo, contains the following configuration:
    - The security token reference.
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The gen_scttoken protection token generator, as follows:
      - Contains the Secure Conversation Token v1.3 token type.
      - Contains the http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct value type for the local part value.
      - Contains the wss.generate.sct JAAS login.
    - The WS-Trust Callback Handler.
  - The data encryption method http://www.w3.org/2001/04/xmlenc#aes128-cbc.
- The sample configuration for authentication token consumption, called con_krb5token, contains the following configuration:
  - The custom token type for Kerberos v5 token, which uses http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ for the local part value.
  - The wss.consume.KRB5BST JAAS login.
  - The custom Kerberos token callback handler.
- The sample configuration for authentication token consumption, called con_ltpaproptoken, contains the following configuration:
  - The token type LTPA propagation token.
  - The wss.consume.ltpaProp JAAS login.
  - The LTPA token callback handler.
- The sample configuration for authentication token consumption, called con_ltpatoken, contains the following configuration:
  - The token type LTPA Token v2.0, with the following characteristics:
    - Contains LTPAv2 for the local part value.
    - Contains http://www.ibm.com/websphere/appserver/tokentype for the Namespace URI value.
  - The wss.consume.ltpa JAAS login
  - The LTPA token callback handler.

- The sample configuration for authentication token consumption, called con_unametoken, contains the following configuration:
  - Token type Username Token v1.0, which uses http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken for the local part value.
  - The wss.consume.unt JAAS login.
  - The Username token callback handler, with the following custom properties:
    - com.ibm.wsspi.wssecurity.token.username.verifyNonce for verifying the nonce value.
    - com.ibm.wsspi.wssecurity.token.username.verifyTimestamp for verifying the time stamp value.

Provider sample bindings V2

Two new general sample bindings, Client sample V2, and Provider sample V2, have been added to the product. While many of the configurations are the same as previous versions of the client sample and provider sample bindings, there are several additional, new sample configurations. To use these new bindings, create a new profile after installing the product. For more information, read the topic Configuring Kerberos policy sets and V2 general sample bindings.

- The sample configuration for signing information generation, called symmetric-KrbsignInfoRequest, contains the following configuration:
  - References the con_respKRBsignkeyinfo signing key information.
  - The part reference configuration, which contains the transform configuration using the http://www.w3.org/2001/10/xml-exc-c14n# algorithm.
  - The signing key information, con_respKRBsignkeyinfo, which contains this configuration:
    - The security token reference.
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The con_krb5token protection token consumer, as follows:
      - Contains the Kerberos V5 GSS AP_REQ binary security token type.
      - Contains the http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ value type as the local part value.
      - Contains wss.consume.KRB5BST JAAS login.
    - The com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler.
  - The signature method http://www.w3.org/2000/09/xmldsig#hmac-sha1.
  - The canonicalization method http://www.w3.org/2001/10/xml-exc-c14n#.
- The sample configuration for encryption information generation, called symmetric-KrbEncInfoRequest, contains the following configuration:
  - References the con_reqKRBenckeyinfo encryption key information.
  - The encryption key information, con_reqKRBenckeyinfo, which contains this configuration:
    - The security token reference.
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.

- The con_krb5token protection token consumer, which contains the following configuration:
  - Contains the Kerberos V5 GSS AP_REQ binary security token type.
  - Contains the http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ value type for the local part value.
  - Contains wss.consume.KRB5BST JAAS login.
- The com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler.

– The data encryption method http://www.w3.org/2001/04/xmlenc#aes128-cbc.

- The sample configuration for signing information consumption, called symmetric-KrbsignInfoResponse, contains the following configuration:
  – References the gen_respKRBsignkeyinfo signing key information.
  – The part reference configuration, which uses the transform configuration http://www.w3.org/2001/10/xml-exc-c14n# algorithm.
  – The signing key information, named gen_respKRBsignkeyinfo, which contains the following configuration:
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The gen_krb5token protection token generator, as follows:
      - Contains the Kerberos V5 GSS AP_REQ binary security token type.
      - Contains the http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ value type for the local part value.
      - Contains the wss.generate.KRB5BST JAAS login.
    - The com.ibm.websphere.wssecurity.callbackhandler.KRBTokenGenerateCallbackHandler.
  – The signature method http://www.w3.org/2000/09/xmldsig#hmac-sha1.
  – The canonicalization method http://www.w3.org/2001/10/xml-exc-c14n#.
- The sample configuration for encryption information consumption, called symmetric-KrbEncInfoResponse, contains the following configuration:
  – References the gen_respKRBenckeyinfo encryption key information.
  – The encryption key information, named gen_respKRBenckeyinfo, contains the following configuration:
    - The derived key, as follows:
      - Requires explicit derived key token.
      - WS-SecureConversation as the client label.
      - WS-SecureConversation as the service label.
      - Key length of 16 bytes.
      - Nonce length of 16 bytes.
    - The gen_krb5token protection token generator, as follows:
      - Contains the Kerberos V5 GSS AP_REQ binary security token type.
      - Contains the http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ value type for the local part value.
      - Contains the wss.generate.KRB5BST JAAS login.
    - The com.ibm.websphere.wssecurity.callbackhandler.KRBTokenGenerateCallbackHandler
  – The data encryption method http://www.w3.org/2001/04/xmlenc#aes128-cbc.

- The sample configuration for authentication token generation, called `gen_krb5token`, contains the following configuration:
  - The custom token type for the Kerberos V5 token, which uses `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ` for the local part value.
  - The `wss.generate.KRB5BST` JAAS login.
  - The custom Kerberos token callback handler.
- The sample configuration for authentication token generation, called `con_krb5token`, contains the following configuration:
  - The custom token type for the Kerberos V5 token, which uses `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ` for the local part value.
  - The `wss.consume.KRB5BST` JAAS login.
  - The custom Kerberos token callback handler.

Default sample configurations for JAX-RPC:

Use sample configurations with the administrative console for testing purposes. The configurations that you specify are reflected on the cell or server level.

This information describes the sample default bindings, key stores, key locators, collection certificate store, trust anchors, and trusted ID evaluators for WebSphere Application Server using the API for XML-based RPC (JAX-RPC) programming model. You can develop web services using the Java API for XML-based RPC (JAX-RPC) programming model, or for WebSphere Application Server Version 7 and later, using the Java API for XML-Based Web Services (JAX-WS) programming model. Sample default bindings, key stores, key locators, collection certificate store, trust anchors, and trusted ID evaluator may differ depending on which programming model you use.

best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

Do not use these configurations in a production environment as they are for sample and testing purposes only. To make modifications to these sample configurations, it is recommended that you use the administrative console provided by WebSphere Application Server.

For a Web Services Security-enabled application, you must correctly configure a deployment descriptor and a binding. In WebSphere Application Server, one set of default bindings is shared by the applications to make application deployment easier. The default binding information for the cell level and the server level can be overridden by the binding information on the application level. The Application Server searches for binding information for an application on the application level before searching the server level, and then the cell level.

The following sample configurations are for WebSphere Application Server using the API for XML-based RPC (JAX-RPC) programming model.

Default generator binding

WebSphere Application Server provides a sample set of default generator bindings. The default generator bindings contain both signing information and encryption information.

The sample signing information configuration is called gen_signinfo and contains the following configurations:

- Uses the following algorithms for the gen_signinfo configuration:
  – Signature method: http://www.w3.org/2000/09/xmldsig#rsa-sha1
  – Canonicalization method: http://www.w3.org/2001/10/xml-exc-c14n#
- References the gen_signkeyinfo signing key information. The following information pertains to the gen_signkeyinfo configuration:
  – Contains a part reference configuration that is called gen_signpart. The part reference is not used in default binding. The signing information applies to all of the Integrity or Required Integrity elements within the deployment descriptors and the information is used for naming purposes only. The following information pertains to the gen_signpart configuration:
    - Uses the transform configuration called transform1. The following transforms are configured for the default signing information:
      - Uses the http://www.w3.org/2001/10/xml-exc-c14n# algorithm
      - Uses the http://www.w3.org/2000/09/xmldsig#sha1 digest method
  – Uses the security token reference, which is the configured default key information.
  – Uses the SampleGeneratorSignatureKeyStoreKeyLocator key locator. For more information on this key locator, see “Sample key locators” on page 260.
  – Uses the gen_signtgen token generator, which contains the following configuration:
    - Contains the X.509 token generator, which generates the X.509 token of the signer.
    - Contains the gen_signtgen_vtype value type URI.
    - Contains the http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3 value type local name value.
  – Uses X.509 Callback Handler. The callback handler calls the ${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks key store.
    - The key store password is client.
    - The alias name of the trusted certificate is soapca.
    - The alias name of the personal certificate is soaprequester.
    - The key password client issued by intermediary certificate authority Int CA2, which is in turn issued by soapca.

The sample encryption information configuration is called gen_encinfo and contains the following configurations:

- Uses the following algorithms for the gen_encinfo configuration:
  – Data encryption method: http://www.w3.org/2001/04/xmlenc#tripledes-cbc
  – Key encryption method: http://www.w3.org/2001/04/xmlenc#rsa-1_5
- References the **gen_enckeyinfo** encryption key information. The following information pertains to the gen_enckeyinfo configuration:
  – Uses the key identifier as the default key information.
  – Contains a reference to the SampleGeneratorEncryptionKeyStoreKeyLocator key locator. For more information on this key locator, see “Sample key locators” on page 260.
  – Uses the gen_signtgen token generator, which has the following configuration:
    - Contains the X.509 token generator, which generates the X.509 token of the signer.
    - Contains the gen_enctgen_vtype value type URI.
    - Contains the http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3 value type local name value.
  – Uses X.509 Callback Handler. The callback handler calls the ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks key store.

- The key store password is storepass.
- The secret key CN=Group1 has an alias name of Group1 and a key password of keypass.
- The public key CN=Bob, O=IBM, C=US has an alias name of bob and a key password of keypass.
- The private key CN=Alice, O=IBM, C=US has an alias name of alice and a key password of keypass.

Default consumer binding

WebSphere Application Server provides a sample set of default consumer binding. The default consumer binding contain both signing information and encryption information.

The sample signing information configuration is called con_signinfo and contains the following configurations:

- Uses the following algorithms for the con_signinfo configuration:
  - Signature method: http://www.w3.org/2000/09/xmldsig#rsa-sha1
  - Canonicalization method: http://www.w3.org/2001/10/xml-exc-c14n#
- Uses the con_signkeyinfo signing key information reference. The following information pertains to the con_signkeyinfo configuration:
  - Contains a part reference configuration that is called con_signpart. The part reference is not used in default binding. The signing information applies to all of the Integrity or RequiredIntegrity elements within the deployment descriptors and the information is used for naming purposes only. The following information pertains to the con_signpart configuration:
    - Uses the transform configuration called reqint_body_transform1. The following transforms are configured for the default signing information:
      - Uses the http://www.w3.org/2001/10/xml-exc-c14n# algorithm.
      - Uses the http://www.w3.org/2000/09/xmldsig#sha1 digest method.
  - Uses the security token reference, which is the configured default key information.
  - Uses the SampleX509TokenKeyLocator key locator. For more information on this key locator, see “Sample key locators” on page 260.
  - References the con_signtcon token consumer configuration. The following information pertains to the con_signtcon configuration:
    - Uses the X.509 Token Consumer, which is configured as the consumer for the default signing information.
    - Contains the signtconsumer_vtype value type URI.
    - Contains the http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3 value type local name value.
  - Contains a JAAS configuration called system.wssecurity.X509BST that references the following information:
    - Trust anchor: SampleClientTrustAnchor
    - Collection certificate store: SampleCollectionCertStore

The encryption information configuration is called con_encinfo and contains the following configurations:

- Uses the following algorithms for the con_encinfo configuration:
  - Data encryption method: http://www.w3.org/2001/04/xmlenc#tripledes-cbc
  - Key encryption method: http://www.w3.org/2001/04/xmlenc#rsa-1_5
- References the con_enckeyinfo encryption key information. This key actually decrypts the message. The following information pertains to the con_enckeyinfo configuration:
  - Uses the key identifier, which is configured as the key information for the default encryption information.

– Contains a reference to the SampleConsumerEncryptionKeyStoreKeyLocator key locator. For more information on this key locator, see “Sample key locators.”
– References the con_enctcon token consumer configuration. The following information pertains to the con_enctcon configuration:
  - Uses the X.509 token consumer, which is configured for the default encryption information.
  - Contains the enctconsumer_vtype value type URI.
  - Contains the http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3 value type local name value.
– Contains a JAAS configuration called system.wssecurity.X509BST.

Sample key store configurations

The following sample key stores are for testing purposes only; do not use these key stores in a production environment:

- ${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks
  – The key store format is JKS.
  – The key store password is client.
  – The trusted certificate has a soapca alias name.
  – The personal certificate has a soaprequester alias name and a client key password that is issued by the Int CA2 intermediary certificate authority, which is, in turn, issued by soapca.
- ${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
  – The key store format is JKS.
  – The key store password is server.
  – The trusted certificate has a soapca alias name.
  – The personal certificate has a soapprovider alias name and a server key password that is issued by the Int CA2 intermediary certificate authority, which is, in turn, issued by soapca.
- ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks
  – The key store format is JCEKS.
  – The key store password is storepass.
  – The CN=Group1 DES secret key has a Group1 alias name and a keypass key password.
  – The CN=Bob, O=IBM, C=US public key has a bob alias name and a keypass key password.
  – The CN=Alice, O=IBM, C=US private key has a alice alias name and a keypass key password.
- ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks
  – The key store format is JCEKS.
  – The key store password is storepass.
  – The CN=Group1 DES secret key has a Group1 alias name and a keypass key password.
  – The CN=Bob, O=IBM, C=US private key has a bob alias name and a keypass key password.
  – The CN=Alice, O=IBM, C=US public key has a alice alias name and a keypass key password.
- ${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer
  – The intermediary certificate is signed by soapca and it signs both the soaprequester and the soapprovider.

Sample key locators

Key locators are used to locate the key for digital signature, encryption, and decryption. For information on how to modify these sample key locator configurations, see the following articles:

- “Configuring the key locator using JAX-RPC for the generator binding on the application level” on page 943

- "Configuring the key locator using JAX-RPC for the consumer binding on the application level" on page 950
- "Configuring the key locator using JAX-RPC on the server or cell level" on page 952

SampleClientSignerKey
This key locator is used by the request sender for a Version 5.x application to sign the SOAP message. The signing key name is `clientsignerkey`, which is referenced in the signing information as the signing key name.

SampleServerSignerKey
This key locator is used by the response sender for a Version 5.x application to sign the SOAP message. The signing key name is `serversignerkey`, which can be referenced in the signing information as the signing key name.

SampleSenderEncryptionKeyLocator
This key locator is used by the sender for a Version 5.x application to encrypt the SOAP message. It is configured to use the `${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks` key store and the `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` key store key locator. The implementation is configured for the DES secret key. To use asymmetric encryption (RSA), you must add the appropriate RSA keys.

SampleReceiverEncryptionKeyLocator
This key locator is used by the receiver for a Version 5.x application to decrypt the encrypted SOAP message. The implementation is configured to use the `${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks` key store and the `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` key store key locator. The implementation is configured for symmetric encryption (DES or TRIPLEDES). To use RSA, you must add the private key `CN=Bob, O=IBM, C=US`, alias name `bob`, and key password `keypass`.

SampleResponseSenderEncryptionKeyLocator
This key locator is used by the response sender for a Version 5.x application to encrypt the SOAP response message. It is configured to use the `${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks` key store and the `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator` key store key locator. This key locator maps an authenticated identity (of the current thread) to a public key for encryption. By default, WebSphere Application Server is configured to map to public key `alice`, and you must change WebSphere Application Server to the appropriate user. The `SampleResponseSenderEncryptionKeyLocator` key locator also can set a default key for encryption. By default, this key locator is configured to use public key `alice`.

SampleGeneratorSignatureKeyStoreKeyLocator
This key locator is used by generator to sign the SOAP message. The signing key name is `SOAPRequester`, which is referenced in the signing information as the signing key name. It is configured to use the `${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks` key store and the `com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator` key store key locator.

SampleConsumerSignatureKeyStoreKeyLocator
This key locator is used by the consumer to verify the digital signature in the SOAP message. The signing key is `SOAPProvider`, which is referenced in the signing information. It is configured to use the `${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks` key store and the `com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator` key store key locator.

SampleGeneratorEncryptionKeyStoreKeyLocator
This key locator is used by the generator to encrypt the SOAP message. It is configured to use the `${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks` key store and the `com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator` key store key locator.

SampleConsumerEncryptionKeyStoreKeyLocator
This key locator is used by the consumer to decrypt an encrypted SOAP message. It is configured

to use the `${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks` key store and the `com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator` key store key locator.

SampleX509TokenKeyLocator
This key locator is used by the consumer to verify a digital certificate in an X.509 certificate. It is configured to use the `${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks` key store and the `com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator` key store key locator.

Sample collection certificate store

Collection certificate stores are used to validate the certificate path. For information on how to modify this sample collection certificate store, see the following articles:

- "Configuring the collection certificate store for the generator binding on the application level" on page 961
- "Configuring the collection certificate store for the consumer binding on the application level" on page 971
- "Configuring the collection certificate on the server or cell level" on page 973

SampleCollectionCertStore
This collection certificate store is used by the response consumer and the request generator to validate the signer certificate path.

Sample trust anchors

Trust anchors are used to validate the trust of the signer certificate. For information on how to modify the sample trust anchor configurations, see the following articles:

- "Configuring trust anchors for the generator binding on the application level" on page 954
- "Configuring trust anchors for the consumer binding on the application level" on page 959
- "Configuring trust anchors on the server or cell level" on page 960

SampleClientTrustAnchor
This trust anchor is used by the response consumer to validate the signer certificate. This trust anchor is configure to access the `${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks` key store.

SampleServerTrustAnchor
This trust anchor is used by the request consumer to validate the signer certificate. This trust anchor is configure to access the `${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks` key store.

Sample trusted ID evaluators

Trusted ID evaluators are used to establish trust before asserting the identity in identity assertion. For information on how to modify the sample trusted ID evaluator configuration, see "Configuring trusted ID evaluators on the server or cell level" on page 975.

SampleTrustedIDEvaluator
This trusted ID evaluator uses the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl implementation. The default implementation of com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator contains a list of trusted identities. This list, which is used for identity assertion, defines the key name and value pair for the trusted identity. The key name is in the form trustedId_* and the value is the trusted identity. For more information, see the example in "Configuring trusted ID evaluators on the server or cell level" on page 975.

Complete the following steps to define this information for the cell level in the administrative console:

1. Click **Security > Web services**.

2. Under Additional properties, click **Trusted ID evaluators** > `SampleTrustedIDEvaluator`.

Default implementations of the Web Services Security service provider programming interfaces:

This information describes the default implementations of the service provider interfaces (SPI) for Web Services Security within WebSphere Application Server. The default implementation classes and their functionality for both the JAX-RPC run time and the JAX-WS run time are discussed. You can use this information to create or modify the Web Services Security binding configuration.

best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

Default implementations for the JAX-RPC run time

com.ibm.wsspi.wssecurity.token.X509TokenGenerator
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side. It is responsible for creating the X.509 token object from the X.509 certificate, which is returned by the com.ibm.wsspi.wssecurity.auth.callback.{X509,PKCS7,PkiPath}CallbackHandler interface. Encode the token using the base 64 format and insert its XML representation into the SOAP message, if necessary.

com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface and it retrieves the X.509 certificate from the keystore file.

com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side. It is responsible for creating the username token object from user name and password that is returned by a javax.security.auth.callback.CallbackHandler implementation such as the following callback handler:

com.ibm.wsspi.wssecurity.auth.callback{GUIPrompt,NonPrompt,StdinPrompt}CallbackHandler.

It also inserts the XML representation of the token into the SOAP message, if necessary.

com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side and to validate (authenticate) the security token on the consumer side. This class retrieves the keys from the keystore files for digital signature and encryption.

com.ibm.wsspi.wssecurity.token.X509TokenConsumer
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to validate (authenticate) the security token on the consumer side. This class processes the X.509 token from the binary security token. This class decodes the Base64 encryption within the X.509 token and then invokes the system.wssecurity.X509BST Java Authentication and Authorization Service (JAAS) Login Configuration with the com.ibm.wsspi.wssecurity.auth.module.X509LoginModule login module to validate the X.509 token. An object of the com.ibm.wsspi.wssecurity.auth.token.X509Token is created for the validated X.509 token and stored in JAAS Subject.

com.ibm.wsspi.wssecurity.token.IDAssertionUsernameTokenConsumer
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to validate (authenticate) the security token on the consumer side. This class processes the username token for identity assertion (IDAssertion), which does not have a password element. This interface invokes the system.wssecurity.IDAssertionUsernameToken JAAS login configuration with the com.ibm.wsspi.wssecurity.auth.module.IDAssertionUsernameLoginModule login module to validate the IDAssertion user name token. An object of the com.ibm.wsspi.wssecurity.auth.token.UsernameToken class is created for the validated username token and stored in the JAAS Subject.

com.ibm.wsspi.wssecurity.auth.module.IDAssertionUsernameLoginModule
This class implements the javax.security.auth.spi.LoginModule interface and checks whether the username value is not empty. The login module assumes that the UsernameToken is valid if the username value is not empty.

com.ibm.wsspi.wssecurity.token.LTPATokenGenerator
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side. This class is responsible for Base 64 encoding the LTPA token object obtained from the com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler callback handler. The object is inserted into the Web Services Security header within the SOAP message, if necessary.

com.ibm.wsspi.wssecurity.token.LTPATokenConsumer
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to validate (authenticate) the security token on the consumer side. This class processes the LTPA token from the binary security token, and decodes the Base64 encoding within the LTPA token. An object of the com.ibm.wsspi.wssecurity.auth.token.LTPAToken class is created for the validated LTPA token and stored in the JAAS Subject.

com.ibm.wsspi.wssecurity.auth.module.X509LoginModule
This class implements the javax.security.auth.spi.LoginModule interface and validates the X.509 Certificate based on the trust anchor and the collection certification store configuration.

com.ibm.wsspi.wssecurity.token.UsernameTokenConsumer
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to validate (authenticate) the security token on the consumer side. This class processes the username token, extracts the user name and password, and then invokes the system.wssecurity.UsernameToken JAAS login configuration using the com.ibm.wsspi.wssecurity.auth.module.UsernameLoginModule login module to validate the user name and password. An object of the com.ibm.wsspi.wssecurity.auth.token.UsernameToken class is created for the validated username token and stored in the JAAS Subject.

com.ibm.wsspi.wssecurity.keyinfo.X509TokenKeyLocator
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side and to validate (authenticate) the security token on the consumer side. This class is used to retrieve a public key from a X.509 certificate. The X.509 certificate is stored in the X.509 token (com.ibm.wsspi.wssecurity.auth.token.X509Token) in the JAAS Subject. The X.509 token is created by the X.509 Token Consumer (com.ibm.wsspi.wssecurity.tokenX509TokenConsumer).

com.ibm.wsspi.wssecurity.keyinfo.SignerCertKeyLocator
The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side and to validate (authenticate) the security token on the consumer side. This class is used to retrieve a public key from the X.509 certificate of the request signer and encrypt the response. You can use this key locator in the response generator binding configuration only.

Important: This implementation assumes that only one signer certificate is used in the request.

com.ibm.wsspi.wssecurity.auth.token.UsernameToken
This implementation extends the com.ibm.wsspi.wssecurity.auth.token.WSSToken abstract class to represent the username token.

com.ibm.wsspi.wssecurity.auth.token.X509Token
This implementation extends the com.ibm.wsspi.wssecurity.auth.token.WSSToken abstract class to represent the X.509 binary security token (X.509 certificate).

com.ibm.wsspi.wssecurity.auth.token.LTPAToken
This implementation extends the com.ibm.wsspi.wssecurity.auth.token.WSSToken abstract class as a wrapper to the LTPA token that is extracted from the binary security token.

com.ibm.wsspi.wssecurity.auth.callback.PKCS7CallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface and is responsible for creating a certificate and binary data with or without a certificate revocation list (CRL) using the PKCS#7 encoding. The certificate and the binary data is passed back to the com.ibm.wsspi.wssecurity.token.X509TokenGenerator implementation through the com.ibm.wsspi.wssecurity.auth.callback.X509BSCallback callback handler.

com.ibm.wsspi.wssecurity.auth.callback.PkiPathCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface and it is responsible for creating a certificate and binary data without a CRL using the PkiPath encoding. The certificate and binary data is passed back to the com.ibm.wsspi.wssecurity.token.X509TokenGenerator implementation through the com.ibm.wsspi.wssecurity.auth.callback.X509BSCallback callback handler.

com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface and it is responsible for creating a certificate from the keystore file. The X.509 token certificate is passed back to the com.ibm.wsspi.wssecurity.token.X509TokenGenerator implementation through the com.ibm.wsspi.wssecurity.auth.callback.X509BSCallback callback handler.

com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler
This implementation generates a Lightweight Third Party Authentication (LTPA) token in the Web Services Security header as a binary security token. If basic authentication data is defined in the application binding file, it is used to perform a login, to extract the LTPA token from the WebSphere Application Server credentials, and to insert the token in the Web Services Security header. Otherwise, it extracts the LTPA security token from the invocation credentials (run as identity) and inserts the token in the Web Services Security header.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This implementation reads the basic authentication data from the application binding file. You might use this implementation on the server side to generate a username token.

com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
This implementation presents you with a login prompt to gather the basic authentication data. Use this implementation on the client side only.

com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
This implementation collects the basic authentication data using a standard in (stdin) prompt. Use this implementation on the client side only.

Restriction: If you have a multi-threaded client and multiple threads attempt to read from standard in at the same time, all the threads will not successfully obtain the user name and password information. Therefore, you cannot use the com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler implementation with a multi-threaded client where multiple threads might attempt to obtain data from standard in concurrently.

com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator
This interface is used to evaluate the level of trust for identity assertion. The default implementation is com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl, which enables you to define a list of trusted identities.

com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl
This default implementation enables you to define a list of trusted identities for identity assertion.

com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorException
This exception class is used by an implementation of the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator to communicate the exception and errors to the Web Services Security run time.

Default implementations for the JAX-WS run time

com.ibm.ws.wssecurity.wssapi.token.impl.CommonTokenGenerator
This implementation invokes the JAAS CallbackHandler and JAAS login configuration that are specified in the binding to create the SecurityToken at run time on the outbound SOAP message.

com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on the outbound SOAP message, and retrieves the X.509 certificate. The following properties may be specified:

- com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed. This property takes a boolean value, and the default value is `false`.
- com.ibm.wsspi.wssecurity.token.cert.useRequestorCert. This property takes a boolean value, and the default value is `false`.

com.ibm.ws.wssecurity.wssapi.token.impl.X509GenerateLoginModule
The wss.generate.x509 JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.X509GenerateLoginModule. X509GenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML Username token structure, and also a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the X.509 token at run time.

com.ibm.ws.wssecurity.wssapi.token.impl.PKCS7GenerateLoginModule
The wss.generate.pkcs7 JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.PKCS7GenerateLoginModule. PKCS7GenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML token structure and a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the token at run time.

com.ibm.ws.wssecurity.wssapi.token.impl.PkiPathGenerateLoginModule
The wss.generate.pkiPath JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.PkiPathGenerateLoginModule. PkiPathGenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML token structure and a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the token at run time.

com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on the outbound SOAP message, and it retrieves the binding configuration and user name and password authentication data. The following properties may be specified. These properties take a boolean value, and the default value is `false`.

- com.ibm.wsspi.wssecurity.token.username.addNonce
- com.ibm.wsspi.wssecurity.token.username.addTimestamp
- com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed
- com.ibm.wsspi.wssecurity.token.IDAssertion.useRunAsIdentity

- com.ibm.wsspi.wssecurity.token.IDAssertion.sendRealm
- com.ibm.wsspi.wssecurity.token.IDAssertion.trustedRealm

com.ibm.ws.wssecurity.wssapi.token.impl.UNTGenerateLoginModule
The wss.generate.unt JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl. UNTGenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML Username token structure and also a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the token at run time. When com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed has a the value of `true`, the generated username token does not contain a password. When com.ibm.wsspi.wssecurity.token.IDAssertion.sendRealm has the value of `true`, the user name is qualified by the local realm name. When com.ibm.wsspi.wssecurity.token.IDAssertion.trustedRealm has the value of `true`, the user name field contains both the user name and a registry-dependent unique identifier for the user. Both the user name and the unique identifier are qualified by the local realm name.

com.ibm.websphere.wssecurity.callbackhandler.KRBTokenGenerateCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on the outbound SOAP message, and it retrieves the Kerberos user name and password, along with other binding configuration properties. The following properties may be specified. The properties take a string that specifies the target service name as part of a service principal name (SPN), in the form of `service_name/host_name@Kerberos_realm_name`.

- com.ibm.wsspi.wssecurity.krbtoken.targetServiceName
- com.ibm.wsspi.wssecurity.krbtoken.targetServiceHost
- com.ibm.wsspi.wssecurity.krbtoken.targetServiceRealm

com.ibm.ws.wssecurity.wssapi.token.impl.KRBGenerateLoginModule
The wss.generate.KRB5BST JAAS system login configuration contains the classes com.ibm.ws.wssecurity.wssapi.token.impl.KRBGenerateLoginModule, and com.ibm.ws.wssecurity.wssapi.token.impl.DKTGenerateLoginModule. KRBGenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML token structure and a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the token at run time.

com.ibm.ws.wssecurity.wssapi.token.impl.DKTGenerateLoginModule
The wss.generate.KRB5BST JAAS system login configuration contains the classes com.ibm.ws.wssecurity.wssapi.token.impl.KRBGenerateLoginModule, and com.ibm.ws.wssecurity.wssapi.token.impl.DKTGenerateLoginModule. DKTGenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML token structure and a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the token at run time when the **Requires derived keys** option is enabled.

com.ibm.websphere.wssecurity.callbackhandler.LTPAGenerateCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on the outbound SOAP message, and it retrieves the user name and password binding data if they are specified.

com.ibm.ws.wssecurity.wssapi.token.impl.LTPAGenerateLoginModule
The wss.generate.ltpa JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.LTPAGenerateLoginModule. LTPAGenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML token structure and a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the token at run time. The security token contains an LTPA token that is generated from the user name and password if they are defined in the binding data, or the LTPA authentication token from the RunAs Subject, in that order.

com.ibm.ws.wssecurity.wssapi.token.impl.LTPAPropagationGenerateLoginModule
The wss.generate.ltpaProp JAAS system login configuration contains com.ibm.ws.wssecurity.wssapi.token.impl.LTPAPropagationGenerateLoginModule.

LTPAPropagationGenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML token structure and a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the token at run time. The security token contains the serialized RunAs Subject.

com.ibm.ws.wssecurity.impl.auth.callback.WSTrustCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on the outbound SOAP message, and it retrieves security context token configuration data.

com.ibm.ws.wssecurity.wssapi.token.impl.SCTGenerateLoginModule
The wss.generate.sct JAAS system login configuration contains the classes com.ibm.ws.wssecurity.wssapi.token.impl.SCTGenerateLoginModule, and com.ibm.ws.wssecurity.wssapi.token.impl.DKTGenerateLoginModule. SCTGenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML token structure and a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the security context token at run time.

com.ibm.ws.wssecurity.wssapi.token.impl.DKTGenerateLoginModule
The wss.generate.sct JAAS system login configuration contains the classes com.ibm.ws.wssecurity.wssapi.token.impl.SCTGenerateLoginModule, and com.ibm.ws.wssecurity.wssapi.token.impl.DKTGenerateLoginModule. DKTGenerateLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for generating an XML token structure and a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the token at run time when the **Requires derived keys** option is enabled.

com.ibm.ws.wssecurity.wssapi.token.impl.CommonTokenConsumer
This implementation invokes the JAAS CallbackHandler and JAAS login configuration that are specified in the binding to extract the security token from the inbound SOAP message and to create the SecurityToken object at run time.

com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on SOAP message inbound to retrieve the trust store and certificate file information that are required to validate the X.509 certificate.

com.ibm.ws.wssecurity.wssapi.token.impl.X509ConsumeLoginModule
The wss.consume.x509 JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.X509ConsumeLoginModule. X509ConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving and validating the X.509 certificate. It creates a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the X.509 token at run time.

com.ibm.ws.wssecurity.wssapi.token.impl.PKCS7ConsumeLoginModule
The wss.consume.pkcs7 JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.PKCS7ConsumeLoginModule PKCS7ConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving and validating the X.509 certificate. It creates a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the X.509 token at run time.

com.ibm.ws.wssecurity.wssapi.token.impl.PkiPathConsumeLoginModule
The wss.consume.pkiPath JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.PkiPathConsumeLoginModule. PkiPathConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving and validating the X.509 certificate. It creates a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the X.509 token at run time.

com.ibm.websphere.wssecurity.callbackhandler.UNTConsumeCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on SOAP message inbound to retrieve binding configuration data. The following properties may be specified. These properties take a boolean value and the default value is `false`.

- com.ibm.wsspi.wssecurity.token.username.verifyTimestamp
- com.ibm.wsspi.wssecurity.token.username.verifyNonce
- com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed
- com.ibm.wsspi.wssecurity.token.IDAssertion.trustedRealm
- com.ibm.wsspi.wssecurity.token.UsernameToken.disableUserRegistryCheck

com.ibm.ws.wssecurity.wssapi.token.impl.UNTConsumeLoginModule
The wss.consume.unt JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.UNTConsumeLoginModule. UNTConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving and validating the username token. It creates a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the username token at run time. When com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed has the value of `false`, UNTConsumeLoginModule validates the username and password against the local user registry. An incorrect user name or incorrect or missing password will cause the token validation to fail. When com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed has a value of `true`, and com.ibm.wsspi.wssecurity.token.IDAssertion.trustedRealm has a value of `false`, the user name is validated against the local user registry. There should be no password in the username token. When both com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed and com.ibm.wsspi.wssecurity.token.IDAssertion.trustedRealm have a value of `true`, the user name field must contain a realm-qualified user name and unique user identifier data, and the realm must be one of the trusted realms in the multiple security domain inbound trust configuration.

com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on the inbound SOAP message, and it retrieves the binding configuration data.

com.ibm.ws.wssecurity.wssapi.token.impl.KRBConsumeLoginModule
The wss.consume.KRB5BST JAAS system login configuration contains the classes com.ibm.ws.wssecurity.wssapi.token.impl.KRBConsumeLoginModule, and com.ibm.ws.wssecurity.wssapi.token.impl.DKTConsumeLoginModule. KRBConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving and validating the Kerberos AP_REQ token. It creates a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the AP_REQ token at run time.

com.ibm.ws.wssecurity.wssapi.token.impl.DKTConsumeLoginModule
The wss.consume.KRB5BST JAAS system login configuration contains the classes com.ibm.ws.wssecurity.wssapi.token.impl.KRBConsumeLoginModule, and com.ibm.ws.wssecurity.wssapi.token.impl.DKTConsumeLoginModule. DKTConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving the derived key when a derived key is required.

com.ibm.websphere.wssecurity.callbackhandler.LTPAConsumeCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on the inbound SOAP message, and it retrieves the binding configuration data.

com.ibm.ws.wssecurity.wssapi.token.impl.LTPAConsumeLoginModule
The wss.consume.ltpa JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.LTPAConsumeLoginModule. LTPAConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving and validating the LTPA v2 or LTPA token. It creates a com.ibm.websphere.wssecurity.wssapi.token.SecurityToken that represents the LTPA v2 or LTPA token at run time.

com.ibm.ws.wssecurity.wssapi.token.impl.LTPAPropagationConsumeLoginModule
The wss.consume.ltpaProp JAAS system login configuration contains the class com.ibm.ws.wssecurity.wssapi.token.impl.LTPAPropagationConsumeLoginModule. LTPAPropagationConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving, deserializing, and validating the propagation token and reconstructing the security context.

com.ibm.ws.wssecurity.impl.auth.callback.SCTConsumeCallbackHandler
This class implements the javax.security.auth.callback.CallbackHandler interface on the outbound SOAP message, and it retrieves the binding configuration data.

com.ibm.ws.wssecurity.wssapi.token.impl.SCTConsumeLoginModule
The wss.consume.sct JAAS system login configuration contains the classes com.ibm.ws.wssecurity.wssapi.token.impl.SCTConsumeLoginModule, and com.ibm.ws.wssecurity.wssapi.token.impl.DKTConsumeLoginModule. SCTConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving and validating the security context token.

com.ibm.ws.wssecurity.wssapi.token.impl.DKTConsumeLoginModule
The wss.consume.sct JAAS system login configuration contains the classes com.ibm.ws.wssecurity.wssapi.token.impl.SCTConsumeLoginModule, and com.ibm.ws.wssecurity.wssapi.token.impl.DKTConsumeLoginModule. DKTConsumeLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for retrieving the derived key when a derived key is required.

com.ibm.ws.wssecurity.impl.auth.module.PreCallerLoginModule
The wss.caller JAAS system login configuration contains the class com.ibm.ws.wssecurity.impl.auth.module.PreCallerLoginModule. PreCallerLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for validating whether it has received any security token that may be used to establish caller identity or trusted identity.

com.ibm.ws.wssecurity.impl.auth.module.UNTCallerLoginModule
The wss.caller JAAS system login configuration contains the class com.ibm.ws.wssecurity.impl.auth.module.UNTCallerLoginModule. UNTCallerLoginModule implements the javax.security.auth.spi.LoginModule interface. UNTCallerLoginModule also determines if the user identity is authorized to make an identity assertion if the username is configured to be a trusted identity, or if there is exactly one caller identity if the username token is configured to be a caller identity. It sets the validated caller and trusted identity into the shared state.

com.ibm.ws.wssecurity.impl.auth.module.X509CallerLoginModule
The wss.caller JAAS system login configuration contains com.ibm.ws.wssecurity.impl.auth.module.X509CallerLoginModule. X509CallerLoginModule implements the javax.security.auth.spi.LoginModule interface. X509CallerLoginModule checks to see if the user identity is authorized to make an identity assertion if the X509 token is configured to be a trusted identity, or if there is exactly one caller identity if the X509 token is configured to be a caller identity. It sets the validated caller and trusted identity into the shared state.

com.ibm.ws.wssecurity.impl.auth.module.LTPACallerLoginModule
The wss.caller JAAS system login configuration contains the class com.ibm.ws.wssecurity.impl.auth.module.LTPACallerLoginModule. LTPACallerLoginModule implements the javax.security.auth.spi.LoginModule interface. LTPACallerLoginModule also checks to see if the user identity is an authorized to make an identity assertion if the LTPA token is configured to be a trusted identity, or if there is exactly one caller identity if the LTPA token is configured to be a caller identity. It sets the validated caller and trusted identity into the shared state.

com.ibm.ws.wssecurity.impl.auth.module.LTPAPropagationCallerLoginModule
The wss.caller JAAS system login configuration contains the class com.ibm.ws.wssecurity.impl.auth.module.LTPAPropagationCallerLoginModule.

LTPAPropagationCallerLoginModule implements the javax.security.auth.spi.LoginModule interface. LTPAPropagationCallerLoginModule also checks to see if the user identity is an authorized to make an identity assertion if the propagation token is configured to be a trusted identity, or if there is exactly one caller identity if the propagation token is configured to be a caller identity. It sets the validated caller and trusted identity into the shared state.

com.ibm.ws.wssecurity.impl.auth.module.KRBCallerLoginModule
The wss.caller JAAS system login configuration contains com.ibm.ws.wssecurity.impl.auth.module.KRBCallerLoginModule. KRBCallerLoginModule implements the javax.security.auth.spi.LoginModule interface. KRBCallerLoginModule also checks to see if the user identity is an authorized to make an identity assertion if the Kerberos token is configured to be a trusted identity, or if there is exactly one caller identity if the Kerberos token is configured to be a caller identity. It sets the validated caller and trusted identity into the shared state.

com.ibm.ws.wssecurity.impl.auth.module.WSWSSLoginModule
The wss.caller JAAS system login configuration contains the class com.ibm.ws.wssecurity.impl.auth.module.WSWSSLoginModule. WSWSSLoginModule implements the javax.security.auth.spi.LoginModule interface and is responsible for asserting the caller identity to the ltpaLoginModule and the wsMapDefaultInboundLoginModule to establish the caller security context.

com.ibm.ws.security.server.lm.ltpaLoginModule
The wss.caller JAAS system login configuration contains the class com.ibm.ws.security.server.lm.ltpaLoginModule.

com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule
The wss.caller JAAS system login configuration contains the class com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule.

XML digital signature:

XML-Signature Syntax and Processing (XML digital signature) is a specification that defines XML syntax and processing rules to sign and verify digital signatures for digital content. The specification was developed jointly by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF).

XML digital signature does not introduce new cryptographic algorithms. WebSphere Application Server uses XML digital signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method.

XML canonicalization (c14n) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. For example, although their octet representations are different, the following examples are identical:

- `<person first="John" last="Smith"/>`
- `<person last="Smith" first="John"></person>`

C14n is a process that is used to canonicalize XML information. Select an appropriate c14n algorithm because the information that is canonicalized is dependent upon this algorithm. One of the major c14n algorithms, Exclusive XML Canonicalization, canonicalizes the character encoding scheme, attribute order, namespace declarations, and so on. The algorithm does not canonicalize white space outside tags, namespace prefixes, or data type representation.

XML signature in the Web Services Security-Core specification

The Web Services Security-Core (WSS-Core) specification defines a standard way for SOAP messages to incorporate an XML signature. You can use almost all of the XML signature features in WSS-Core except

enveloped signature and enveloping signature. However, WSS-Core has some recommendations such as exclusive canonicalization for the c14n algorithm and some additional features such as SecurityTokenReference and KeyIdentifier.

The KeyIdentifier is the value of the SubjectKeyIdentifier field within the X.509 certificate. For more information on the KeyIdentifier, see “Reference to a Subject Key Identifier” within the OASIS Web Services Security X.509 Certificate Token Profile documentation.

By including XML signature in SOAP messages, the following issues are realized:

Message integrity
A message receiver can confirm that attackers or accidents have not altered parts of the message after these parts are signed by a key.

Authentication
You can assume that a valid signature is proof of possession. A message with a digital certificate that is issued by a certificate authority and a signature in the message that is validated successfully by a public key in the certificate, is proof that the signer has the corresponding private key. The receiver can authenticate the signer by checking the trustworthiness of the certificate.

Collection certificate store:

A *collection certificate store* is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs is used to check the signature of a digitally signed SOAP message.

A collection certificate store is used when WebSphere Application Server is processing a received SOAP message. For JAX-RPC applications, this collection is configured in the Request Consumer Service Configuration Details section of the binding file for servers and in the Response Consumer Configuration section of the binding file for clients. You can configure these two sections using one of the assembly tools provided by WebSphere Application Server. See the assembly tools information in the topic Assembly tools.

For JAX-WS applications, this collection is configured using the administrative console in the Keys and certificates panel of the WS-Security policy set bindings.

A collection certificate store is one kind of certificate store. A certificate store is defined as javax.security.cert.CertStore in the Java CertPath application programming interface (API). The Java CertPath API defines the following types of certificate stores:

Collection certificate store
A collection certificate store accepts the certificates and CRLs as Java collection objects.

Lightweight Directory Access Protocol certificate store
The Lightweight Directory Access Protocol (LDAP) certificate store accepts certificates and CRLs as LDAP entries.

The CertPath API uses the certificate store and the trust anchor to validate the incoming X.509 certificate that is embedded in the SOAP message. The Web Services Security implementation in the WebSphere Application Server supports the collection certificate store. Each certificate and CRL is passed as an encoded file.

Certificate revocation list:

A *certificate revocation list* is a time-stamped list of certificates that have been revoked by a certificate authority (CA).

A certificate that is found in a certificate revocation list (CRL) might not be expired, but is no longer trusted by the certificate authority that issued the certificate. The certificate authority creates the CRL that contains the serial number and issuing CA distinguished name of the certificate that has been revoked. The CA might add the certificate to the certificate revocation list if it believes that the client certificate is compromised. The certificate revocation list is maintained and issued by the certificate authority.

XML encryption:

XML encryption is a specification that was developed by World Wide Web (WWW) Consortium (W3C) in 2002 and that contains the steps to encrypt data, the steps to decrypt encrypted data, the XML syntax to represent encrypted data, the information to be used to decrypt the data, and a list of encryption algorithms, such as triple DES, AES, and RSA.

You can apply XML encryption to an XML element, XML element content, and arbitrary data, including an XML document. For example, suppose that you need to encrypt the <CreditCard> element that is shown in example 1.

Example 1: Sample XML document:

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Example 2: XML document with a common secret key:

Example 2 shows the XML document after encryption. The <EncryptedData> element represents the encrypted <CreditCard> element. The <EncryptionMethod> element describes the applied encryption algorithm, which is triple DES in this example. The <KeyInfo> element contains the information that is needed to retrieve a decryption key, which is a <KeyName> element in this example. The <CipherValue> element contains the cipher text that is obtained by serializing and encrypting the <CreditCard> element.

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <EncryptionMethod
       Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
       <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
         <KeyName>John Smith</KeyName>
       </KeyInfo>
       <CipherData>
         <CipherValue>ydUNqHkMrD...</CipherValue>
       </CipherData>
  </EncryptedData>
</PaymentInfo>
```

Example 3: XML document encrypted with the public key of the recipient:

In example 2, it is assumed that both the sender and recipient have a common secret key. If the recipient has a public and private key pair, which is commonly the case, the <CreditCard> element can be encrypted as shown in example 3. The <EncryptedData> element is the same as the <EncryptedData> element found in Example 2. However, the <KeyInfo> element contains an <EncryptedKey> element.

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <EncryptionMethod
      Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
    <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
```

```
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
      </EncryptedKey>
    </KeyInfo>
    <CipherData>
      <CipherValue>ydUNqHkMrD...</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

XML Encryption in the WSS-Core:

The WSS-Core specification is under development by Organization for the Advancement of Structured Information Standards (OASIS). The specification describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. The message confidentiality is realized by encryption based on XML Encryption.

The WSS-Core specification supports encryption of any combination of body blocks, header blocks, their substructures, and attachments of a SOAP message. When you encrypt parts of a SOAP message, the specification also requires that you prepend a reference from the security header block to the encrypted parts of the message. The reference can be a clue for a recipient to identify which encrypted parts of the message to decrypt.

The XML syntax of the reference varies according to what information is encrypted and how it is encrypted. For example, suppose that the <CreditCard> element in example 4 is encrypted with either a common secret key or the public key of the recipient.

Example 4: Sample SOAP Version 1.1 message:

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <CreditCard Limit='5,000' Currency='USD'>
        <Number>4019 2445 0277 5567</Number>
        <Issuer>Example Bank</Issuer>
        <Expiration>04/02</Expiration>
      </CreditCard>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Version 1.2 does not support `encodingStyle` so the example changes to the following:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <CreditCard Limit='5,000' Currency='USD'>
        <Number>4019 2445 0277 5567</Number>
        <Issuer>Example Bank</Issuer>
        <Expiration>04/02</Expiration>
      </CreditCard>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The resulting SOAP messages are shown in Examples 5 and 6. In these example, the <ReferenceList> and <EncryptedKey> elements are used as references, respectively.

Example 5: SOAP Version 1.1 message encrypted with a common secret key

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <ReferenceList xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <DataReference URI='#ed1'/>
      </ReferenceList>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>John Smith</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Version 1.2 does not support `encodingStyle` and the example changes to the following:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <ReferenceList xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <DataReference URI='#ed1'/>
      </ReferenceList>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>John Smith</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 6: SOAP message encrypted with the public key of the recipient:

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
        <ReferenceList>
```

```
        <DataReference URI='#ed1'/>
      </ReferenceList>
    </EncryptedKey>
  </Security>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
  <PaymentInfo xmlns='http://example.org/paymentv2'>
    <Name>John Smith</Name>
    <EncryptedData Id='ed1'
      Type='http://www.w3.org/2001/04/xmlenc#Element'
      xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <EncryptionMethod
        Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
      <CipherData>
        <CipherValue>ydUNqHkMrD...</CipherValue>
      </CipherData>
    </EncryptedData>
  </PaymentInfo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Version 1.2 does not support `encodingStyle` and the example changes to the following:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI='#ed1'/>
        </ReferenceList>
      </EncryptedKey>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Relationship to digital signature:

The WSS-Core specification also provides message integrity, which is realized by a digital signature that is based on the XML-Signature specification.

A combination of encryption and digital signature over common data introduces cryptographic vulnerabilities.

Symmetric versus asymmetric encryption

For XML encryption, the application server supports two types of encryption:

- *Symmetric encryption*

  In releases of the application server prior to WebSphere Application Server Version 7, including the IBM WebSphere Application Server Version 6.1 Feature Pack for Web Services, by default the KeyName

reference was used to refer to the shared key outside of the SOAP message. However, the Web Services Security (WS-Security) Version 1.1 standard does not recommend using the KeyName reference. Because KeyName is not supported by the security policy, it is not supported in the application server.

The Web Services Secure Conversation (WS-SecureConversation) standard defines how to exchange the shared key between the client and the service and how to refer to the shared key in the message. The use of Kerberos with Web Services Security, as described in the Kerberos Token Profile, also defines how to use a Kerberos session key or key derived from the session key to perform symmetric encryption. Therefore, you can use symmetric encryption by using WS-SecureConversation or Kerberos. WebSphere Application Server supports DerivedKeyToken when using WS-SecureConversation. When using Kerberos, WebSphere Application Server supports both the use of DerivedKeyToken and the use of the Kerberos session key directly.

- *Asymmetric encryption*

  For asymmetric encryption, XML Encryption introduces the idea of key wrapping. The data, such as the contents of the SOAP body element, is encrypted with a shared key that is dynamically generated while processing. Then, the generated shared key is encrypted with the public key of the receiver. WebSphere Application Server supports the X509Token for asymmetric encryption.

Security token:

Web Services Security provides a general-purpose mechanism to associate security tokens with messages for single message authentication. A security token represents a set of claims made by a client that might include a name, password, identity, key, certificate, group, privilege, and so on.

A specific type of security token is not required by Web Services Security. Web Services Security is designed to be extensible and support multiple security token formats to accommodate a variety of authentication mechanisms. For example, a client might provide proof of identity and proof of a particular business certification. However, the security token usage for Web Services Security is defined in separate profiles such as the Username token profile, the X.509 token profile, the Security Assertion Markup Language (SAML) token profile, the eXtensible rights Markup Language (XrML) token profile, the Kerberos token profile, and so on.

A security token is embedded in the SOAP message within the SOAP header. The security token within the SOAP header is propagated from the message sender to the intended message receiver. On the receiving side, the WebSphere Application Server Web Services Security handler authenticates the security token and sets up the caller identity on the running thread.

WebSphere Application Server contains an enhanced security token that has the following features:

- The client can send multiple tokens to downstream servers.
- The receiver can determine which security token to use for authorization based upon the type or signed part for X.509 tokens.
- You can use the custom token or derived key token for digital signing or encryption.

LTPA and LTPA Version 2 tokens:

Web services security supports both LTPA (Version 1) and LTPA Version 2 (LTPA2) tokens. The LTPA2 token, which is more secure than Version 1, is supported by the JAX-WS runtime only.

Note: The support statements in this topic apply to the web services security implementation for WebSphere Application Server and not the security implementation for non-web services functionality.

The Lightweight Third Party Authentication (LTPA) token is a specific type of binary security token. The web services security implementation for WebSphere Application Server, Version 5 and later supports the LTPA Version 1 token. WebSphere Application Server Version 7 and later supports the LTPA Version 2 token using the JAX-WS runtime environment.

Although the same LTPAToken assertion is used in the policy for both LTPA Version 1 and LTPA Version 2, the valuetype value for the Version 2 token is different than Version 1. The valuetype value is composed of the URI and the local name. The following table shows the valuetype values for the LTPA token versions when they are selected as the token type for the policy set bindings. These values are not editable.

Table 32. LTPA token versions and their valuetype values. This table lists the valuetype values for both LTPA (Version 1) and LTPA2 tokens.

LTPA Version token	Valuetype value
LTPA (Version 1)	`http://www.ibm.com/websphere/appserver/tokentype/5.0.2/LTPA`
LTPA2	`http://www.ibm.com/websphere/appserver/tokentype/LTPAv2`

To allow for interoperability between servers that are running different versions of WebSphere Application Server, by default, the JAX-WS web services security runtime in Version 7.0 and later can successfully consume an LTPA Version 1 token when the binding is configured to expect an LTPA2 token. However, you can configure the binding for the JAX-WS runtime to accept only LTPA2 tokens. For more information, see the documentation about Authentication generator or consumer token settings.

If the web services security run time receives a token with a unrecognized valuetype value and the SOAP security header contains a mustUnderstand attribute value that is equal to `'1'`, the web services security run time issues a SOAPFaultException error. If the mustUnderstand attribute value is equal to `'0'`, the token is ignored.

If an LTPA2 token is sent with a mustUnderstand attribute value that is equal to `'1'` to a web services security run time in which the LTPA2 token is not supported, the run time does not recognize the LTPAv2 valuetype value. Thus, the receiving run time issues a SOAPFaultException error. The following table illustrates these different configurations and their potential error messages..

Table 33. LTPA token configurations. This table lists whether the LTPA Version 1 token is optional or required, lists the associated mustUnderstand attribute value, lists its run time, and provides the resulting SOAPFaultException error, if applicable

Run time	LTPA Version 1 token status	MustUnderstand attribute value	SOAPFaultException error
JAX-RPC	Required	1	`com.ibm.wsspi.wssecurity.SoapSecurityException: WSEC5509E: A security token whose type is [{http://www.ibm.com/websphere/appserver/tokentype/5.0.2}LTPA] is required.`
JAX-RPC	Required	0	`com.ibm.wsspi.wssecurity.SoapSecurityException: WSEC5509E: A security token whose type is [{http://www.ibm.com/websphere/appserver/tokentype/5.0.2}LTPA] is required.`
JAX-RPC	Optional	1	`com.ibm.wsspi.wssecurity.SoapSecurityException: WSEC5502E: Unexpected element as the target element: s:BinarySecurityToken.`
JAX-RPC	Optional	0	None
JAX-RPC	Not Configured	1	`com.ibm.wsspi.wssecurity.SoapSecurityException: WSEC5502E: Unexpected element as the target element: s:BinarySecurityToken.`
JAX-RPC	Not Configured	0	None
JAX-WS (Version 6.1 Feature Pack for Web Services)	Not Configured	1	`CWWSS5502E: The target element: s:BinarySecurityToken was not expected.`
JAX-WS (Version 6.1 Feature Pack for Web Services)	Not Configured	0	None

Table 33. LTPA token configurations (continued). This table lists whether the LTPA Version 1 token is optional or required, lists the associated mustUnderstand attribute value, lists its run time, and provides the resulting SOAPFaultException error, if applicable

Run time	LTPA Version 1 token status	MustUnderstand attribute value	SOAPFaultException error
JAX-WS (Version 6.1 Feature Pack for Web Services)	Configured	1	`CWWSS5509E: A security token whose type is [{http://www.ibm.com/websphere/appserver/tokentype/5.0.2}LTPA] is required.`
JAX-WS (Version 6.1 Feature Pack for Web Services)	Configured	0	`CWWSS5509E: A security token whose type is [{http://www.ibm.com/websphere/appserver/tokentype/5.0.2}LTPA] is required.`

You can configure the JAX-WS run time to generate either LTPA (Version 1) or LTPA2 tokens. If you configure the LTPA token generator in a policy binding to generate an LTPA (Version 1) token, you must do one of the following:

- Enable the single sign-on interoperability mode, which is available on the Single sign-on (SSO) panel within the administrative console. For more information on this option, see the documentation about single sign-on settings.
- Set the `com.ibm.wsspi.wssecurity.tokenGenerator.ltpav1.pre.v7` custom property to true for the LTPA token generator.

If you do not perform at least one of the steps previously indicated, an error occurs when the application, which is attached to these bindings, is started.

Username token:

You can use the <UsernameToken> element to propagate a user name and, optionally, password information. Also, you can use this token type to carry basic authentication information. Both a user name and a password are used to authenticate the SOAP message.

OASIS: Web Services Security UsernameToken Profile 1.0

A UsernameToken element containing the user name is used in identity assertion. Identity assertion establishes the identity of the user based on the trust relationship.

The following example shows the syntax of the <UsernameToken> element:

```
<wsse:UsernameToken wsu:Id="Example-1">
   <wsse:Username>
   ...
   </wsse:Username>
   <wsse:Password Type="...">
   ...
   </wsse:Password>
   <wsse:Nonce EncodingType="...">
   ...
   </wsse:Nonce>
   <wsu:Created>
   ...
   </wsu:Created>
</wsse:UsernameToken>
```

The Web Services Security specification defines the following password types:

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText (default)
: This type is the actual password for the user name.

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest
: The type is the digest of the password for the user name. The value is a base64-encoded SHA1 hash value of the UTF8-encoded password.

WebSphere Application Server supports the default PasswordText type. However, it does not support password digest because most user registry security policies do not expose the password to the application software.

The following example illustrates the use of the <UsernameToken> element:

```
<S:Envelope
       xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
    <S:Header>
           ...
        <wsse:Security>
            <wsse:UsernameToken>
                <wsse:Username>Joe</wsse:Username>
                <wsse:Password>ILoveJava</wsse:Password>
            </wsse:UsernameToken>
        </wsse:Security>
    </S:Header>
</S:Envelope>
```

OASIS: Web Services Security UsernameToken Profile 1.1

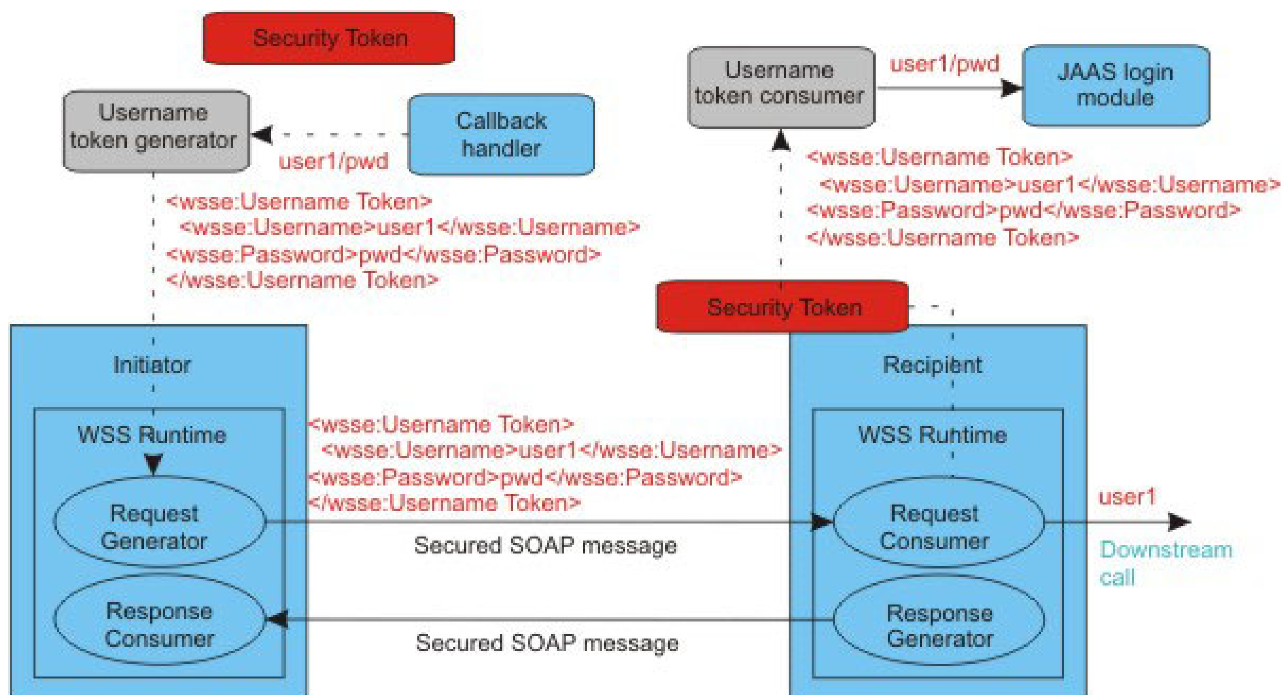
WebSphere Application Server supports both Username Token Profile 1.0 and Version 1.1 standards.

WebSphere Application Server does not support the following functions:

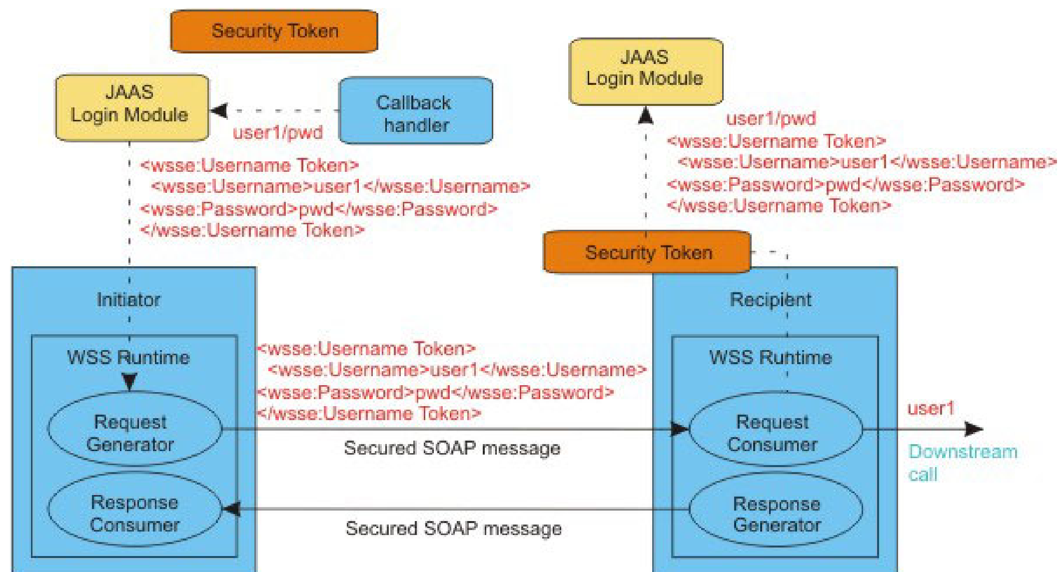
- In both versions of the Username Token Profile specification, the digest password type is not supported
- In both versions of the Username Token Profile specification, key derivation based on a password is not supported.

You can use policy sets to configure the UsernameToken using the administrative console. Also, you can use the Web Services Security APIs to attach the Username token to the SOAP message. The following figure describes the creation and validation of the Username token for the JAX-RPC and the JAX-WS programming models.

Creating and validating the Username token using the JAAS Login Module and the JAAS CallbackHandler in JAX-RPC



Creating and validating the Username token using the JAAS Login Module and the JAAS CallbackHandler in JAX-WS



Note: The WSS API is available only when you are using the Java API for XML-Based Web Services (JAX-WS) programming model.

On the generator side, the Username token is created by using the JAAS LoginModule and by using the JAAS CallbackHandler to pass the authentication data. The JAAS LoginModule creates the UsernameToken object and passes it to the Web Services Security run time.

On the consumer side, the Username Token XML format is passed to the JAAS LoginModule for validation or authentication, and the JAAS CallbackHandler is used to pass the authentication data from the Web Services Security run time to the JAAS LoginModule. After the token is authenticated, a UsernameToken object is created and is passed to the Web Service Security run time.

The following example provides sample code for creating Username tokens:

```
WSSFactory factory = WSSFactory.getInstance();
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Attach the username token to the message.
   UNTGenerationCallbackHandler ugCallbackHandler =
      newUNTGenerationCallbackHandler("alice", "ecila");
   SecurityToken ut = factory.newSecurityToken(ugCallbackHandler,
                                               UsernameToken.class);
   gencont.add(ut);

// Generate the WS-Security header
gencont.process(msgctx);
```

XML token:

XML tokens are offered in two well-known formats called Security Assertion Markup Language (SAML) and eXtensible rights Markup Language (XrML).

In WebSphere Application Server Versions 6 and later, you can plug in your own implementation. By using extensibility of the `<wsse:Security>` header in XML-based security tokens, you can directly insert these security tokens into the header. SAML assertions are attached to Web Services Security messages using web services by placing assertion elements inside the `<wsse:Security>` header. The following example illustrates a Web Services Security message with a SAML assertion token.

```
<S:Envelope xmlns:S="...">
<S:Header>
      <wsse:Security xmlns:wsse="...">
```

```
        <saml:Assertion
                 MajorVersion="1"
                 MinorVersion="0"
                 AssertionID="SecurityToken-ef375268"
                    Issuer="elliotw1"
                    IssueInstant="2002-07-23T11:32:05.6228146-07:00"
                  xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
                  ...
         </saml:Assertion>
     </wsse:Security>
 </S:Header>
 <S:Body>
 ...
 </S:Body>
</S:Envelope>
```

For a complete list of the supported standards and specifications, read about web services specifications and APIs.

Binary security token:

The ValueType attribute identifies the type of the security token, for example, a Lightweight Third Party Authentication (LTPA) token. The EncodingType type indicates how the security token is encoded, for example, Base64Binary. The BinarySecurityToken element defines a security token that is binary encoded. The encoding is specified using the EncodingType attribute. The value type and space are specified using the ValueType attribute. The Web Services Security implementation for WebSphere Application Server, Version 6 and later supports LTPA,, LTPA version 2, and X.509 certificate binary security tokens.

A binary security token has the following attributes that are used for interpretation:

- Value type
- Encoding type

The following example depicts an LTPA binary security token in a Web services security message header:

```
&ltwsse:BinarySecurityToken xmlns:wsst=
     "http://www.ibm.com/websphere/appserver/tokentype/5.0.2"
     EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary"
     ValueType="wsst:LTPA"&gt;
           MIZ6LGPt2CzXBQfio9wZTo1VotWov0NW3Za6lU5K7Li78DSnIK6iHj3hxXgrUn6p4wZI
           8Xg26havepvmSJ8XxiACMihTJuh1t3ufsrjbFQJOqh5VcRvI+AKEaNmnEgEV65jUYAC9
           C/iwBBWk5U/6DIk7LfXcTT0ZPAd+3D3nCS0f+6tnqMou8EG9mtMeTKccz/pJVTZjaRSo
           msu0sewsOKfl/WPsjW0bR/2g3NaVvBy18VlTFBpUbGFVGgzHRjBKAGo+ctkl80nlVLIk
           TUjt/XdYvEpOr6QoddGi4okjDGPyyoDxcvKZnReXww5UsoqlpfXwN4KG9as=
&lt/wsse:BinarySecurityToken&gt;
</wsse:Security>
</soapenv:Header>
```

As shown in the example, the token is Base64Binary encoded.

The following example depicts an LTPA version 2 binary security token:

```
<wsse:BinarySecurityToken
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:wsst="http://www.ibm.com/websphere/appserver/tokentype"
  wsu:Id="ltpa_20"
  ValueType="wsst:LTPAv2">
    bRYI0Z59k/P1gIkgSaxeJIQoI1BdojxjdoD+6qMmiH37lqS6U90Wx6EArMA05FHVyTmxvIJACGD
    UVfqVcPDQCdPlWAn9Brhz/bXw9OEVx0wx/eNYQuiBvEVNam7urd8SxZkqppOZyeN6APZ4Z4Rox0M
    jqQv9lFIB/AKBpJyaK8V9Z9gFO8k6J5HmE/G9jdBov9Su6hXlfF50Bhy6tx8BEm4Zn/pkeNc1H1d+
    tOxwDOfS0ORWH0tjzDCTFpAMPjMmfR0/o7o3DivONtZG61ylbcwB4hx01iQC/FN5DJwrEy8kCwCeF
    ywubKVVt5pyM1k6uVXI8ik5Pjf9aU1ei86y5iXc9CirhvqosXiZvjObHTYKZSjtGiMYw3q9NKbZxs
    SzfCuAdht8sjGfaVo43i0iz7CuFYAywqVldUPjwSTvCGNtmWB/3MRtBDrmq3fqYSomjw5ZWDFex/n
    98ZaOz8mUjNHinJc4APTtEx6S10CxUkUc8b8hoCdqbcOGdZcGqYF7xgcFXvsezsXw0eRmhra54x6g
    CJs1skMMNvi0vF2pic1cg4GClQ74NKxV1oTrDZPaQPTikYGJOLKHBPYnbPda0hPkX+iCOYN0IIRBa
    Vwjj1T0G+Y/MgokiNJRGwUQ7VHXEo0+Q2HsmCkmAFrIp4lZc9fGcFyVY/EUBBpkGchL0eKNv4DoVJW
    6EhFXWZdeiVk8
</wsse:BinarySecurityToken>
```

The following example depicts an X.509 v3 binary security token containing an X.509 public certificate:

```
<wsse:BinarySecurityToken
  EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary"
  ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
  wsu:Id="x509bst_12"
```

```
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
    MIIDQDCCAqmgAwIBAgICAQUwDQYJKoZIhvcNAQEFBQAwTjELMAkGA1UEBhMCSlAxETAPBg
    NVBAgTCEthbmFnYXdhMQwwCgYDVQQKEwNJQk0xDDAKBgNVBAsTA1RSTDEQMA4GA1UEAxMH
    SW50IENBMjAeFw0wMTEwMDExMDAwMzlaFw0xMTEwMDExMDAwMzlaMFMxCzAJBgNVBAYTAk
    pQMREwDwYDVQQIEwhLYW5hZ2F3YTEMMAoGA1UEChMDSUJNMQwwCgYDVQQLEwNUUkwxFTAT
    BgNVBAMTDFNPQVBQcm92aWRlcjCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAraakNJ
    1JzkPUuvPdXRvPOOCl12nBwmqvt65dk/x+QzxxarDNwH+eWRbLyyKcrAyd0XGV+Zbvj6V3
    O9DSVCZUCJttw6bbqqeYhwAP3V8s24sID77tk3gOhUTEGYxsljX2orL26SLqFJMrvnvk2F
    RS2mrdkZEBUG97mD4QWcln4d0CAwEAAaOCASYwggEiMAkGA1UdEwQCMAAwCwYDVR0PBAQD
    AgXgMCwGCWCGSAGG+EIBDQQfFh1PcGVuU1NMIEdlbmVyYXRlZCBDZXJ0aWZpY2F0ZTAdBg
    NVHQ4EFgQUlXSsrVRfZOLGdJdjEIwTbuSTe4UwgboGA1UdIwSBsjCBr4AUvfkg1Tj5ZHLT
    29p/3M6w/tC872+hgZKkgY8wgYwxCzAJBgNVBAYTAkpQMREwDwYDVQQIEwhLYW5hZ2F3YT
    EPMA0GA1UEBxMGWWFtYXRvMQwwCgYDVQQKEwNJQk0xDDAKBgNVBAsTA1RSTDEZMBcGA1UE
    AxMQU09BUCAyLjEgVGVzdCBDQTEiMCAGCSqGSIb3DQEJARYTbWFydXlhbWFAanAuaWJtLm
    NvbYICAQEwDQYJKoZIhvcNAQEFBQADgYEAXE7mE1RPb3lYAYJFzBb3VAHvkCWa/HQtCOZd
    yniCHp3MJ9EbNTq+QpOHV60YE8u0+5SejCzFSOHOpyBgLPjWoz8JXQnjV7VcAbTglw+ZoO
    SYy64rfhRdr9giSs47F4D6woPsAd2ubg/YhMaXLTSyGxPdV3VqQsutuSgDUDoqWCA=
</wsse:BinarySecurityToken>
```


X.509 Binary Security Token:

An X.509 binary security token is the base64 encoded representation of an X.509 public certificate.

The following table describes the X.509 token type.

X.509 token type	Description
X.509 version 1	Contains just the X.509 public certificate.
X.509 version 3	Contains just the X.509 public certificate.
PKIPath	Contains an ordered list of X.509 public certificates packaged in a PKIPath. The X509PKIPathv1 token type may be used to represent a certificate path.
PKCS7	Contains a list of X.509 certificates and, optionally, certificate revocation lists (CRLs) packaged in a PKCS#7 wrapper. The PKCS7 token may be used to represent a certificate path.

X.509 tokens are generally used to protect a SOAP message with XML Digital Signature or XML Encryption. Although not recommended, an X.509 token can also be used as an authentication token.

Using X.509 tokens for Authentication

When you authenticate a token, you are verifying that the sender of a token is who he says he is. You take a piece of public information that is sent in the message, such as a a user id, and verify it somehow with a piece of private information that only they can provide, such as a password.

As a very simple example, when you authenticate a UsernameToken, the user name and password are passed in the SOAP message and they are checked against the user registry at the endpoint.

For an X.509 certificate, the public information is the public key/DN and the private information is the private key. Unlike the password for a UsernameToken, the private key is not sent in the message.

When an X.509 token is used to sign a message, the following process is used:

1. If trust is enabled, the certificate is evaluated against the trust store and cert store, if configured. This will catch trust errors, certificate chaining errors, revocation errors, certificate expiration, etc. For example, you can have specific DNs in the trust store to trust each certificate explicitly, or just the root CA to trust all certificates issued from this CA, but no others.
2. The runtime verifies that the sender of the message has the private key associated with the certificate by verifying the signature. If the signature cannot be verified, then one of the following conditions occurred:
   a. The message was signed with a private key that did not match the public key in the message.

b. The message was modified after it was sent.

After the signature is verified, you know that the sender of the message is the holder of the private key; you know that he is who he says he is.

If you pass an X.509 token in a message without using its private key to sign the message, you will not perform step #2. You will not verify that the sender of the message is the holder of the private key, or he is who he says he is. When you sign a message, you are doing something that only the holder of the private key can do.

With an unprotected X.509 token, that is, the x.509 token was not used to sign the message, the system can be compromised in the following way:

Capture of the valid message and substitution of an attacker's X.509 token in the same message.

It is not recommend that an unprotected X.509 token be sent in a message. If an X.509 token is to be used for authentication, it is recommend that one X.509 token be sent in the message and that X.509 token be used for both Digital Signature and authentication, with a caller configuration. The signing part reference is used in the caller settings.

The following table describes the X.509 token value types.

X.509 token type	Value type
X.509 Version 1	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509
X.509 Version 3	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3
PKIPath	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1
PKCS7	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

Kerberos token:

IBM WebSphere Application Server provides Kerberos token support for web services message-level security. The support is based on the Organization for the Advancement of Structured Information Standards (OASIS) Web Services Security Kerberos Token Profile Version 1.1. Use this topic to understand the Kerberos support that is available for web services.

Kerberos token profile version 1.1

Kerberos Version 5 is a mature, open standard that provides a secure third-party authentication mechanism. The OASIS Web Services SOAP Message Security specification references the Kerberos token in the SOAP message. Web services applications can use the Kerberos token to send identities and protect messages more securely. Overall, Kerberos support involves Kerberos support in Java Platform, Enterprise Edition (Java EE) security and the Kerberos token support in Web Services Security. This topic covers the Kerberos token support in Web Services Security only.

In WebSphere Application Server Version 7.0 and later, Web Services Security supports the Kerberos token, which is based on OASIS WS-Security Kerberos Token Profile Version 1.1 specification. The Kerberos token is a binary security token for web services message-level security. Web Services Security provides SOAP message-level security, such as security token propagation, message signature, and message encryption. The Kerberos token is used for message security, specifically with the SOAP message security specification for web services, and is another supported token, such as the username token and the secure conversation token.

For more information, see the Web Services Security Kerberos Token Profile Version 1.1 specification. The specification explains how to use Kerberos security with the Web Services Security and how the Kerberos token is propagated and used to secure the SOAP message through signing and encryption.

Kerberos token profile enablement

The WebSphere Application Server configuration model leverages existing tools and frameworks for the Kerberos token profile configuration of authentication and message protection, such as:

- Policy set and binding configuration to enable the Kerberos token profile for Java API for XML-Based Web Services (JAX-WS) applications
- Deployment descriptor and binding configuration to enable the Kerberos token profile for JAX-RPC applications
- Token profile enablement with a Kerberos token for JAX-WS applications
- Minimal client configuration to enable the Kerberos token profile using the JAX-WS programming model

For JAX-WS client applications, the design updates the application programming interfaces (APIs) for Web Services Security and enforces a Web Services Security policy with a Kerberos token, which is based on the OASIS token profile. To enable a Kerberos token profile by using a policy set, you must first establish the Web Services Security policy and binding files by using a custom token. For more information, see the "Kerberos configuration models for web services" topic.

Kerberos support

The following Kerberos-related function is supported by web services in WebSphere Application Server:

- Client programming models for JAX-WS applications with Web Services Security APIs
- Interoperability with Web Services Enhancements (WSE) Version 3.5 and Windows Communication Foundation (WCF) Version 3.5 for Microsoft .NET
- Recovery of web services message security tokens for JAX-WS applications
- Kerberos token profile enablement
- Integration with the base security for the application server
- Kerberos token generation for the client and service
- Kerberos consumption at the service
- Clustering and high-availability for JAX-WS applications
- Kerberos token profile configuration of authentication and message protection for JAX-WS applications
- Integration in a single realm with either a Microsoft or z/OS operating system Key Distribution Center (KDC).
- Kerberos token profile configuration of authentication for JAX-RPC applications
-

The application server does not support the following function:

- Key name references
- Message protection using session keys for JAX-RPC applications
- Message protection using derived keys for JAX-RPC applications
- Generation of SHA1 keys for JAX-RPC applications
- Kerberos delegation is not supported when you are using JAX-RPC applications configured with the Kerberos authentication security mechanism
- A Kerberos token is not recoverable when JAX-WS applications are enabled with web services Reliable Messaging

Kerberos message protection for web services:

Message-level security is based on the Organization for the Advancement of Structured Information Standards (OASIS) Web Services Security Kerberos Token Profile Version 1.1 specification. Use this topic to gain an overall understanding of how message protection is implemented with a Kerberos token for web services.

Message protection

The application server can interoperate with other web services technology because of the implementation of the OASIS web services Kerberos token profile. This specification defines the standards for securing a SOAP message with the Kerberos token. However, mutual authentication is not defined by the token profile. The OASIS Web Services SOAP Message Security specification describes how to secure a SOAP message through signing and encryption by using and referencing a Kerberos token. Specifically, the OASIS specification defines how the Kerberos token, as a wrapped or unwrapped AP_REQ packet, is encoded and attached to the SOAP message. The token that is described in the OASIS Kerberos token profile is limited to the AP_REQ packet, which consists of a service ticket and an authenticator. The AP_REQ packet is obtained from the Key Distribution Center (KDC), which serves as the third-party authentication service.

Multiple formats exist for the Kerberos token, as defined in the OASIS Web Services Security Kerberos Token Profile 1.1. The `@ValueType` attribute is used to specify the token format. You must specify one of the following `<@ValueType>` attributes for the element:

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

The resulting AP_REQ token can be either GSS-API framed (wrapped) or raw (unwrapped). The token must be Base-64 encoded.

Kerberos usage overview for web services:

You can use a Kerberos token to complete similar functions that you might currently complete with other binary security tokens, such as Lightweight Third Party Authentication (LTPA) and Secure Conversation tokens.

Token generator

After the Kerberos token is created from the Key Distribution Center (KDC), the Web Services Security generator encodes and inserts the token into the SOAP message and propagates the token for token consumption or acceptance. If a message integrity or confidentiality key is required, a Kerberos sub-key or a Kerberos session key from the Kerberos ticket is used. A key can be derived from either the Kerberos sub-key or the Kerberos session key. Web Services Security uses the key from the Kerberos token to sign and encrypt the message parts as described in the OASIS Web Services Security Kerberos Token Profile Version 1.1 specification. The type of key to use is predetermined by the Web Services Security configuration or policy. Also, the size of the derived key is configurable.

The value of the signature or encryption key is constructed from the value of one of the following keys:

- The Kerberos sub-key when it is present in the authenticator
- A session key directly from the ticket if the sub-key is absent
- A key that is derived from either of the previous keys

When the Kerberos token is referenced as a signature key, the signature algorithm must be a hashed message authentication code, which is http://www.w3.org/2000/09/xmldsig#hmac-sha1. When the Kerberos token is referenced as an encryption key, you must use one of the following symmetric encryption algorithms:

- http://www.w3.org/2001/04/xmlenc#aes128-cbc
- http://www.w3.org/2001/04/xmlenc#aes256-cbc
- http://www.w3.org/2001/04/xmlenc#tripledes-cbc

Attention:

- The Application Server supports Kerberos Version 5 only.
- You can use a AES-type symmetric algorithm suite in Web Services Security when the Kerberos ticket complies with RFC-4120 only.
- A Kerberos key with the RC4-HMAC 128–bit key type only is used when the KDC is on a Microsoft Windows 2003 server.
- A Kerberos key with AES 128–bit or 256–bit key types is used when the KDC is on a Microsoft Windows 2008 server.
- A Kerberos ticket must be forwardable and not contain an address when the service provider is running in a cluster.
- You must import an unrestricted Java security policy when you use an AES 256–bit encryption algorithm.

For information about using a Kerberos token in a cross or trusted realm environment, read the topic "Kerberos token security in a single, cross, or trusted realm environment."

Token consumer

The Web Services Security consumer receives and extracts the Kerberos token from the SOAP message. The consumer then accepts the Kerberos token by validating the token with its own secret key. The secret key of the service is stored in an exported keytab file. After acceptance, the Web Services Security consumer stores the associated request token information into the context Subject. You can also derive the corresponding key to the request token. The key is used to verify and decrypt the message. If the request token is forwardable and does not contain an address, the application server can use the stored token for downstream calls.

Token format and reference

For JAX-WS applications, use the existing custom policy set or administrative command scripts for the custom policy to specify the Kerberos token type, the message signing, and message encryption. The JAX-WS programming model for WebSphere Application Server provides minimal configuration to enable the Kerberos token profile with the Kerberos token.

For JAX-RPC applications, use the deployment descriptor to specify that the custom token use the Kerberos token. You can use the Kerberos token for authentication, but you cannot use it for message signing or encryption.

WebSphere Application Server supports the following callback handler classes for the Kerberos Version 5 token:

- com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler

  This class is a callback handler for Kerberos Version 5 token on the consumer side. This instance is used to generate the WSSVerification and WSSDecryption objects to validate a Kerberos binary security token.
- com.ibm.websphere.wssecurity.callbackhandler.KRBTokenGenerateCallbackHandler

This class is a callback handler for Kerberos Version 5 token on the generator side. This instance is used to generate the WSSSignature object and the WSSEncryption object to generate a Kerberos binary security token.

The OASIS Web Services Security Kerberos Token Profile Version 1.1 specification states that the Kerberos token is attached to the SOAP message with the `<wsse:BinarySecurityToken>` element. The following example shows the message format. The boldface type shows delineates the binary security token information from the other parts of the example.

```
<S11:Envelope xmlns:S11="..." xmlns:wsu="...">
   <S11:Header>
      <wsse:Security xmlns:wsse="...">
          <wsse:BinarySecurityToken
              EncodingType="http://docs.oasis-open.org/wss/2004/01/
                 oasis-200401-wss-soap-message-security-1.0#Base64Binary"
              ValueType="http://docs.oasis-open.org/wss/
                 oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ"
              wsu:Id="MyToken">boIBxDCCAcCgAwIBBaEDAgEOogcD...
          </wsse:BinarySecurityToken>
          ...
      </wsse:Security>
    </S11:Header>
    <S11:Body>
        ...
    </S11:Body>
</S11:Envelope>
```

The Kerberos token is referenced by the `<wsse:SecurityTokenReference>` element. The `<wsu:Id>` element, which is specified within the `<wsse:BinarySecurityToken>` element and is shown within the following example in boldface type, directly references the token in the `<wsse:SecurityTokenReference>` element.

The `@wsse:TokenType` attribute value within the `<wsse:SecurityTokenReference>` element matches the `ValueType` attribute value of the `<wsse:BinarySecurityToken>` element. The `Reference/@ValueType` attribute is not required. However, if the attribute is specified, its value must be equivalent to the `@wsse11:TokenType` attribute.

The following example shows the message format, the correlation between the`<wsu:Id>` and `<wsse:SecurityTokenReference>` elements, and the relationship between the `@wsse:TokenType` and `ValueType` attribute values.

```
<S11:Envelope xmlns:S11="..." xmlns:wsu="...">
   <S11:Header>
      <wsse:Security xmlns:wsse="...">
          <wsse:BinarySecurityToken
               EncodingType="http://docs.oasis-open.org/wss/2004/01/
                    oasis-200401-wss-soap-message-security-1.0#Base64Binary"
               ValueType="http://docs.oasis-open.org/wss/
                    oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ"
               wsu:Id="MyToken">boIBxDCCAcCgAwIBBaEDAgEOogcD...
          </wsse:BinarySecurityToken>
          </wsse:Security>
   </S11:Header>
</S11:Envelope>
      <wsse:Security>
       </wsse:Security>
          <wsse:SecurityTokenReference
               TokenType="http://docs.oasis-open.org/wss/
                    oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ">
               <wsse:Reference URI="#MyToken"
                    ValueType="http://docs.oasis-open.org/wss/
                        oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ">
              </wsse:Reference>
          </wsse:SecurityTokenReference>
         ...
      <wsse:Security>
      </wsse:Security>
    <S11:Header>
    </S11:Header>
    <S11:Body>
        ...
    </S11:Body>
 <S11:Envelope>
 </S11:Envelope>
```

The `<wsse:KeyIdentifier>` element is used to specify an identifier for the Kerberos token. The value of the identifier is a SHA1 hash value of the encoded Kerberos token in the previous message. The element

must have a ValueType attribute with a `#Kerberosv5APREQSHA1` value. The KeyIdentifier reference mechanism is used on subsequent message exchanges after the initial Kerberos token is accepted. The following example shows the key identifier information in boldface type:

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="...">
    <S11:Header>
      <wsse:Security>
         ...
         <wsse:SecurityTokenReference
               wsse11:TokenType=http://docs.oasis-open.org/wss/
                   oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ>
               <wsse:KeyIdentifier
                   ValueType="http://docs.oasis-open.org/wss/
                       oasis-wss-kerberos-token-profile-1.1#Kerberosv5APREQSHA1">
                       GbsDt+WmD9XlnUUWbY/nhBveW8I=
               </wsse:KeyIdentifier>
          </wsse:SecurityTokenReference>
          ...
      </wsse:Security>
   </S11:Header>
   <S11:Body>
       ...
    </S11:Body>
</S11:Envelope>
```

Multiple references to the Kerberos token

The client is not required to send a Kerberos token in every request after the Kerberos identity is validated and accepted by the service. The OASIS Web Services Security Kerberos Token Profile Version 1.1 specification suggests that you use a SHA1 encoded key with the `<wsse:KeyIdentifier>` element within the `<wsse:SecurityTokenReference>` element for every subsequent message after the initial AP_REQ packet is accepted. However, the runtime environment for Web Services Security must map the key identifier to a cached Kerberos token for further processing. IBM WebSphere Application Server 7.0 and later supports this SHA1 caching as described in the profile, by default. However, the application server also provides the ability to generate new AP_REQ tokens for each request with the existing service Kerberos ticket. When you interoperate with Microsoft .NET, do not use pSHA1 caching; generate an AP_REQ packet for each request.

Kerberos configuration models for web services:

The IBM WebSphere Application Server configuration model leverages existing frameworks.

The configuration model features include:

- Deployment descriptors and bindings configuration to enable the Kerberos token profile for Java API for XML-based RPC (JAX-RPC) applications
- Policy sets and bindings configuration to enable the Kerberos token profile for Java Architecture for XML Web Services (JAX-WS) applications
- Web Services Security APIs for JAX-WS applications
- Administrative command scripts
- Interoperability with Microsoft Web Services Enhancements (WSE) Version 3.5

Following are some examples of possible configurations when using the Kerberos token:

- A JAX-WS client on Windows operating systems
- A JAX-RPC client on Windows operating systems
- A Windows JAX-RPC client on z/OS operating systems
- Web Services Security APIs on Windows operating systems
- A Microsoft .NET WSE 3.5 client on Windows operating systems
- A Microsoft .NET WSE 3.5 client on z/OS operating systems

JAX-WS configuration model

For JAX-WS applications, the WebSphere Application Server client configuration model uses the policy set and leverages a custom policy set for the Kerberos token. You can specify the Kerberos token type and message signing and the encryption by using the custom policy set. The Web Services Security (WS-Security) policy is the security policy that is used to secure the application messages.

Using the administrative console, you can specify the Kerberos token type, message signing, and message encryption by using an existing custom policy set. Kerberos token generation and consumption includes the Kerberos token generation for unmanaged JAX-WS clients.

The JAX-WS programming model also provides capabilities to enable the Kerberos token profile and identity assertion by configuring the Kerberos token using policy sets, Web Services Security APIs, and administrative command scripts.

For JAX-WS applications, you can use administrative commands to configure the policy set as an alternative to using the administrative console.

JAX-RPC configuration model

JAX-RPC applications are configured using a deployment model. The deployment descriptor specifies the custom token to use for the Kerberos token. A JAX-RPC client can generate the specified Kerberos token. A JAX-RPC web service can successfully authenticate the Kerberos token by using a custom or the default Kerberos identity mapping login module.

API configuration model

A set of APIs is provided by WebSphere Application Server. To successfully use these APIs, application developers must have knowledge about the OASIS Web Services Security Version 1.0 and 1.1 specifications. When you use these APIs, the application server assumes that a policy set is not attached to the client resources; however, a warning is still issued when the application server detects any policy set information.

For JAX-WS client applications, the APIs include and enforce Web Services Security policy for the Kerberos token, which is based on the OASIS token profile. To enable the Kerberos token profile with the policy set, you must first configure the WS-Security policy and the binding files with the custom token.

For JAX-RPC applications, APIs for Web Services Security are not provided. You must use the deployment descriptor to specify the custom token to use the Kerberos token. You can use the custom token panels within an assembly tool, such as Rational Application Developer, to configure the deployment information.

Kerberos clustering for web services:

Clusters are groups of servers that are managed together and participate in workload management.

In a clustered environment, the Kerberos token needs to be distributed and recoverable. The Web Services Security configuration saves and distributes Kerberos tokens among the cluster members. The Kerberos tokens that are created or validated in one server are available to the other cluster members. The distributed cache or database repository need to be configured as the caching mechanism.

Web Services Security Kerberos token for authentication in a single or cross Kerberos realm environment:

To secure web services messages, you can use a Kerberos token as either an authentication token or a message protection token. For Kerberos authentication, both the single Kerberos realm environment, and the cross or trusted Kerberos realm environment are supported.

Single realm environment

In a single Kerberos realm environment, both the client application and the service provider use the same Kerberos realm. The client application obtains a Kerberos token based on the Kerberos realm used by the service provider. To configure the token, the client application defines the Kerberos service principal name (SPN) for the service provider in the client policy token generator bindings. The format of the SPN is shown later in this section, where `Kerberos_Realm_Name` is optional.

```
ServiceName/HostName@Kerberos_Realm_Name
```

For cell-level configuration in WebSphere Application Server, all service providers use the same Kerberos realm.

If the service provider uses the Kerberos identity from the client for downstream web services requests, a delegated Kerberos ticket must exist in the Kerberos token that is specified in the Kerberos configuration file. The system JAAS login module for Kerberos is added to the provided Web Services Security caller. For more information on using the Kerberos token for caller credentials, read about updating the system Java Authentication and Authorization Service (JAAS) login with the Kerberos login module, and creating a Kerberos configuration file.

Cross realm environment or trusted realm environment

The following configuration procedures must be completed for the trusted realm environment:

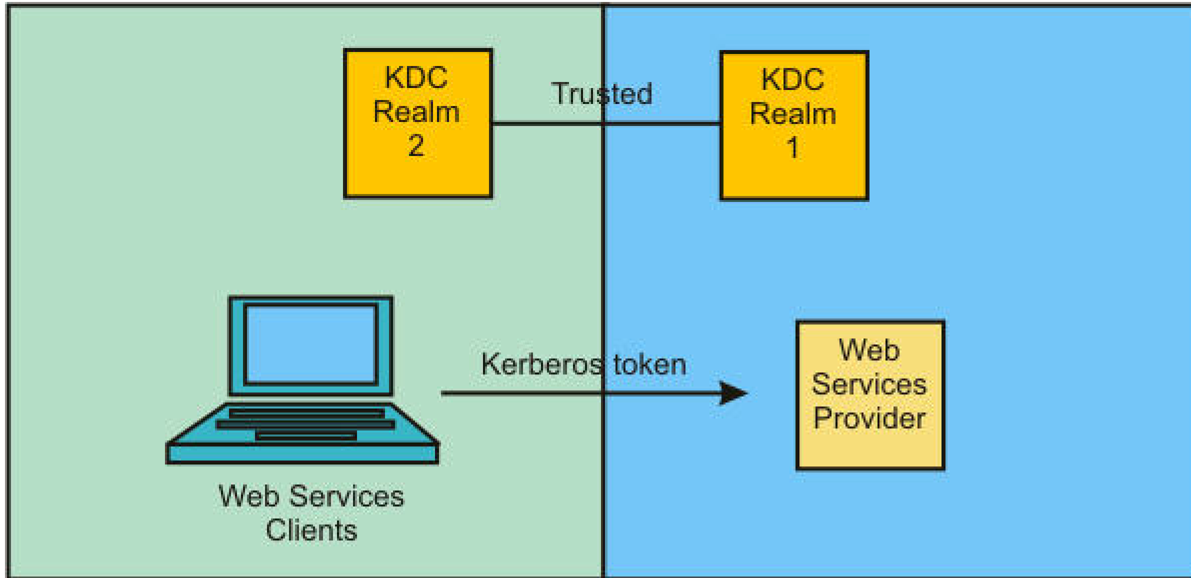
- The Kerberos trusted realm setup must be completed for all the configured Kerberos KDCs. See your Kerberos Administrator and User's Guide for more information about how to set up a Kerberos trusted realm.
- The Kerberos configuration file (krb5.ini on Windows, and krb5.conf for Unix and z/OS platforms) must list the trusted realms. See your Kerberos Administrator and User's Guide for more information.
- The client application token generator bindings must be configured with the Kerberos SPN information from the service provider. For more information, see configuring the bindings for message protection for Kerberos.

In a cross or trusted Kerberos realm environment, the client application and the service provider use different Kerberos realms that have established trust with each other. The client application obtains a Kerberos token based on the Kerberos realm used by the service provider. To configure the token, the client application defines the Kerberos SPN for the service provider in the client policy token generator bindings. The format of the SPN is shown later in this section, where `Kerberos_Realm_Name` is required.

```
ServiceName/HostName@Kerberos_Realm_Name
```

The client application must specify the Kerberos realm name for the client in the callback handler portion of the client policy token generator bindings. At the cell level, all service providers use the same Kerberos realm. However, client applications can still define their own Kerberos realm. Only peer-to-peer and transitive trust cross-realm authentication are supported.

The following figure illustrates the relationship between trusted realms as defined in the Kerberos Key Distribution Center (KDC):



If the service provider uses the Kerberos identity from the client for downstream web services requests, a delegated Kerberos ticket must exist in the Kerberos token that is configured in the Kerberos configuration file. The system JAAS login module for Kerberos is added to the provided Web Services Security caller. For more information on using the Kerberos token for caller credentials, read about updating the system JAAS login with the Kerberos login module, and creating a Kerberos configuration file.

SAML token:

The Security Assertion Markup Language (SAML) is an XML-based OASIS standard for exchanging user identity and security attributes information.

Using the product SAML function, you can apply policy sets to JAX-WS applications to use SAML assertions in web services messages and in web services usage scenarios. Use SAML assertions to represent user identity and user security attributes, and optionally, to sign and to encrypt SOAP message elements. WebSphere Application Server supports SAML assertions using the bearer subject confirmation method and the holder-of-key subject confirmation method as defined in the OASIS Web Services Security SAML Token Profile Version 1.1 specification. Policy sets and general bindings that support SAML are included with the product SAML function. To use SAML assertions, you must modify the provided sample general binding.

The SAML function also provides a set of application programming interfaces (APIs) that can be used to request SAML tokens from a Security Token Service (STS) using the WS-Trust protocol. APIs are also provided to locally generate and validate SAML tokens. For more information, read about application programming interfaces (APIs) for SAML.

Time stamp:

A *time stamp* is the value of an object that indicates the system time at some critical point in the history of the object.

A time stamp is included in a message to reduce the vulnerability of an application to replay attacks. In web services, a replay attack occurs when an HTTP request is intercepted and the content is resent to the provider in its original form.

Note: When you include a time stamp in a message, you must protect its integrity using transport security, such as secure sockets layer (SSL) or message-level security, such as XML digital signature. If you

do not protect the integrity of the time stamp, it is possible to capture the message and retransmit the content with a different time stamp, message expiration date, or both.

For both the JAX-RPC and JAX-WS WS-Security run times, 5 minutes is the default message expiration time that is used for the receiver if a value is not specified in the message. If a different expiration is required for a specific client or you are unsure of the target service default value, configure a message expiration time value for the outbound time stamp.

Note:

- When the Web Services Security JAX-RPC and JAX-WS run times generate or consume a message, they do not enforce that the integrity of the time stamp is protected.
- The Web Services Security JAX-RPC and JAX-WS run times do not have a default outbound message expiration value. If you want to include a message expiration value in a message, you must configure it. Although the JAX-WS run time does not have a default outbound message expiration value, you can configure an outbound message expiration value in the default general bindings. This value is acquired by all applications at the level for which the default bindings apply. For example, the value might be acquired at the cell or application level.
- For the JAX-RPC run time, the time stamp expiration value is specified in the web services deployment descriptor extension. You cannot modify the web services deployment descriptor extension from the administrative console; you can only view it. To modify the deployment descriptor extension, you must use an assembly tool and add or change the time stamp expiration value for a JAX-RPC application.
- If WS-Security constraints exist to consume a timestamp, the client must send a timestamp.

The JAX-WS WS-Security runtime complies with the OASIS WS-SecurityPolicy 1.2 specification Timestamp Required requirement. If you want to configure an application to not require an inbound time stamp when an outbound time stamp is configured you can add the com.ibm.wsspi.wssecurity.consumer.timestampRequired custom property as either an inbound or an inbound/outbound web services security custom property.

The JAX-WS runtime always puts the timestamp first, but the JAX-RPC runtime does not. If you are using the JAX-RPC WS-Security 1.0 runtime, and want to emit the Timestamp first in the Security header, you must:

- Set the property **com.ibm.wsspi.wssecurity.timestamp.keyword** to `SecurityFirst`.
- Set the property **com.ibm.wsspi.wssecurity.timestamp.dialect** to `http://www.ibm.com/websphere/webservices/wssecurity/dialect-was`. The default value for **com.ibm.wsspi.wssecurity.timestamp.dialect** is dialect-was, but for the desired function to work, the property must be set explicitly.

These properties are specified as Web Services Security property configuration settings.

Security considerations for web services:

When you configure Web Services Security, you should make every effort to verify that the result is not vulnerable to a wide range of attack mechanisms. There are possible security concerns that arise when you are securing web services.

In WebSphere Application Server, when you enable integrity, confidentiality, and the associated tokens within a SOAP message, security is not guaranteed. This list of security concerns is not complete. You must conduct your own security analysis for your environment.

- Ensuring the message freshness

  Message freshness involves protecting resources from a replay attack in which a message is captured and resent. Digital signatures, by themselves, cannot prevent a replay attack because a signed message can be captured and resent. It is recommended that you allow message recipients to detect

message replay attacks when messages are exchanged through an open network. You can use the following elements, which are described in the Web Services Security specifications, for this purpose:

Timestamp
: You can use the timestamp element to keep track of messages and to detect replays of previous messages. The WS-Security 2004 specification recommends that you cache time stamps for a given period of time. As a guideline, you can use five minutes as a minimum period of time to detect replays. Messages that contain an expired timestamp are rejected.

Nonce
: A nonce is a child element of the <UsernameToken> element in the UsernameToken profile. Because each nonce element has a unique value, recipients can detect replay attacks with relative ease.

Important: Both the time stamp and nonce element must be signed. Otherwise, these elements can be altered easily and, therefore, cannot prevent replay attacks.

- Using XML digital signature and XML encryption properly to avoid a potential security hole

  The Web Services Security 2004 specification defines how to use XML digital signature and XML encryption in SOAP headers. Therefore, users must understand XML digital signature and XML encryption in the context of other security mechanisms and their possible threats to an entity. For XML digital signature, you must be aware of all of the security implications resulting from the use of digital signatures in general and XML digital signature in particular. When you build trust into an application based on a digital signature, you must incorporate other technologies such as certification trust validation based upon the Public Key Infrastructure (PKI). For XML encryption, the combination of digital signing and encryption over a common data item might introduce some cryptographic vulnerabilities. For example, when you encrypt digitally signed data, you might leave the digital signature in plain text and leave your message vulnerable to plain text guessing attacks. As a general practice, when data is encrypted, encrypt any digest or signature over the data. For more information, see http://www.w3.org/TR/xmlenc-core/#sec-Sign-with-Encrypt.
- Protecting the integrity of security tokens

  The possibility of a token substitution attack exists. In this scenario, a digital signature is verified with a key that is often derived from a security token and is included in a message. If the token is substituted, a recipient might accept the message based on the substituted key, which might not be what you expect. One possible solution to this problem is to sign the security token (or the unique identifying data from which the signing key is derived) together with the signed data. In some situations, the token that is issued by a trusted authority is signed. In this case, there might not be an integrity issue. However, because application semantics and the environment might change over time, the best practice is to prevent this attack. You must assess the risk assessment based upon the deployed environment.
- Verifying the certificate to leverage the certificate path verification and the certificate revocation list

  It is recommended that you verify that the authenticity or validity of the token identity that is used for digital signature is properly trusted. Especially for an X.509 token, this issue involves verifying the certificate path and using a certificate revocation list (CRL). In the Web Services Security implementation in WebSphere Application Server Version 6 and later, the certificate is verified by the <TokenConsumer> element. WebSphere Application Server provides a default implementation for the X.509 certificate that uses the Java CertPath library to verify and validate the certificate. In the implementation, there is no explicit concept of a CRL. Rather, proper root certificates and intermediate certificates are prepared in files only. For a sophisticated solution, you might develop your own TokenConsumer implementation that performs certificate and CRL verification using the online CRL database or the Online Certificate Status Protocol (OCSP).
- Protecting the username token with a password

  It is recommended that you do not send a password in a username token to a downstream server without protection. You can use transport-level security such as SSL (for example, HTTPS) or use XML encryption within Web Services Security to protect the password. The preferred method of protection

depends upon your environment. However, you might be able to send a password to a downstream server as plain text in some special environments where you are positive that you are not vulnerable to an attack.

Securing web services involves more work than just enabling XML digital signature and XML encryption. To properly secure a Web service, you must have knowledge about the PKI. The amount of security that you need depends upon the deployed environment and the usage patterns. However, there are some basic rules and best practices for securing web services. It is recommended that you read some books on PKI and also read information on the Web Services Interoperability Organization (WS-I) Basic Security Profile (BSP).

Nonce, a randomly generated token:

Nonce is a randomly-generated, cryptographic token that is used to prevent replay attacks. Although nonce can be inserted anywhere in the SOAP message, it is typically inserted in the <UsernameToken> element.

Without nonce, when a UsernameToken is passed from one machine to another machine using a nonsecure transport, such as HTTP, the token might be intercepted and used in a replay attack. The same password might be reused when the user name token is transmitted between the client and the server, which leaves it vulnerable to attack. The user name token can be stolen even if you use XML digital signature and XML encryption. However, nonce alone, used in a non-secure transport, cannot adequately address the replay problem. Nonce is most useful when the SOAP message is transmitted via a communication channel that is secured, either at the transport level, or at the message level.

To help eliminate these replay attacks, the <wsse:Nonce> and <wsu:Created> elements are generated within the <wsse:UsernameToken> element and used to validate the message. The server checks the freshness of the message by verifying that the difference between the nonce creation time, which is specified by the <wsu:Created> element, and the current time falls within a specified time period. Also, the server checks a cache of used nonces to verify that the user name token in the received SOAP message has not been processed within the specified time period. These two features are used to lessen the chance that a user name token is used for a replay attack.

To add a nonce for the UsernameToken, you can specify it in the token generator for the user name token. When the token generator for the UsernameToken is specified, you can select the **Add nonce** option if you want to include nonce in the user name token.

Basic Security Profile compliance tips:

The Web Services Interoperability Organization (WS-I) Basic Security Profile (BSP) 1.0 promotes interoperability by providing clarifications and amplifications to a set of nonproprietary web services specifications. WebSphere Application Server Web Services Security provides configuration options to ensure that the BSP recommendations and security considerations can be enabled to ensure interoperability. The degree to which you follow these recommendations is then a measure of how well the application you are configuring complies with the Basic Security Profile (BSP).

Support for applications to comply to the Basic Security Profile (BSP) is new in WebSphere Application Server Version 8.5. For more information on the Basic Security Profile, see Web Services Interoperability Organization (WS-I) Basic Security Profile (BSP), Basic Security Profile Version 1.0.

You can use either a predefined list of keywords or XPath expressions to comply to the BSP. Both the keywords and the XPath expressions are specified in the deployment descriptor configuration file and are configured using an assembly tool.

Basic Security Profile recommendations

Follow these recommendations to ensure that your configured applications are Basic Security Profile (BSP) compliant.

- Do not use the original XPath transform, http://www.w3.org/TR/1999/REC-xpath-19991116

  When you refer to an element in a SECURE_ENVELOPE that does not carry an ID attribute type from a ds:Reference in a SIGNATURE element, you must use the XPath Filter 2.0 transform, http://www.w3.org/2002/06/xmldsig-filter2 to refer to that element.

  Any ds:Transform/@Algorithm attribute in a SIGNATURE element must have one of these values:
  - http://www.w3.org/2001/10/xml-exc-c14n#
  - http://www.w3.org/2002/06/xmldsig-filter2
  - `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform`
  - http://www.w3.org/2000/09/xmldsig#enveloped-signature
  - `http://docs.oasis-open.org/wss/2004/XX/oasis-2004XX-wss-swa-profile-1.0#Attachment-Content-Only-Transform`
  - `http://docs.oasis-open.org/wss/2004/XX/oasis-2004XX-wss-swa-profile-1.0#Attachment-Complete-Transform`
- Do not use the http://www.w3.org/2000/09/xmldsig#dsa-sha1 signature algorithm.

  Any ds:SignatureMethod/@Algorithm element in a SIGNATURE that is based on a symmetric key must have one of the following values:
  - http://www.w3.org/2000/09/xmldsig#rsa-sha1
  - http://www.w3.org/2000/09/xmldsig#hmac-sha1
- Do not specify the digestvalue keyword for the message part to encrypt. Instead, use the signature keyword.

  If the value of a ds:DigestValue element in a SIGNATURE element requires encryption, the entire parent ds:Signature element must be encrypted. A SIGNATURE must not have any xenc:EncryptedData elements among its descendants.
- Do not use the KEYNAME key information type

  KEYNAME references can be ambiguous and compliance with the BSP disallows the use of KEYNAME.

  A SECURITY_TOKEN_REFERENCE must not use a key name to reference a SECURITY_TOKEN. The child element of a ds:KeyInfo element in an ENCRYPTED_KEY must be either a SECURITY_TOKEN_REFERENCE or a ds:MgmtData element. Using a KEYNAME key information type for an encryption key results in a KeyName child element of a ds:KeyInfo element and is disallowed for BSP compliance.
- Do not use the http://www.w3.org/2001/04/xmlenc#aes192-cbc bit data encryption algorithm.

  Any xenc:EncryptionMethod/@Algorithm attribute in an ENCRYPTED_DATA element must have one of these values:
  - http://www.w3.org/2001/04/xmlenc#tripledes-cbc
  - http://www.w3.org/2001/04/xmlenc#aes128-cbc
  - http://www.w3.org/2001/04/xmlenc#aes256-cbc
- Do not use the advanced encryption standard (AES) key wrap (aes192): http://www.w3.org/2001/04/xmlenc#kw-aes192 key encryption algorithm.

  When used for key wrap, any xenc:EncryptionMethod/@Algorithm attribute in an ENCRYPTED_KEY element must have one of these values:
  - http://www.w3.org/2001/04/xmlenc#kw-tripledes
  - http://www.w3.org/2001/04/xmlenc#kw-aes128
  - http://www.w3.org/2001/04/xmlenc#kw-aes256

Configuration Options for BSP Compliance

You achieve BSP compliance when certain configuration choices are made. The assembly tool assists you in using appropriate choices when configuring the application by issuing warning messages. The following configuration descriptions comprise these warnings:

- When configuring the ds:Transforms element in a signature, the list of transforms must include as its last child element http://www.w3.org/2001/10/xml-exc-c14n# or `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform`
- Add a wsse:Nonce or wsse:Created element to a Username token to prevent replay. After the element is added, sign the Username token to prevent undetected alteration of these fields; otherwise, replay can occur.

Distributed nonce cache:

In previous releases of WebSphere Application Server, the nonce was cached locally. WebSphere Application Server Versions 6 and later use distributed nonce caching. The distributed nonce cache makes it possible to replicate nonce data among servers in a WebSphere Application Server cluster.

If nonce elements are in a SOAP header, all nonce values are cached by the server in the cluster. If the distributed nonce cache is enabled, the cached nonce values are copied to other servers in the same cluster. Then, if the message with the same nonce value is sent to (one of) other servers, the message is rejected. A received nonce cache value is cached and replicated in a push manner among other servers in the cluster with the same replication domain. The replication is an out-of-process call and, in some cases, is a remote call. Therefore, there is latency when the content of the cache in the cluster is updated.

For example, you might have application server A and application server B in cluster C.

- A SOAP client sends a message with nonce `abc` to application server A.
- The server caches the value and pushes it to the other application server B.
- If the client sends the message with nonce `abc` to application server B after a certain time frame, the message is rejected and if the application server B receives the nonce with the same value within a specified period of time, a SoapSecurityException is thrown by application server B.

  For more information, see the information that explains nonce cache timeout, nonce maximum age, and nonce clock skew in "Token generator configuration settings" on page 921.
- If the client sends the message with another nonce value of `xyz`, the message is accepted, the value is cached by application server B and is copied into the other application servers within the same cluster.

Important: The distributed nonce caching feature uses the WebSphere Application Server data replication service (DRS). The data in the local cache is pushed to the cache in other servers in the same replication domain. The replication is an out-of-process call and, in some cases, is a remote call. Therefore, there is a possible delay in replication while the content of the cache in each application server within the cluster is updated. The delay might be due to network traffic, network workload, machine workload, and so on.

Web Services Security token propagation:

Web Services Security has the ability to send security tokens in the security header of a SOAP message. These security tokens can be used to sign, verify, encrypt or decrypt message parts. Security tokens can also be sent as stand-alone security tokens and set as the caller on the request consumer. *Web Services Security token propagation* is used to send these stand-alone security tokens in a wsse:BinarySecurityToken element within the security header of the SOAP message.

Web Services Security has the following built-in token types:

- Username token
- X.509 token

- Lightweight Third-Party Authentication (LTPA) token

You can configure Web Services Security to use custom security tokens. Web Services Security uses the same propagation token format as the Security attribute propagation feature. Web Services Security can propagate all of the built-in security token types and can propagate custom token types as long as they are serializable by the security attribute propagation feature.

When you configure a propagation token in a token generator or token consumer, use the following values for the token type Uniform Resource Identifier (URI) and local name:
- **Token type URI:** http://www.ibm.com/websphere/appserver/tokentype
- **Token type local name:** LTPA_PROPAGATION

When a propagation token is generated, Web Services Security gathers all of the serializable security tokens in the *RunAs* subject for the current thread and serialize the security tokens within a wsse:BinarySecurityToken token. To have a RunAs subject and the credentials that are necessary on the current thread, a JAAS login must occur on the current thread before a propagation token can be created.

Under ordinary circumstances, for a service provider, the Java Authentication and Authorization Service (JAAS) login is achieved by including a defined caller part for the inbound token in the WS-Security configuration. For a web services client, the JAAS login is achieved by configuring HTTP basic authentication.

There are two common uses for a propagation token:
- A client from within a secured service propagates the serializable security tokens and credentials from the current RunAs subject to a downstream server.
- A server-based client that is secured in the web container with HTTP basic authentication can use a propagation token.

  For a server-based client, the overhead for propagation tokens is not necessary as only the identity is required and not the full set of credentials. However, if the client application makes modifications to the subject after it is invoked by the web container, it might be appropriate to use a propagation token. If only an identity token is required, an ordinary LTPA token might be appropriate. You can generate this LTPA token from the RunAs subject that is created by the JAAS login.

Important: For the receiver of the LTPA propagation token to make proper use of the credentials that were sent to it in the propagation token, you must configure and define a caller part for the token in the WS-Security configuration on the receiver side.

Overview of standards and programming models for web services message-level security

Web Services Security standards and profiles describe how to provide security and protection for SOAP messages that are exchanged in a web services environment.

To secure web services, you must consider a broad set of security requirements, including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, delegation, and auditing across a spectrum of application and business topologies. One of the key requirements for the security model in today's business environment is the ability to inter-operate between formerly incompatible security technologies in heterogeneous environments. The complete Web Services Security protocol stack and technology roadmap is described in Security in a Web Services World: A Proposed Architecture and Roadmap.

Web Services Security standards

The Organization for the Advancement of Structured Information Standards (OASIS) Web Services Security (WS-Security) specification defines the core facilities for protecting the integrity and confidentiality of a message and provides mechanisms for associating security-related claims with the message. Web

Services Security is a message-level standard based on securing SOAP messages through XML digital signature, confidentiality through XML encryption, and credential propagation through security tokens. WebSphere Application Server supports Version 1.1 of the Web Services Security specification, including features such as encrypted header, thumbprint and signature configuration, username token profile and X.509 token profile. In addition, limited security scenario support is provided for the Kerberos Version 1.1 token profile, WS-SecureConversation Version 1.3, WS-Trust Version 1.3, and WS-SecurityPolicy Version 1.2.

The Web Services Security SOAP Message Security 1.1 specification outlines a standard set of SOAP 1.1 extensions that you can use to build secure web services. These standards provide integrity and confidentiality protection, which are generally implemented with digital signature and encryption technologies. In addition, Web Services Security provides a general purpose mechanism for associating security tokens with messages. A typical example of the security token is a username token, in which a user name and password are included as text. Web Services Security defines how to encode binary security tokens using methods such as X.509 certificates. However, the required security tokens are not defined in the SOAP Message Security 1.1 specification. Instead, the tokens are defined in separate profiles such as the Username token profile, the X.509 token profile, and so on.

It is important to note that while Web Services Security can be used to provide message level integrity and confidentiality protection for normal SOAP message requests from a client to a service, and normal SOAP message responses from a service to a client, Web Services Security cannot be used to protect SOAP fault messages.

Compatibility between WS-Security Draft 13 and WS-Security standard Versions 1.0 and 1.1

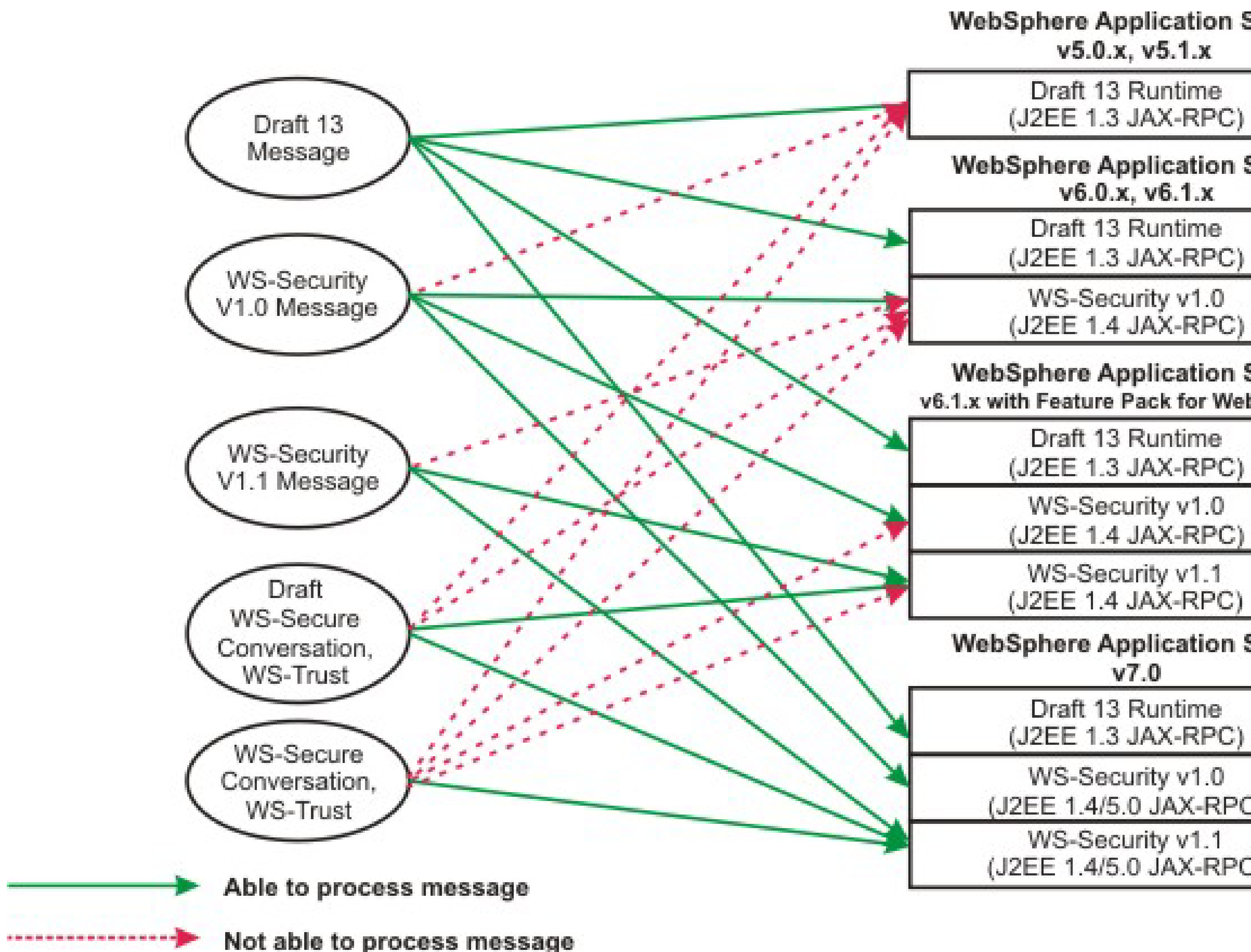
The WS-Security standard has evolved over the years, from a draft to an OASIS standard. WebSphere Application Server Version 5.02 introduced support for the WS-Security Draft 13, and support for WS-Security 1.0 was introduced beginning with WebSphere Application Server Version 6.0. WS-Security Version 1.1 is supported by WebSphere Application Server Version 6.1 Feature Pack for Web Services, using the JAX-WS runtime only. The topic Web Services Security specification - a chronology provides more details about the evolution of this support.

It is important to note that a WS-Security Draft 13 client is not compatible with providers that use WS-Security Version 1.0 or Version 1.1. You must use Draft 13 client to communicate with a Draft 13 web services provider. You cannot use a Draft 13 client to communicate with a WS-Security Version 1.0 provider, or a Version 1.1 provider. This issue arises because the SOAP message format for the WS-Security header and namespace is different between a WS-Security Draft 13–enabled application and a WS-Security Version 1.0 or Version 1.1–enabled application.

The version of the WS-Security standard that is used also has implications for the required version of the Java Platform, Enterprise Edition (Java EE) application:

- Java EE Version 1.3 is used only with WS-Security Draft 13.
- Java EE Version 1.4 and later is used with WS-Security Version 1.0 (JAX-RPC and JAX-WS), and also WS-Security Version 1.1 (JAX-WS).

The following diagram illustrates these compatibility considerations:



To secure web services with WebSphere Application Server, you must specify several different configurations. Although there is not a specific sequence in which you must specify these different configurations, some configurations reference other configurations. See “Web Services Security configuration considerations” on page 222.

Because of the relationship between the different Web Services Security configurations, it is recommended that you specify the configurations on each level of the configuration in the order described in the following sections. You can choose to configure Web Services Security for the application level, the server level or the cell level as it depends upon your environment and security needs.

Web Services Security programming models

Take advantage of the easy-to-implement Java™ API for XML-Based Web Services (JAX-WS) programming model to develop new web services applications and clients. JAX-WS is the next generation web services programming model. Using JAX-WS, development of web services and clients is simplified, with greater platform independence for Java applications through the use of dynamic proxies and Java annotations. JAX-WS simplifies application development through support of a standard, annotation-based model to develop web service applications and clients. JAX-WS applications can be secured with Web Services Security in one of two ways. The application can be secured using policy sets, or through the use

of the Web Services Security API (WSS API). To secure web services using the Java API for XML-Based Web Services (JAX-WS) programming model, begin with the topic "Securing JAX-WS web services using message-level security" on page 191.

The Java™ API for XML-based RPC (JAX-RPC) specification enables you to develop SOAP-based interoperable and portable web services and web service clients. JAX-RPC 1.1 provides core APIs for developing and deploying web services on a Java platform and is a part of the Web Services for Java Platform, Enterprise Edition (Java EE) platform. IBM® WebSphere® Application Server supports both the JAX-WS programming model and the JAX-RPC programming model. JAX-WS is the next generation web services programming model, extending the foundation provided by the JAX-RPC programming model. To secure web services using the Java API for XML-based RPC (JAX-RPC) programming model, begin with the topic "Securing JAX-RPC web services using message-level security" on page 192.

SAML concepts

SAML is an XML-based, OASIS standard for exchanging user identity and security attributes information. In a typical SAML usage scenario, you authenticate to a security domain and request an identity provider to issue SAML assertions.

The SAML assertions are presented to a security provider when you request access to business resources. In many cases, the services provider and identity provider are in different security domains, meaning that you must authenticate to an identity provider user directory, which is not the same as the user directory of the service provider. WebSphere Application Server multiple security domain support allows a service provider to assert user identity and security attributes to a local security domain, based on trust relationship without requiring identity mapping. You can use the SAML function to quickly build a Single Sign-On (SSO) solution across enterprises and across the Internet with industry standard SAML security tokens.

See the following topics to learn about the product SAML function.

SAML assertions defined in the SAML Token Profile standard:

The Web Services Security SAML Token Profile OASIS standard specifies how to use Security Assertion Markup Language (SAML) assertions with the Web Services Security SOAP Message Security specification.

This product supports two versions of the OASIS SAML standard: Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1 and Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0.

The standard describes the use of SAML assertions as security tokens in the <wsse:Security> header, as defined by the WSS: SOAP Message Security specification. An XML signature can be used to bind the subjects and statements in the SAML assertion to the SOAP message.

Subject confirmation methods define the mechanism by which an entity provides evidence (proof) of the relationship between the subject and the claims of the SAML assertions. The WSS: SAML Token Profile describes the usage of three subject confirmation methods: bearer, holder-of-key, and sender-vouches. This product supports all three confirmation methods. When using the bearer subject confirmation method, proof of the relationship between the subject and claims is implicit. No specific steps are taken to establish the relationship.

Since there is no key material associated with a bearer token, protection of the SOAP message, if required, must be performed using a transport level mechanism or another security token, such as an X.509 or Kerberos token, for message level protection. When using the holder-of-key subject confirmation method, proof of the relationship between the subject and claims is established by signing part of the

SOAP message with the key specified in the SAML assertion. Since there is key material associated with a holder-of-key token, this token can be used to provide message level protection (signing and encryption) of the SOAP message.

The sender-vouches confirmation method is used when a server needs to propagate the client identity with SOAP messages on behalf of the client. This method is similar to identity assertion, but it has the added flexibility of using SAML assertions to propagate not only the client identity, but also propagate client attributes. The attesting entity must protect the vouched for SAML assertions and SOAP message content so that the receiver can verify that it has not been altered by another party. Two sender-vouches confirmation method usage scenarios are supported that ensure message protection either at the transport level or the message level. A receiver verifies that one of the following scenarios occurs:

- A sender sets up a secure sockets layer (SSL) session with a receiver using client certificate authentication.
- A sender digitally signs SAML assertions with the containing SOAP message using security token reference transformation algorithm. A sender can use either SSL or SOAP message encryption to protect confidentiality.

In either case, the SAML assertions are either issued by an external Security Token Services (STS) or self- issued by the application server.

Bearer assertion

A SAML assertion is a bearer assertion if it includes the bearer <saml:ConfirmationMethod> element, as defined in the OASIS Web Services Security specifications. For more information, refer to the specification documents.

A SAML Version 1.1 bearer assertion must also contain the following SubjectConfirmation element:

```
<saml:SubjectConfirmation>
  <saml:ConfirmationMethod>
  urn:oasis:names:tc:SAML:1.0:cm:bearer
  </saml:ConfirmationMethod>
</saml:SubjectConfirmation>
```

A SAML Version 2.0 bearer assertion must also contain the following SubjectConfirmation element:

```
<saml2:SubjectConfirmation
      Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
</saml2:SubjectConfirmation>
```

Holder-of-key assertion

A SAML assertion is a holder-of-key assertion if it includes the `SubjectConfirmation` element containing a `saml:ConfirmationMethod` element with the value of `holder-of-key`, and a `ds:KeyInfo` element. For more information, refer to the specification documents.

The ds:KeyInfo information inside the `SubjectConfirmation` element identifies a public or secret key that is used to confirm the identity of the subject. The holder-of-key assertion also contains a `ds:Signature` element that protects the integrity of the confirmation `ds:KeyInfo` element as established by the assertion authority.

A SAML Version 1.1 holder-of-key assertion must contain the following `SubjectConfirmation` element:

```
<saml:SubjectConfirmation>
  <saml:ConfirmationMethod>
   urn:oasis:names:tc:SAML:1.0:cm: holder-of-key
   </saml:ConfirmationMethod>
  <ds:KeyInfo>
  <ds:KeyValue>. . .</ds:KeyValue>
  </ds:KeyInfo>
</saml:SubjectConfirmation>
```

A SAML Version 2.0 holder-of-key assertion must contain the following `SubjectConfirmation` element:

```
<saml2:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:holder-of-key">
    <saml2:SubjectConfirmationData>
   <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                     ......
   </ds:KeyInfo>
     </saml2:SubjectConfirmationData>
</saml2:SubjectConfirmation>
```

Sender-vouches assertion

A SAML assertion is a sender-vouches assertion if it includes the sender-vouches <saml:ConfirmationMethod> element, as defined in the OASIS Web Services Security specifications. A SAML Version 1.1 sender-vouches assertion must contain the following SubjectConfirmation element:

```
<saml:SubjectConfirmation>
  <saml:ConfirmationMethod>
  urn:oasis:names:tc:SAML:1.0:cm:sender-vouches
  </saml:ConfirmationMethod>
</saml:SubjectConfirmation>
```

A SAML Version 2.0 sender-vouches assertion must contain the following SubjectConfirmation element:

```
<saml2:SubjectConfirmation
          Method="urn:oasis:names:tc:SAML:2.0:cm:sender-vouches">
</saml2:SubjectConfirmation>
```

Symmetric key in the holder-of-key assertion

A SAML holder-of-key assertion is used as a protection token. This type of protection token can use a symmetric key as a proof key. The client uses the proof key to demonstrate to the relying party that the client actually owns the issued SAML token. When a Security Token Service (STS) issues a SAML token that uses a symmetric proof key, the token contains a key that is encrypted for the target service. The STS also sends the same proof key to the requester in a <RequestedProofToken> element as part of the RequestSecurityTokenResponse (RSTR). The web service client then presents the SAML token to the target service, also known as the relying party, and signs the application message with the received proof key.

The STS can be pre-configured to issue a symmetric proof key. Typically, the following two parameters are specified inside the RequestSecurityTokenTemplate in the RequestSecurityToken (RST) when the symmetric key is requested from the STS:

```
<t:KeyType> http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey
  </t:KeyType>
<t:KeySize>256</t:KeySize>
```

The following sample SubjectConfirmation element contains a SymmetricKey encrypted for the relying party.

```
<saml:SubjectConfirmation>
      <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:holder-of-key</saml:ConfirmationMethod>
          <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
             <enc:EncryptedKey xmlns:enc="http://www.w3.org/2001/04/xmlenc#">
             <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
          <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
             </enc:EncryptionMethod>
          <ds:KeyInfo>
            <ds:X509Data>
              <ds:X509Certificate>MIIB3 . . . vO3bdg</ds:X509Certificate>
            </ds:X509Data>
          </ds:KeyInfo>
     <enc:CipherData>
         <enc:CipherValue>P5Kb . . . rOTvII</enc:CipherValue>
            </enc:CipherData>
            </enc:EncryptedKey>
     </ds:KeyInfo>
</saml:SubjectConfirmation>
```

Public key in the holder-of-key assertion

When a SAML holder-of-key assertion is used as a protection token, the token can use a public key as a proof key. The client uses the proof key to demonstrate to the relying party that the client actually owns the issued SAML token. The advantage of a public proof key over a symmetric key is that the client does not share the secret key with the Security Token Service (STS) and relying party. The public proof key can be an X509 certificate, or a Rivest Shamir Adleman (RSA) public key.

The STS can be pre-configured to issue a public key proof key. Typically, the parameter `<t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey</t:KeyType>` is specified inside a RequestSecurityTokenTemplate as part of the RequestSecurityToken (RST) when the public key is requested from the STS. The optional UseKey element could also be used by the client to indicate the key, as follows:

```
<trust:UseKey>
    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <X509Data>
        <X509Certificate>MIIGCzCCBPOgAwIBAgIQcSgVwaoQv6dG. . .1GqB
        </X509Certificate>
      </X509Data>
    </KeyInfo>
</trust:UseKey>
```

The following example is a SubjectConfirmation that contains a PublicKey proof key.

```
<saml:SubjectConfirmation>
        <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:holder-of-key</saml:ConfirmationMethod>
           <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
               <ds:KeyValue>
                   <ds:RSAKeyValue>
                     <ds:Modulus>hYHQm. . . ZnH1S0=</ds:Modulus>
           <ds:Exponent>AQAB</ds:Exponent>
                   </ds:RSAKeyValue>
               </ds:KeyValue>
           </ds:KeyInfo>
</saml:SubjectConfirmation>
```

Default policy sets and sample bindings for SAML:

SAML-specific default policy sets and general bindings are provided when the SAML function is installed. These policy sets and sample general bindings are used to request SAML tokens from an external Security Token Service (STS), and to propagate SAML tokens to downstream web services.

SAML default policy sets

WebSphere Application Server with SAML provides eight default policy sets and several sample general bindings that support the OASIS Web Services Security SAML Token Profile 1.1 standard. You must import the SAML default policy sets before you can use them. You can attach these SAML policy sets and sample bindings to web services and begin using SAML capability without writing any additional code. However, there are a few configuration parameters that you need to set in the sample bindings documents before using them. These parameters include the external STS URL, and the SAML token issuer X.509 certificate. For more information, read about configuring client and provider bindings for the SAML bearer token and configuring client and provider bindings for the SAML holder-of-key (HoK) symmetric key token.

The SAML function installation includes the Username WSHTTPS default application policy set, which sends a trust request to an external STS. A Security Token Service (STS) is a special-purpose web service. You can use any of the default WebSphere Application Server policy sets to communicate with the STS. However, the Username WSHTTPS default policy set sends a Username token in the SOAP message, and protects the message using Secure Sockets Layer (SSL). You must configure a user name and password in the bindings document to use this policy set. For step-by-step instructions, read about configuring policy sets and bindings to communicate with STS.

Four SAML 1.1 default policy sets and four SAML 2.0 policy sets are provided as part of the SAML function installation. As described in the topic, Setting up the SAML configuration, you must add these policy sets to an existing profile, or create a new profile after installing WebSphere Application Server, before you can use them. The following SAML policy sets exist:

- SAML11 Bearer WSHTTPS default: sends a SAML token using the bearer confirmation method in SOAP messages, and protects SOAP messages using SSL
- SAML11 Bearer WSSecurity default: sends a SAML token using the bearer confirmation method in SOAP messages, and protects SOAP messages using X.509 signing and encryption
- SAML11 HoK Public WSSecurity default: sends a SAML token using the holder-of-key confirmation method with a client X.509 certificate in the SAML token, and protects SOAP messages using the client certificate in the SAML token and the X.509 certificate of the recipient
- SAML11 HoK Symmetric WSSecurity default: sends a SAML token using the holder-of-key confirmation method with a shared key that is encrypted by recipient public key, and protects the SOAP message using the shared key for signing and encryption
- SAML20 Bearer WSHTTPS default
- SAML20 Bearer WSSecurity default
- SAML20 HoK Public WSSecurity default
- SAML20 HoK Symmetric WSSecurity default

The only difference between the SAML 1.1 policy sets and SAML 2.0 policy sets is the SAML token namespace.

SAML sample bindings

Two pairs of sample bindings are provided to support the SAML bearer token and the SAML holder-of-key token that uses a symmetric key.

- Saml Bearer Client sample
- Saml Bearer Provider sample
- Saml HoK Symmetric Client sample
- Saml HoK Symmetric Provider sample

You can use the Saml Bearer Client sample and the Saml Bearer Provider sample with any of the SAML bearer token default policy sets. You can use the Saml HoK Symmetric Client sample and the Saml HoK Symmetric Provider sample with one of the SAML HoK Symmetric key default policy sets: either SAML11 HoK Symmetric WSSecurity default or SAML20 HoK Symmetric WSSecurity default.

Trust client bindings

WebSphere Application Server with SAML supports both application-specific bindings and general bindings for trust client policy sets. In addition, default bindings are supported if the application is running in a server environment. Default bindings are not supported in the thin client environment.

You can create application-specific bindings only at a policy set attachment point. These bindings are specific to and defined by the characteristics of the policy. Application-specific bindings can provide configuration for advanced policy requirements, such as multiple signatures; however, these bindings are only reusable within an application. Furthermore, application-specific bindings have limited reuse across policy sets. When you create an application-specific binding for a policy set, the binding begins in an unconfigured state. You must add each policy, such as WS-Security or HTTP transport, that you want to override the default binding, and fully configure the bindings for each policy that you have added. For more information about configuring trust client bindings for SAML, refer to configuring policy sets and bindings to communicate with STS.

General bindings can be configured for use across a range of policy sets and can be reused across applications and for trust service attachment points. Although general bindings are highly reusable, they do not provide configuration for advanced policy requirements, such as multiple signatures. There are two types of general bindings: general provider policy set bindings, and general client policy set bindings.

If you do not specify the **wstrustClientBindingScope** property for the trust client binding, the system searches first in the application for an application-specific binding with the binding name that you specified. If a binding is found, the binding is used for the trust client request. If no application-specific binding is found, the system searches the available general bindings for a binding with the name that you specified. If a general binding is found, the binding is used for the trust client request. If no binding with the name that you specified is found, then default bindings from the server environment are used. The default bindings are used only when the application is running in the server environment. If the application is running in a thin client environment, and no **wstrustClientBindingScope** property is specified, and no application-specific or general bindings are found, then no bindings are used because default bindings are not supported for a thin client.

Overview of application programming interfaces (APIs) for SAML:

WebSphere Application Server support for SAML provides public application programming interfaces (APIs) that you can use to build SAML token aware applications.

Use the SAMLTokenFactory API to create, validate, and authenticate SAML tokens, and to create JAAS subjects that represent SAML tokens. The SAMLTokenFactory implementation supports both the OASIS SAML v1.1 Token Specification and OASIS SAML v2.0 Token Specification. Use the WSSTrustClient API to send, issue, and validate WS-Trust request messages to the specified STS. The WSSTrustClient implementation supports both WS-Trust v1.3 Specification and the WS-Trust v1.2 Specification, and supports both the SOAP v1.1 namespace and the SOAP v1.2 namespace.

Note: Starting with WebSphere Application Server Release 8, you can use the `com.ibm.websphere.wssecurity.wssapi.token.SAMLToken` class in Web Services Security (WSS) application programming interface (API). When there is no concern of confusion we use the term `SAMLToken` instead of using its complete package name. You can use WSS API to request SAMLToken processing from an external Security Token Service (STS), to propagate SAMLTokens in SOAP request messages, and to use a symmetric or asymmetric key identified by SAMLTokens to protect SOAP messages.

The WSS API SAML support complements the `com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory` and `com.ibm.websphere.wssecurity.wssapi.trust.WSSTrustClient` interfaces. SAMLTokens that are generated using the `com.ibm.websphere.wssecurity.wssapi.WSSFactory newSecurityToken()` method can be processed by the `SAMLTokenFactory` and `WSSTrustClient` programming interfaces. Conversely, SAMLTokens that are generated by `SAMLTokenFactory` or returned by `WSSTrustClient` can be used in WSS API. Deciding which API to use in your application depends on your specific needs. WSS API SAML support is self contained in the sense that it provides functionality equivalent to that of the `SAMLTokenFactory` and `WSSTrustClient` interfaces as far as web services client applications are concerned. The `SAMLTokenFactory` interface has additional functions to validate SAMLTokens and to create the JAAS Subject that represents authenticated SAMLTokens. This validation is useful for the Web services provider side. When you develop applications to consume SAMLTokens, the `SAMLTokenFactory` programming interface is more suitable for you.

WebSphere Application Server with SAML provides the following APIs that implement SAML as a security token. For information about the methods in these APIs, refer to the SAML token library API documentation, which describes each of the APIs and provides sample code.

The SAMLTokenFactory API is the main SAML token programming interface. Using this API, you can create SAML tokens, insert SAML attributes, parse and validate SAML assertions as XML representations for the SAML tokens, and create JAAS subjects that represent user identity and attributes as defined in SAML tokens.

- com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory
- com.ibm.websphere.wssecurity.wssapi.token.SAMLToken

The SAMLAttribute and SAMLNameID System Programming Interfaces (SPIs) represent the SAML attributes and SAML user name identifiers.

- com.ibm.wsspi.wssecurity.saml.data.SAMLAttribute
- com.ibm.wsspi.wssecurity.saml.data.SAMLNameID

Using the following SAML SPIs, you can specify how SAML tokens are created, and SAML assertion XML documents are validated. The ProviderConfig objects specify configuration information for the SAML assertion issuer, which includes issuer name, issuer signing key, and signing certificate. The RequesterConfig objects contain configuration parameters that define the characteristics of SAML assertions and SAML tokens as they are created. These configuration parameters include confirmation method, signing of SAML assertions, embedded encryption key type, and authentication method. The ConsumerConfig objects contain configuration parameters that define how SAML assertion XML documents are validated, including decryption key information, encryption algorithm, timer value for clock skew, and if digital signing is required for issuers.

- com.ibm.wsspi.wssecurity.saml.config.ConsumerConfig
- com.ibm.wsspi.wssecurity.saml.config.CredentialConfig
- com.ibm.wsspi.wssecurity.saml.config.ProviderConfig
- com.ibm.wsspi.wssecurity.saml.config.RequesterConfig
- com.ibm.wsspi.wssecurity.saml.config.SamlConstants
- com.ibm.wsspi.wssecurity.core.token.config.ConsumerConfiguration
- com.ibm.wsspi.wssecurity.core.token.config.CredentialConfiguration
- com.ibm.wsspi.wssecurity.core.token.config.ProviderConfiguration
- com.ibm.wsspi.wssecurity.core.token.config.RequesterConfiguration

The following SAML Callback and CallbackHandler classes specify configuration parameters that you can use to define the characteristics of SAML assertions and control the behavior of the SAML LoginModule in the runtime environment. These parameters are stored in the Web Services Security binding documents. The SAML token configuration is modeled by the CustomToken extension in the Web Services Security policy configuration.

- com.ibm.websphere.wssecurity.callbackhandler.SAMLConsumeCallback
- com.ibm.websphere.wssecurity.callbackhandler.SAMLConsumerCallbackHandler
- com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallback
- com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler

The getXML() method, in the com.ibm.websphere.wssecurity.wssapi.token.SecurityToken programming interface, returns objects which implement the XMLStructure interface. An enhancement that is added by WebSphere Application Server supports a custom SecurityToken implementation. Using the following two SPIs, you can deploy an Axis2 Axiom OM implementation or a DOM implementation. The SAML token extends the GenericSecurityToken interface, which in turn extends the SecurityToken interface. GenericSecurityToken is a new interface added by WebSphere Application Server with SAML. The SAML token also implements the OMStructure interface.

- com.ibm.wsspi.wssecurity.wssapi.DOMStructure
- com.ibm.wsspi.wssecurity.wssapi.OMStructure

The WS-Trust Client API includes the WSSTrustClient class and other auxiliary APIs and SPIs. The WSSTrustClient API sends WS-Trust SOAP requests to a specified external security token service (STS) so that the STS can issue or validate one or more SAML assertions, or other type of security tokens. WSSTrustClient supports both WS-Trust Version 1.3 and WS-Trusts Version 1.2 specifications. The WSSTrustClient API returns a SAML token when API callers request the SAML token type. The API also uses the GenericSecurityToken interface when the API caller requests a non-SAML token type. Read about the WS-Trust Client API for more information and sample code. Refer to the API documentation for a detailed discussion of the APIs and SPIs.

- com.ibm.websphere.wssecurity.wssapi.trust.WSSTrustClient
- com.ibm.websphere.wssecurity.wssapi.token.GenericSecurityToken

Use the following SPIs specify the characteristics of tokens and their behavior in the runtime environment. The ProviderConfig objects contain configuration parameters that specify the STS endpoint, the Web Services Security policy set, the bindings documents used to access the STS, and whether general bindings or application-specific bindings are used. The ProviderConfig objects also specify whether the RequestSecurityTokenResponse (RSTR) XML document is stored in the GenericSecurityToken objects or SAMLToken objects. The RequesterConfig objects contain configuration parameters that are sent in the WS-Trust requests to the specified STS. The ConsumerConfig interface defines attributes and data for the WS-Trust response messages, and for the RequestSecurityTokenResponse element. This data is retrieved using the GenericSecurityToken getProperties() method. The RequestSecurityTokenResponse XML element is stored as a property when the ProviderConfig setIncludeRSTRProperties() has been invoked. If a specific attribute needed by an application is not defined by the ConsumerConfig interface, you can retrieve and parse the RSTR element instead.

- com.ibm.wsspi.wssecurity.trust.config.ConsumerConfig
  - com.ibm.wsspi.wssecurity.trust.config.ConsumerConfig.RSTR
- com.ibm.wsspi.wssecurity.trust.config.ProviderConfig
- com.ibm.wsspi.wssecurity.trust.config.RequesterConfig
- com.ibm.wsspi.wssecurity.core.token.config.WSSConstants
  - com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.Namespace
  - com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.TokenType
  - com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.WST12
  - com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.WST13
  - com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.Algorithm
  - com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.SAML

For additional information about using the APIs, including practical scenarios that illustrate how and when to apply the APIs, read about SAML usage scenarios.

SAML usage scenarios:

SAML function is described through four basic usage scenarios. The first three scenarios demonstrate web services single sign-on, configured using a policy set. The fourth scenario describes custom SAML single sign-on, which you can build using the SAML API and the trust client API. The scenarios demonstrate using SAML building blocks and APIs to authenticate to a Security Token Service (STS), request SAML tokens, and implement the SAML tokens to obtain access to business services.

Overview

WebSphere Application Server with SAML provides building blocks and APIs that enable you to create single sign-on and identity federation business solutions using SAML tokens. SAML policy sets are the building blocks for configuring web services applications to request SAML tokens, propagate SAML tokens in SOAP messages, and validate and authenticate SAML tokens, all without a single line of programming.

Using the SAML and WS-Trust client APIs, you can programmatically create SAML tokens, parse and validate SAML tokens, and authenticate SAML tokens received from protocols other than Web services SOAP messages. Specifically, use the SAML APIs to process custom SAML token attributes, create personalized application interfaces and perform claim-based authorization. Use the WS-Trust client API to request SAML tokens, or other types of tokens, from an STS, and to validate and exchange security tokens with an STS.

The WebSphere Application Server product does not include an STS for issuing SAML or any other type of security token. However, WebSphere Application Server with SAML does interoperate with Tivoli Federated Identity Manager and other third-party STS implementations.

Scenario 1: Web services single sign-on (SSO)

In this single sign-on (SSO) usage scenario, a user authenticates to an STS and requests SAML tokens using the bearer confirmation method. The user then uses SAML tokens to access a business services provider. The business services provider validates the SAML tokens and asserts the identity and attributes of the user based on the trust relationship between the provider and the issuing STS. The received SAML tokens are considered valid if the token signing certificates are trusted by the business services provider, and if the expiration time of the tokens is less than the business services provider local time plus a configurable clock skew.

The business services provider can access downstream services on behalf of the user using SAML tokens based on the policy configuration of the service. Using the policy configuration, the business services provider can either propagate the original SAML tokens or generate self-issued SAML based on the original user attributes. For a detailed description of how to configure policy sets and bindings for this scenario, read about configuring client and provider bindings for the SAML bearer token.

Single sign-on using a SAML bearer token has advantages over other SSO technology because SAML is an industry-standard security token, and is supported by multiple vendor products. In addition, the WebSphere Application Server implementation of SAML function interoperates with Tivoli Federated Identity Manager, DataPower®, and other vendor products.

This diagram shows the interaction between the STS and the business services provider in the web services single sign-on usage scenario.

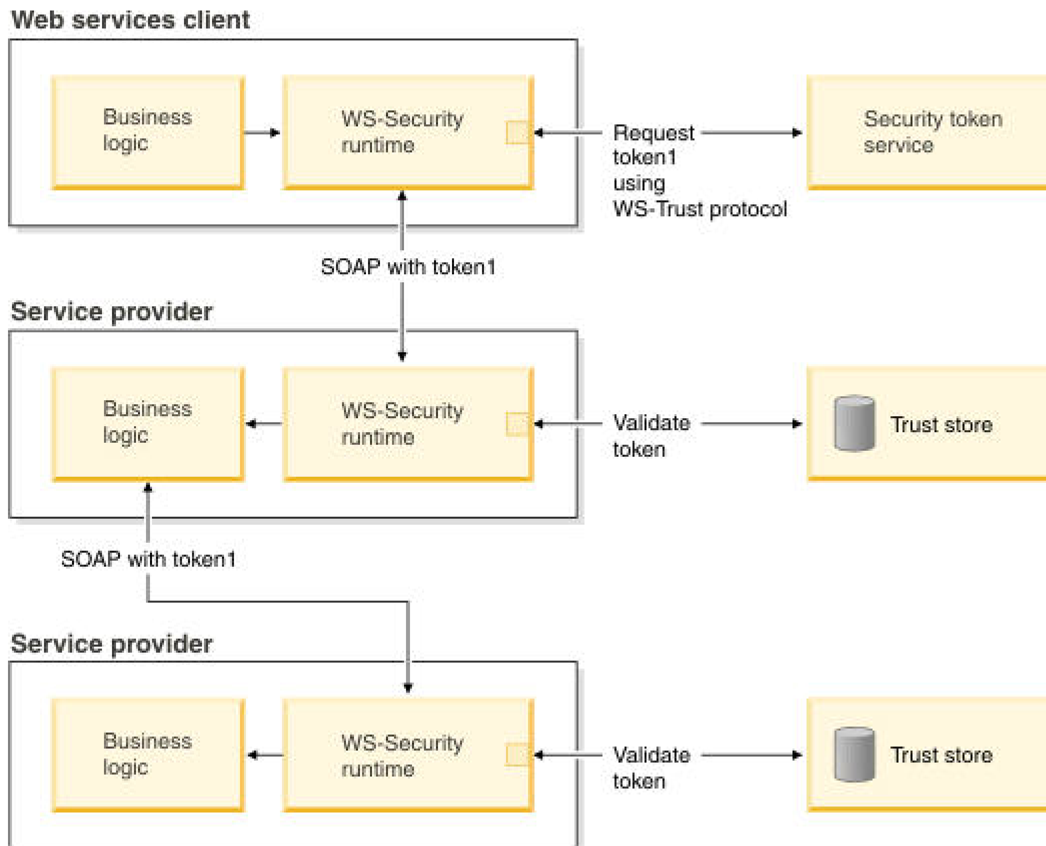


Figure 3. Web services single sign-on usage scenario

You can add a caller binding to the SAML provider sample bindings to select the received SAML token which represents the caller identity. The Web Services Security runtime environment uses the SAML NameId to represent the client identity and search the user security name and group membership from the configured user registry when you use the default system.wss.caller JAAS login configuration.

After this scenario is successfully executed, the WS-Security runtime environment saves the SAML tokens and can reuse them to access the same business service provider, if the tokens are still valid. The token is valid if the token expiration time is less than the cache period, which is configurable. Once the business services provider has validated and authenticated the received SAML tokens, the JAAS subjects, along with corresponding SAML tokens, are saved in the authentication cache. The SAML token remains valid in the hosting application server regardless of the token expiration time. This means that the WS-Security runtime environment does not check the SAML token expiration time within the same process, because the SAML tokens remain valid in the application server process as long as the token is in the authentication cache.

SAML token expiration times are checked when the business services provider uses the SAML tokens to invoke downstream services. Outbound requests will fail if the SAML token expiration time is not less than the current time added to the cache period. This limitation also applies when the policy configuration requires use of the original SAML tokens. However, the limitation does not apply if the policy configuration uses self-issued SAML tokens instead, for example, if the business services provider has created and signed a reproduction of the original SAML tokens.

Scenario 2: Web services single sign-on (SSO) with holder-of-key validation

The single sign-on with holder-of-key validation usage scenario is similar to the previous SSO usage scenario. The SSO with holder-of-key scenario uses SAML tokens with the holder-of-key (HoK)

confirmation method instead of the bearer confirmation method. SAML HoK tokens contain a key that the client uses to prove ownership of the key, and ownership of the token. This embedded key can be a shared key (also known as a symmetric key) or a X.509 certificate with a public key (also known as an asymmetric key). In the case of a shared key, the key is encrypted using the public key of the target business services provider so that only the intended business service can consume the SOAP messages.

When a client requests SAML tokens with the HoK shared key from an STS, the STS encrypts the key in the SAML tokens and sends a copy of the same key in the WS-Trust response to the requesting client. This is necessary because otherwise the client cannot read the encrypted keys in the SAML tokens. To prove ownership of the SAML tokens, the client uses the shared key to sign and to encrypt the SOAP request messages. Business services are required to validate the HoK token ownership by extracting the encrypted shared key and using it to verify the digital signature.

The following diagram shows how a web service can request a SAML token in the scenario.

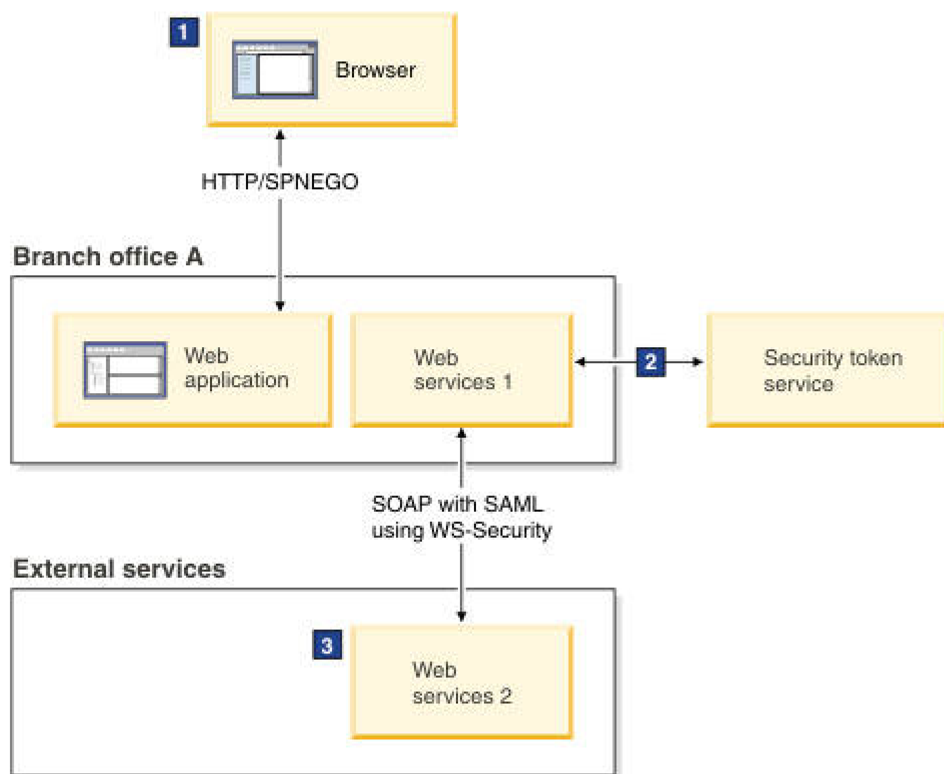


Figure 4. Web services 1 requesting a SAML token from an external service

1 The user logs on with a web browser using SPNEGO and is authenticated. A JAAS subject is created.

2 The credential from the SPNEGO token is used to request a SAML token using WS-Trust. The token is signed with the trust server private key.

3 The signature of the SAML token is validated based on the trust relationship. The security credential is created using the attributes from the SAML token. The cryptographic key from the SAML token is used to decrypt the SOAP message.

As shown in the diagram, a browser client uses Kerberos credentials (represented by SPNEGO token) to access a web application. Assuming that the Kerberos credential can be delegated, the web application can request a SAML token from the STS on behalf of the client, using the delegated client Kerberos credential. For example, the web application obtains a client approval request (APREQ) Kerberos token

from the key distribution center (KDC) on behalf of the client. The web application then uses the client APREQ token to authenticate to the STS and request a client SAML token from the STS on behalf of the client.

In this example, the SAML token requires the holder-of-key confirmation method using a symmetric key. The Web application uses the SAML token to access downstream business services, also on behalf of the client, so the SAML token authenticates the client to the downstream services and also protects the request messages using digital signing and encryption. For more information about how to set up bindings for the HoK token, read about configuring client and provider bindings for the SAML holder-of-key symmetric key token.

When the scenario is successfully executed, the target business services can then use the SAML token to access other downstream services using the same procedure described in the scenario, if the downstream business services can extract the embedded key. Alternatively, the business services can be configured to use self-issued SAML tokens to access downstream services to avoid distributing the private key.

Scenario 3: Web services federated identity management

In the federated identity management usage scenario, a browser client accesses a company portal web application. In this scenario, the basic user authentication data, such as user name and password, are used to request SAML tokens from an STS, and then the SAML tokens are used to obtain access to business services across security domains. The receiving business services provider validates the SAML tokens based on the trust relationship between the provider and the issuing STS, and the provider also asserts the identity and attributes of the user. This means that the user does not need to be previously defined in the user directory in the target business services. An advantage of this scenario is that it provides a more manageable way to federate business services from many business partners together, while removing the requirement to consolidate users into one user directory service.

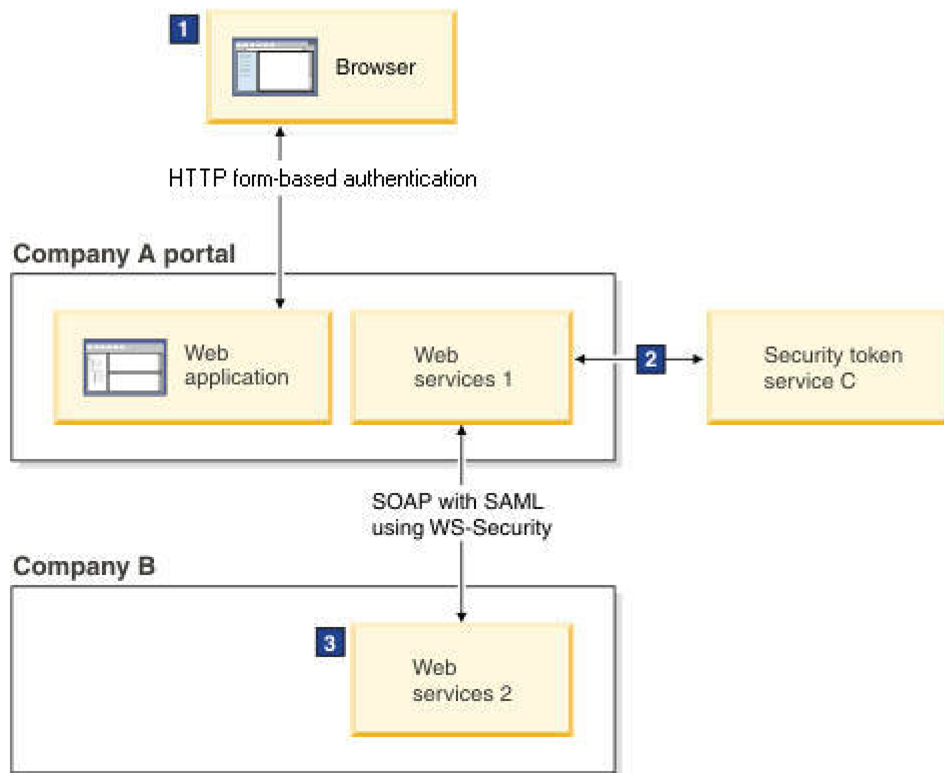


Figure 5. A user logs on to a company portal and uses federated identity management to obtain access to business services

1 The user logs on with a user name and password and is authenticated to the company portal. A JAAS subject is created.

2 The user name and password are used to authenticate to the STS and to request a SAML token representing the user.

3 The signature of the SAML token is validated based on the trust relationship. Security credentials are created using the attributes from the SAML token. The cryptographic key from the SAML token is used to decrypt the SOAP message.

The default system login configuration, wss.caller, maps the user identities that are represented by SAML tokens to user identities in the configured user directory. A custom login module must be added to the wss.caller login configuration to assert SAML token user identities from other security domains to the application server. This custom login module extracts the SAML token user identity and other attributes and constructs the assertion properties used by the application server. Two of these properties, user name and user identity, are required by the application server. Because the two properties are provided in the identity assertion, the application server does not need to validate the user name against the local user directory.

In addition, information about user group membership can be provided to the application server. This information is used to construct the WSCredential object representing application server security credentials for the user. The user properties are passed to the application server WSWSSLoginModule using the JAAS LoginContext shared state. For detailed information about these properties, read about configuring inbound identity mapping.

Scenario 4: Custom single sign-on (SSO) solutions

The custom single sign-On (SSO) usage scenario uses the SAML token library APIs, the WS-Trust client APIs, and other application server APIs and SPIs to build customized SSO solutions to fit a specific business computing environment. The diagram for this scenario shows an example where users are authenticated and issued SAML tokens from an identity provider that has trust relationship with a company server. In this scenario, a company wants to grant users access to protected web applications based on the trust relationship, without asking the users for additional authentication.

Web application clients, such as web browsers, pass SAML tokens to the application server using the HTTPS protocol, instead of using the web services protocol. To intercept and pass along these requests, the company builds a SAML trust association interceptor (TAI) that implements the application server Trust Association Interceptor API. A SAML TAI uses the SAML token library API to validate and authenticate SAML Tokens. Alternately, the SAML TAI can use the WS-Trust client API to validate and authenticate SAML tokens against an external STS. For more information about the TAI interface, read about developing a custom interceptor for trust associations. A custom SAML TAI is not shipped with WebSphere Application Server.

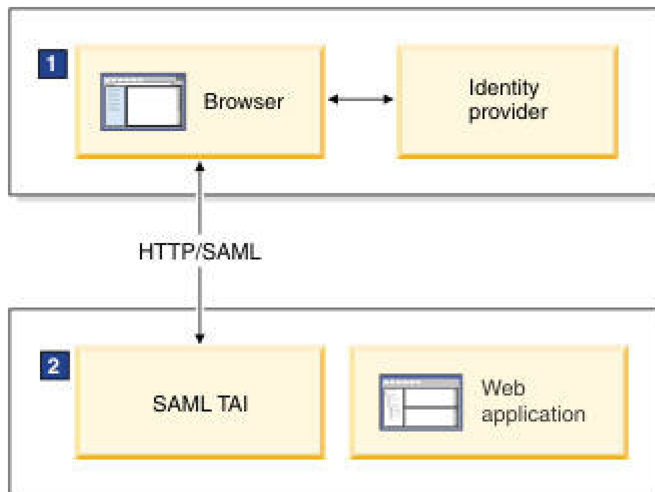


Figure 6. A user logs on to a company server with a web browser using the HTTPS protocol, and is authenticated using a SAML TAI

1 The user logs on to an identity provider using a user name and password. The identity providers issues a SAML token to the user.

2 The SAML TAI uses the SAMLTokenFactory API to validate the SAML token, and to authenticate the SAML token to create user credentials. The SAML TAI can also use the WS-Trust client API to validate the SAML token with an external STS.

For more information about the SAML APIs, read about the WS-Trust client API and the SAML token library APIs.

A more complex solution: Multiple security domains

The previous sections of this document described four basic usage scenarios. However, you might want to assert SAML tokens across multiple security domain boundaries. For more information on this solution, see the documentation about SAML assertions across WebSphere Application Server security domains.

Limitations of the SAML implementation:

Limitations of the SAML implementation are described. These limitations refer to functions that are currently implemented and supported by WebSphere Application Server Version 8.0 and later.

- The WSTrustClient API supports issue and validate operations, but does not support the cancel and renew operations.
- WebSphere Application Server with SAML does not support propagating a SAML token in the response message from a web services provider to a web services client.

Generic security token login modules

The *generic security token login modules* are Java Authentication and Authorization Service (JAAS) login modules. These login modules issue, validate, and exchange security tokens using an external Security Token Service (STS).

Overview

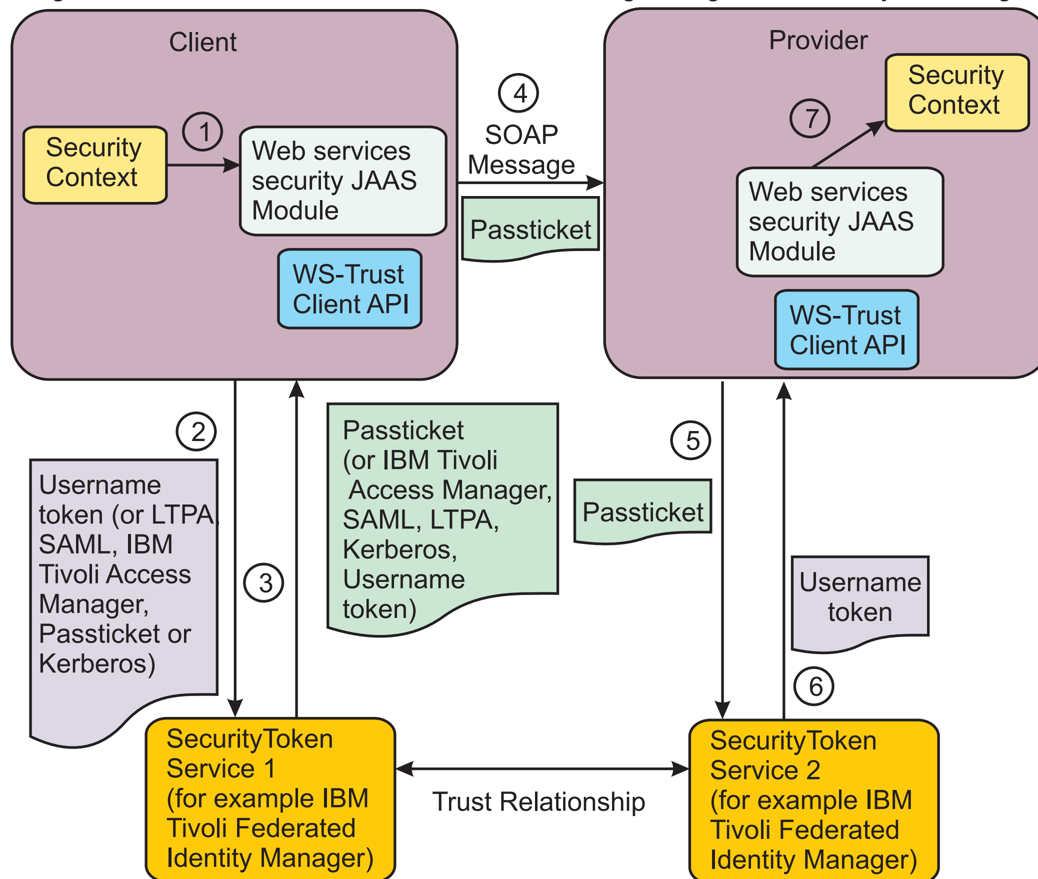
The Web Services Security token generation and consuming processes invoke these login modules. The Web Services Security component provides default login modules for common tokens such as the following examples:

- Username tokens
- X.509 tokens
- Kerberos tokens
- Lightweight Third Party Authentication (LTPA) tokens
- Security Assertion Markup Language (SAML) tokens
- Security context tokens

For more information on the token implementations, see the default implementations of the Web Services Security service provider programming interfaces documentation.

Note: If you are using the IBM Tivoli Federated Identity Manager as an external Security Token Service, you should use Versions 6.2.0.9, 6.2.1.2, 6.2.2 or later to prevent LTPA token exchange failures.

The following illustration shows the flow of information through the generic security token login module



process.

1. The caller's identity is inherited by the runtime environment of the web services client.
2. The generic security token login module *for the token generator* sends a token request to a WS-Trust service using a WS-Trust client using either an issue or validate request.
3. The returned or validated token is set in the security header of the SOAP message as an authentication token. For more information, see the documentation about the generic security token login modules for the token generator.
4. The PassTicket is sent as part of the SOAP message to the service provider.
5. The generic security token login module *for the token consumer* sends the received token in the security header of the SOAP message within a WS-Trust Validate request to a designated WS-Trust service.
6. The request might result in a new token or in a notification that the sent token has been validated successfully.
7. As required, the new or originally validated token is used as the caller token for authorization purposes. For more information, see the documentation about the generic security token login modules for the token consumer.

A *PassTicket* is a dynamically generated, one-time use, substitute password. You can use the PassTicket to authenticate to a service rather than sending the actual password.

Usage scenarios

The generic security token login module might be very useful if token exchange, identity mapping, or authorization to invoke a target web service are required. The following list explains some useful usage scenarios for a generic security token login module:

Token exchange with an intermediate server
The required outgoing security token and the incoming security token are different types.

Token exchange on the requesting side
An identity mapping for the requestor is required before invoking a downstream service.

Token exchange on the receiving side
The invoking identity mapping is required after the token is validated.

Authorization to invoke target service
The login module sends the incoming security token and its target service endpoint address to the WS-Trust service. The WS-Trust service completes the web service-level authorization. The WS-Trust service verifies whether the target web service invocation is authorized for the principal that is contained within the authentication token.

Limitations

The following limitations exist for the generic login modules:

- You can use the token, which is processed by the generic security token login module, for authentication only. You cannot use the token as a protection token to digitally sign and encrypt message parts.
- If the service provider receives an exchanged token, the token must be supported by the default login modules for the application server Web Service Security system. For more information, see the documentation about the generic security token login module for the token consumer.
- If the service provider receives a token that is validated and not exchanged, the received token must be supported by the default login modules for the application server Web Service Security system.
- When you use a security token from the RunAs Subject to validate or exchange for an outbound security token, the security token within the RunAs Subject must be uniquely identified by a token ValueType value. If multiple tokens in the RunAs Subject have the same ValueType value, the login module does not use WS-Trust Validate to exchange a token with the RunAs Subject. Instead, the login modules use WS-Trust Issue to request a token that is based on the configuration of the policy set for the trust client.

Generic security token login module for the token generator

When a web service request is made, the application server calls the generic security login module for the token generator as part of the Web Service Security authentication process.

The login module delegates the token generation process to a Security Token Service (STS) through a WS-Trust Issue or WS-Trust Validate request. The STS processes the request and returns a RequestSecurityTokenResponse message to the login module. The login module includes the token from the STS response message in the security header of the web service request message. If a token is not returned or an error occurs from the STS call, then the login module produces a LoginException message and an error is returned to the web services client.

The login module, and its use of the Security Token Service, permits the following actions:

- An exchange of security tokens when the incoming or outgoing security tokens are different token types
- An exchange of security tokens when mapping one identity to another identity
- The evaluation of authorization checks to ensure that the authenticated users are permitted to invoke the target web service
- The token exchange is invoked from the RunAs Subject or generated by the Web Services Security runtime environment. The exchange is based on the policy set and bindings that are configured for the trust request.

To use the generic security token login module for the token generator, the token generator in the Web Services Security policy set bindings must:

- Specify the proper Java Authentication and Authorization Service (JAAS) login configuration name
- Specify the callback handler class name

The JAAS login configuration name is wss.generate.issuedToken, and the callback handler class name is com.ibm.websphere.wssecurity.callbackhandler.GenericIssuedTokenGenerateCallbackHandler. For more information, see the documentation about configuring a generic login module for an authentication token on the token generator side of the Web Services Security process.

Supported token types

- You can specify any token type whose ValueType value can be processed by the designated STS. Depending on which STS you use, the token types might include:
  - Security Assertion Markup Language (SAML) 2.0
  - SAML 1.1
  - Username
  - PassTicket
  - Kerberos
  - Lightweight Third Party Authentication (LTPA)
  - Tivoli Access Manager credential
- The requested token that is sent in the SOAP message to the service provider is the token that is specified in the policy.
- You can use this token for authentication only. You cannot use this token as a protection token. For SAML Version 2.0, 1.1, and 1.0, only bearer and send voucher confirmation methods are supported.

You can configure the generic security token login module for the token generator to use either a WS-Trust Issue or WS-Trust Validate request to exchange or validate the security token. These two options are described in subsequent sections.

WS-Trust Issue

You can configure the login module for the token generator to use WS-Trust Issue to request a security token. In this scenario, the trust client sends an authentication security token to an STS in the SOAP security header. This authentication token originates from one of the following locations:

- The RunAs Subject in the current security context
- The callback handler that is configured within the bindings for the trust client policy sets

Upon successful processing of the STS request, the STS authenticates the token and issues the requested token.

WS-Trust Validate

You can optionally configure the login module for the token generator to use WS-Trust Validate to request a security token. In this scenario, the login module searches for the authentication security token from the RunAs Subject based on the configured token ValueType value. The login module sends the token in the trust request by embedding it inside the RequestedSecurityToken element as a child element. This token might be wrapped inside either the ValidateTarget element or the Base extension element. The STS validates the embedded token inside the RequestedSecurityToken element and returns a new security token or a validation status code. If only a validation status code is returned, the token generator uses the original security token. Although the returned token can have any ValueType value, as previously described in the WS-Trust Issue usage scenario, the token to be validated must be one of the following token types:

- SAML 2.0
- SAML 1.1
- Username

- PassTicket
- Kerberos
- LTPA
- LTPA Version 2

Use WS-Trust Issue or WS-Trust Validate

The generic login module uses WS-Trust Validate to validate the token from the RunAs Subject if the following conditions are both true:

- A RunAs Subject exists in the current security context
- Only one security token exists whose value type matches the ValueType value for the requested token

If WS-Trust Validate returns a valid status code and a security token, the returned token is the requested token. If WS-Trust Validate returns a valid status code only, the existing token from the RunAs Subject is the requested token.

Also, you can select a token from the RunAs Subject for validation and exchange it for the requested token. The selected token can have a different ValueType value from the requested token. For more information, see the documentation about configuring a generic security token login module for an authentication token on the token generator side of the Web Services Security process.

Note: If the ValueType value for the requested token is an LTPA or LTPA Version 2 type, the generic security token login module automatically extracts a WSCredential. It generates a Web Service Security LTPA or LTPA v2 token for validation and exchange, if the following conditions are true:

- An LTPA or LTPA v2 security token does not exist in the RunAs Subject.
- An WSCredential exists in the RunAs Subject.

When there is only one security token in RunAs Subject that matches the ValueType of the requested token, you can configure the login module to not invoke a WS-Trust Validate request to validate the matching token. Instead, the login module sends the matching token to the downstream service provider without validation. The generic security token login module automatically uses WS-Trust Issue to request the token, if the following conditions are true:

- A RunAs Subject does not exist
- A matching token ValueType value does not exist in the RunAs Subject
- The login module cannot validate the token from the RunAs Subject

A configuration option enforces the use of either the WS-Trust Issue in the generic login module or the WS-Trust Validate. For more information, see the documentation about configuring a generic login module for an authentication token on the token generator side of the Web Services Security process.

Policy sets

The implementation of the generic security token login module does not involve a new token type in a policy set. For example, if you plan to use a generic login module to generate a user name token, you can create a policy set that specifies a user name token as an authentication token. Some custom token types are not supported by the existing default system login modules. However, you can implement these token types using custom login modules. These custom token types are supported by generic security token login modules if they are supported by the designated STS.

Bindings

When you configure bindings for an authentication token, you have the following options:

- Use a generic login module.

- Use an existing system default login module.
- Create your own custom login module.

For example, if you configure a user name token, you can use the wss.generate.unt JAAS login configuration and maintain the existing behavior. However, you can configure the wss.generate.issuedToken JAAS login to use the generic security token login module.

Generic security token login module for the token consumer

When a web service message is received, the application server calls the generic security token login module for the token consumer as part of the Web Services Security authentication process.

The login module delegates the token validation process to the WS-Trust service using WS-Trust Validate. The WS-Trust Security Token Service processes the request and returns a RequestSecurityTokenResponse message to login module, which might contain a new security token or validation status code only. The returned token from the WS-Trust Security Token Service or the original received token is the caller token if the caller token is required.

If the trust service call returns an invalid status code or an error, the token validation process fails and the login module produces a LoginException exception.

The login module, and its use of the WS-Trust Service, permits the following actions:
- The exchange of security tokens when the incoming or outgoing security tokens are different types
- The exchange of security tokens when you map one identity to another identity
- The evaluation of authorization checks to ensure that authenticated users are permitted to invoke the target web service

The Java Authentication and Authorization Service (JAAS) login configuration name is wss.consume.issuedToken, and the callback handler class name is com.ibm.websphere.wssecurity.callbackhandler.GenericIssuedTokenConsumeCallbackHandler.

Supported token types

The receiving token must have a ValueType value that the designated trust service can handle and exchange. The valid token ValidType value might be a known token type that is supported by system default login modules. The valid incoming tokens can be a user name token, an XML token, or a binary security token, including the following token types:
- Security Assertion Markup Language (SAML) 2.0
- SAML 1.1
- Username
- PassTicket
- Kerberos
- Lightweight Third Party Authentication (LTPA)
- Tivoli Access Manager credential

However, if WS-Trust Validate does not complete the token exchange and returns a validation status code only, the incoming token type must be one of following token types:
- SAML 2.0
- SAML 1.1
- Username
- Kerberos
- LTPA v2
- LTPA

Also, the return token value type from WS-Trust call must be one of the previous token types.

Note:
- The received token that is sent by the requesting party is the token that is specified in the policy.
- You can use this token for authentication only. You cannot use this token as a protection token.

Policy sets

The implementation of the generic security token login module can support any authentication tokens that are supported by system default login modules or by a custom login module. The generic security token login module implementation does not add a new security token type in the policy set. For example, if you plan to use a generic security token login module to generate a user name token, you can create a policy set that specifies a user name token as an authentication token. Any token types that are supported by designated security token services can be used with the generic security token login modules. You can implement custom login modules to process any new token types that are not supported by the existing default system login modules.

Bindings

When you configure bindings for an authentication token, you have the following options:
- Use a generic login module.
- Use an existing system default login module.
- Create your own custom login module.

For example, if you configure a user name token, you can use the wss.consume.unt JAAS login configuration and maintain the existing behavior. However, you can configure the wss.consume.issuedToken JAAS login to use the generic login module.

Web Services Security concepts for Version 5.x applications

IBM® supports Web Services Security, which is an extension of the IBM web services engine, to provide a quality of service. The WebSphere® Application Server security infrastructure fully integrates Web Services Security with the Java™ Platform, Enterprise Edition (Java EE) security specification.

Web Services Security specification—a chronology:

This chronology describes the process that has been used to develop the Web Services Security specifications. The chronology includes both the Organization for the Advancement of Structured Information Standards (OASIS) and non-OASIS activities.

Non-OASIS activities

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* applications.

In April 2002, IBM, Microsoft, and VeriSign proposed the *Web Services Security (WS-Security) specification* on their websites. This specification included the basic ideas of security token, XML signature, and XML encryption. The specification also defined the format for user name tokens and encoded binary security tokens. After some discussion and an inter-operability test that was based on the specification, the following issues were noted:
- The specification requires that the Web Services Security processors understand the schema correctly so that the processor distinguishes between the ID attribute for XML signature and XML encryption.

- The freshness of the message, which indicates whether the message complies with predefined time constraints, cannot be determined.
- Digested password strings do not strengthen security.

In August 2002, IBM, Microsoft, and VeriSign published the *Web Services Security Addendum*, which attempted to address the previously listed issues. The following solutions were put in the addendum:
- Require a global ID attribute for XML signature and XML encryption.
- Use time stamp header elements that indicate the time of the creation, receipt, or expiration of the message.
- Use password strings that are digested with a timestamp and nonce (randomly generated token).

OASIS activities

In June 2002, OASIS received a proposed Web Services Security specification from IBM, Microsoft, and Verisign. The Web Services Security Technical Committee (WSS TC) was organized at OASIS soon after the submission. The technical committee included many companies including IBM, Microsoft, VeriSign, Sun Microsystems, and BEA Systems.

In September 2002, WSS TC published its first specification, *Web Services Security Core Specification, Working Draft 01*. This specification included the contents of both the original Web Services Security specification and its addendum.

The coverage of the technical committee became larger as the discussion proceeded. Since the Web Services Security Core Specification allows arbitrary types of security tokens, proposals were published as profiles. The profiles described the method for embedding tokens, including Security Assertion Markup Language (SAML) tokens and Kerberos tokens imbedded into the Web Services Security messages. Subsequently, the definitions of the usage for user name tokens and X.509 binary security tokens, which were defined in the original Web Services Security Specification, were divided into the profiles.

WebSphere Application Server supports the following specifications:
- Web Services Security: SOAP Message Security Draft 13 (formerly Web Services Security Core Specification)
- Web Services Security: Username Token Profile Draft 2

The following figure shows the various Web Services Security-related specifications. As indicated in the figure, the current support level for Web Services Security: SOAP message security is based on Draft 13 from May 2003. The current support level for Web Services Security user name token profiles, is based on Draft 2 from February 2003.

Figure 7. Web Services Security specification support

Web Services Security support:

IBM supports Web Services Security, which is an extension of the IBM Web services engine, to provide a quality of service. The WebSphere Application Server security infrastructure fully integrates Web Services Security with the Java Platform, Enterprise Edition (Java EE) security specification.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server, Versions 4.*x*, 5, and 5.0.1 support digital signature for Apache SOAP Version 2.*x*. Beginning with WebSphere Application Server, Version 5.0.2, IBM supports Web Services Security. The IBM implementation is based on the Web Services Security specification, Web Services Security (WS-Security), originally proposed by IBM, Microsoft, and VeriSign in April 2002. Early versions of the proposed draft specification can be found in Web Services Security (WS-Security) Version 1.0 05 April 2002 and Web Services Security Addendum 18 August 2002. The WebSphere Application Server implementation is based on the Organization for the Advancement of Structured Information Standards (OASIS) working Draft 13 specification. (See the OASIS Web Services Security TC website for the latest working specification.) However, not all the features in the OASIS working Draft 13 specification are implemented.

Web Services Security is not supported in a pure Java client or a nonmanaged client. When a user ID and password are embedded in a request message, authentication is performed with the user ID and password. If authentication is successful, a user identity is established and further resource access is authorized based on that identity. After the user ID and password are authenticated by the Web Services Security run time, a Java EE container performs authorization.

WebSphere Application Server provides an implementation of the key features of Web Services Security based on the following specifications:

- Specification: Web Services Security (WS-Security) Version 1.0 05 April 2002
- Web Services Security Addendum 18 August 2002
- Web Services Security: SOAP Message Security Working 13 May 2003
- Web Services Security: Username Token Profile Draft

The following table provides a summary of Web Services Security elements supported by WebSphere Application Server:

Table 34. Supported Web Services Security elements. Use the table to determine which security elements are supported.

Element	Notes®
UsernameToken	Both the user name and password for the BasicAuth authentication method and the user name for the identity assertion authentication method are supported. WebSphere Application Server supports nonce, a randomly generated value.
BinarySecurityToken	X.509 certificates and Lightweight Third Party Authentication (LTPA) can be embedded, but there is no implementation to embed Kerberos tickets. However, the binary token generation and validation are pluggable and are based on the Java Authentication and Authorization Service (JAAS) Application Programming Interfaces (APIs). You can extend this implementation to generate and validate other types of binary security tokens.
Signature	The X.509 certificate is embedded as a binary security token and can be referenced by the SecurityTokenReference. WebSphere Application Server does not support shared, key-based signature.
Encryption	Both the EncryptedKey and ReferenceList XML tags are supported. KeyIdentifier specifies public keys and KeyName identifies the secret keys. WebSphere Application Server has the capability to map an authenticated identity to a key for encryption or use the signer certificate to encrypt the response message.
Time stamp	WebSphere Application Server supports the Created and Expires attributes. The freshness of the message, which indicates whether the message complies with predefined time constraints, is checked only if the Expires attribute is present in the message. WebSphere Application Server does not support the Received attribute, which is defined in the addendum. Instead, WebSphere Application Server uses the TimestampTraceReceived attribute, which is defined in the OASIS specification.
XML-based token	You can insert and validate an arbitrary format of XML tokens into a message. This format mechanism is based on the JAAS APIs.

Signing and encrypting attachments is not supported by WebSphere Application Server. However, WebSphere Application Server signs and encrypts the following elements for the request message.

Table 35. Elements that are signed and encrypted for the request message. The elements are used to perform authentication.

Method	Element
XML digital signature	• Body • Securitytoken • Timestamp
XML encryption	• Bodycontent • Usernametoken
AuthMethod	• BasicAuth • IDAssertion (from WebSphere Application Server to another WebSphere Application Server • Signature • Lightweight Third Party Authentication (LTPA) on the server side • Other customer tokens

WebSphere Application Server signs and encrypts the following elements for the response message:

Table 36. Elements that are signed and encrypted for the response message. The elements are used to perform authentication.

Method	Element
XML digital signature	• Body • Timestamp
XML encryption	• Bodycontent

The namespaces that are used for sending a message were published by OASIS in draft 13 (http://schemas.xmlsoap.org/ws/2003/06/secext). However, the Web Services Security run time in WebSphere Application Server can accept any of the following namespaces:

April 2002 specification
http://schemas.xmlsoap.org/ws/2002/04/secext

August 2002 addendum
http://schemas.xmlsoap.org/ws/2002/07/secext

http://schemas.xmlsoap.org/ws/2002/07/utility

OASIS draft published on draft 13 May 2003
http://schemas.xmlsoap.org/ws/2003/06/secext

http://schemas.xmlsoap.org/ws/2003/06/utility

Note: WebSphere Application Server only uses the previously mentioned two namespaces for sending out requests and responses. However, the product can process all other mentioned namespaces for incoming requests and responses.

WebSphere Application Server provides the following capabilities for Web Services Security:

- Integrity of the message
- Authenticity of the message
- Confidentiality of the message
- Privacy of the message
- Transport level security: provided by Secure Sockets Layer (SSL)
- Security token propagation (pluggable)
- Identity assertion

Refer to the Web Services Security elements table for a description of capabilities that are not supported.

Web Services Security and Java Platform, Enterprise Edition security relationship:

This article describes the relationship between Web Services Security (message level security) model and the Java Platform, Enterprise Edition (Java EE) security model. It also includes information on Java EE role-based authorization checks.

Important: There is an important distinction between Version 5.x and Version 6 and later applications. The information in this article supports Version 5.x applications only that are used with WebSphere Application Server Version 6.0.x and later. The information does not apply to Version 6.0.x and later applications.

WebSphere Application Server supports Java Specification Requests (JSR) 101 and JSR 109. JSRs 101 and 109 define web services for the Java EE architecture so that you can develop and run web services on the Java EE component architecture.

Important: Web Services Security refers to the Web Services Security: SOAP Message Security specification. For more information, see “Web Services Security support” on page 322.

Securing web services with WebSphere Application Server security (Java EE role-based security)

You can secure web services using the existing security infrastructure of WebSphere Application Server, Java EE role-based security, and Secure Sockets Layer (SSL) transport level security.

Figure 8. SOAP message flow using existing security infrastructure

The web services port can be secured using Java EE role-based security. The web services sender sends the basic authentication data using the HTTP header. SSL (HTTPS) can be used to secure the transport. When the WebSphere Application Server receives the SOAP message, the web container authenticates the user (in this example, `user1`) and sets the security context for the call. After the security context is set, the SOAP router servlet sends the request to the implementation of the web services (the implementation can be JavaBeans or enterprise bean files). For enterprise bean implementations, the EJB container performs an authorization check against the identity of `user1`.

The web services port also can be secured using the Java EE role. Then, authorization is performed by the web container before the SOAP request is dispatched to the web services implementation.

Securing web services with Web Services Security at the message level

You can also secure web services using Web Services Security at the message level. In this case, you can digitally sign or encrypt a certain part of the message. Web services security also supports security token propagation within the SOAP message. The following scenario assumes that the web services port is not secured with Java EE role-based security and the enterprise bean is secured with Java EE role-based security.

Figure 9. SOAP message flow using Web Services Security

In this case, the web services port is not secured with Java EE role-based security. The web services engine processes the SOAP message before the client sends the message to the web services port. The Web Services Security run time acts on the security constraints, such as digitally signing, encrypting, or generating (and inserting) a security token in the SOAP header. In this case `<wsse:UsernameToken>` is generated using `user1` and the password. On the server-side (receiving), the web services process the incoming message and Web Services Security enforces security constraints. This enforcement includes making sure that messages are properly signed, properly encrypted, and decrypted, authenticating the

security token, and setting up the security context with the authenticated identity. (In this case, `user1` is the authenticated identity.) Finally, the SOAP message is dispatched to the web services implementation (if the implementation is an enterprise beans file, the Enterprise JavaBeans (EJB) container performs an authorization check against `user1`). SSL also might be used in this scenario.

Mixing the two

The second scenario shows that Web Services Security can complement Java EE role-based security. For example, SSL can be enabled at the transport level to provide a secure channel. Furthermore, if the web services implementation is an enterprise beans file, you can leverage the EJB role-based authorization by performing authorization checks. Web services security run time leverages the security infrastructure to set the authenticated identity in the security context. The authenticated identity can be used in the downstream call to Java EE resources (or other resource types).

There are subtle consequences of combining the two scenarios. For example, if the HTTP transport is sending basic authentication data with `user1` and password in the HTTP header, but `<wsse:UsernameToken>` with `user99` and `letmein` also is inserted into the SOAP header. In the previous scenarios, there are two authentications performed. One authentication is performed by the web container for authenticating `user1`, and the other is performed by Web Services Security for authenticating `user99`. The Web Services Security run time runs after the web container runs and `user99` is the authenticated identity that is set in the security context.

Web Services Security can also propagate security tokens from the sender to the receiver for SOAP over a Java Message Service (JMS) transport.

Java EE role-based authorization checks

A standard for authorization does not exist for web services. However, IBM, in conjunction with Microsoft Corporation, jointly published a security white paper road map for web services called Security in a Web Services World: A Proposed Architecture and Roadmap in which a proposal exists for the WS-Authorization specification. However, the WS-Authorization specification has not been published.

The existing implementation of Web Services Security is based upon the Web Services for Java EE specification or the Java Specification Requirements (JSR) 109 specification. The implementation of Web Services Security leverages the Java EE role-based authorization checks. For conceptual information, read about role-based authorization. If you develop a web service that requires method-level authorization checks, then you must use stateless session beans to implement your web service. For more information about using stateless session beans to implement a web service, read the topics "Implementing web services applications with JAX-WS" or "Implementing web services applications with JAX-RPC," depending on your programming model. Also, read about securing enterprise bean applications. If you develop a web service that is implemented as a servlet, you can use coarse-grained or URL-based authorization in the web container. However, in this situation, you cannot use the identity from Web Services Security for authorization checks. Instead, you can use the identity from the transport. If you use SOAP over HTTP, then the identity is in the HTTP transport.

Web Services Security model in WebSphere Application Server:

The Web Services Security model used by WebSphere Application Server is the declarative model. WebSphere Application Server does not include any application programming interfaces (APIs) for programmatically interacting with Web Services Security. However, a few Server Provider Interfaces (SPIs) are available for extending some security-related behaviors.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

Figure 10. Web Services Security model

The security constraints for Web Services Security are specified in IBM deployment descriptor extensions for Web services. The Web Services Security run time acts on the constraints to enforce Web Services Security for the SOAP message. The scope of the IBM deployment descriptor extension is at the enterprise bean (EJB) or web module level. Bindings are associated with each of the following IBM deployment descriptor extensions:

Client (Might be either a Java Platform, Enterprise Edition (Java EE) client (application client container) or web services acting as a client)

`ibm-webservicesclient-ext.xmi`

`ibm-webservicesclient-bnd.xmi`

Server

`ibm-webservices-ext.xmi`

`ibm-webservices-bnd.xmi`

It is recommended that you use the assembly tools provided by IBM to create the IBM deployment descriptor extension and bindings. After the bindings are created, you can use the administrative console or an assembly tool to specify the bindings.

Important: The binding information is collected after application deployment rather than during application deployment. The alternative is to specify the required binding information before deploying your application.

Figure 11. Web Services Security message interpretation

The Web Services Security run time enforces Web Services Security based on the defined security constraints in the deployment descriptor and binding files. Web Services Security has the following four points where it intercepts the message and acts on the security constraints defined:

Table 37. Web Services Security message points. The descriptions of the points provides examples of Web Services Security runtime environment behavior.

Message points	Description
Request sender (defined in the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` files)	• Applies the appropriate security constraints to the SOAP message (such as signing or encryption) before the message is sent, generating the time stamp or the required security token.
Request receiver (defined in the `ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` files)	• Verifies that the Web Services Security constraints are met. • Verifies the freshness of the message based on the time stamp. The freshness of the message indicates whether the message complies with predefined time constraints. • Verifies the required signature. • Verifies that the message is encrypted and decrypts the message if encrypted. • Validates the security tokens and sets up the security context for the downstream call.
Response sender (defined in the `ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` files)	• Applies the appropriate security constraints to the SOAP message response, like signing the message, encrypting the message, or generating the time stamp.

Table 37. Web Services Security message points (continued). The descriptions of the points provides examples of Web Services Security runtime environment behavior.

Message points	Description
Response receiver (defined in the `ibm-webservicesclient-ext.xmi` or `ibm-webservicesclient-bnd.xmi` files)	• Verifies that the Web Services Security constraints are met. • Verifies the freshness of the message based on the time stamp. The freshness of the message indicates whether the message complies with predefined time constraints. • Verifies the required signature. • Verifies that the message is encrypted and decrypts the message, if encrypted.

Propagating security tokens:

In this example, security tokens are propagated using Web Services Security, the security infrastructure of the WebSphere Application Server, and Java Platform, Enterprise Edition (Java EE) security.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

An example scenario

In this example, Client 1 invokes Web Services 1. Then Web Services 1 calls the Enterprise JavaBeans (EJB) file 2. The EJB file 2 calls Web Services 3 and Web Services 3 calls Web Services 4.

Figure 12. Propagating security tokens

The previous figure shows security tokens propagated using Web Services Security, the security infrastructure of the WebSphere Application Server, and Java Platform, Enterprise Edition (Java EE) security. Web Services 1 is configured to accept <wsse:UsernameToken> only and use the BasicAuth authentication method. However, Web Services 4 is configured to accept either <wsse:UsernameToken> using the BasicAuth authentication method or Lightweight Third Party Authentication (LTPA) as <wsse:BinarySecurityToken>. The following steps describe the scenario shown in the previous figure:

1. Client 1 sends a SOAP message to Web Services 1 with `user1` and `password` in the <wsse:UsernameToken> element.
2. The `user1` and `password` values are authenticated by the Web Services Security run time and set in the current security context as the Java Authentication and Authorization Service (JAAS) Subject.
3. Web Services 1 invokes EJB file 2 using the Remote Method Invocation over the Internet Inter-ORB Protocol (RMI/IIOP) protocol.
4. The `user1` identity is propagated to the downstream call.
5. The EJB container of EJB file 2 performs an authorization check against `user1`.
6. EJB file 2 calls Web Services 3 and Web Services 3 is configured to accept LTPA tokens.
7. The RunAs role of EJB file 2 is set to `user2`.
8. The LTPA CallbackHandler implementation extracts the LTPA token from the current JAAS Subject in the security context and Web Services Security run time inserts the token as <wsse: BinarySecurityToken> in the SOAP header.
9. The Web Services Security run time in Web Services 3 calls the JAAS login configuration to validate the LTPA token and set it in the current security context as the JAAS Subject.

10. Web Services 3 is configured to send LTPA security to web services 4. In this case, assume that the RunAs role is not configured for Web Services 3. The LTPA token of `user2` is propagated to Web Services 4.
11. Client 2 uses the <wsse:UsernameToken> element to propagate the basic authentication data to Web Services 4.

Web Services Security complements the WebSphere Application Server security run time and the Java EE role-based security. This example demonstrates how to propagate security tokens across multiple resources such as web services and EJB files.

Web Services Security constraints:

The Web Services Security model that is used by WebSphere Application Server is the declarative model. A version 5.*x* application must be secured with Web Services Security by defining the security constraints in the IBM extension deployment descriptors and in IBM extension bindings.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

No Application Programming Interfaces (APIs) exist in WebSphere Application Server for programmatically interacting with Web Services Security. However, Service Provider Programming Interfaces (SPIs) are available for extending some security runtime behaviors. You can secure an application with Web Services Security by defining security constraints in the IBM extension deployment descriptors and in IBM extension bindings.

The development life cycle of a Web Services Security-enabled application is similar to the Java Platform, Enterprise Edition (Java EE) programming model. See the following figure for more details.

Figure 13. Application development life cycle

The Web Services Security constraints are defined by the assembler during the application assembly phase if the Java EE application is web services-enabled. Create, define, and edit the Web Services Security constraints with an assembly tool. For more information, read about assembly tools.

Web Services Security constraints

The security constraints for Web Services Security are specified in the IBM deployment descriptor extension for web services. The assembler defines these constraints during the application assembly phase, if the Java EE application is web services-enabled. Define the Web Services Security constraints using an assembly tool. See more information about assembling applications.

The Web Services Security run time acts on the constraints to enforce Web Services Security for the SOAP message. The scope of the IBM deployment descriptor extension is at the Enterprise JavaBeans (EJB) module or web module level. There also are bindings associated with each of the following IBM deployment descriptor extensions:

Client (might be either a Java EE client (application client container) or web services acting as a client)
- `ibm-webservicesclient-ext.xmi`
- `ibm-webservicesclient-bnd.xmi`

Server

- ibm-webservices-ext.xmi
- ibm-webservices-bnd.xmi

The IBM extension deployment descriptor and bindings are associated with each EJB module or web module. See Figure 2 for more information. If web services is acting as a client, then it contains the client IBM extension deployment descriptors and bindings in the EJB module or web module.

Figure 14. IBM extension deployment descriptors and bindings

The Web Services Security handler acts on the security constraints defined in the IBM extension deployment descriptor and enforces the security constraints accordingly. There are outbound and inbound configurations in both the client and server security constraints.

In a SOAP request, the following message points exist:
- Sender outbound
- Receiver inbound
- Receiver outbound
- Sender inbound

These message points correspond to the following four security constraints:
- Request sender (sender outbound)
- Request receiver (receiver inbound)
- Response sender (receiver outbound)
- Response receiver (sender inbound)

The security constraints of request sender and request receiver must match. Also, the security constraints of the response sender and response receiver must match. For example, if you specify integrity as a constraint in the request receiver, then you must configure the request sender to have integrity applied to the SOAP message. Otherwise, the request is denied because the SOAP message does not include the integrity specified in the request constraint.

The four security constraints are shown in the following figure of Web Services Security constraints.

Figure 15. Web Services Security constraints

Example: Sample configuration for Web Services Security for a version 5.x application:

To secure a version 5.*x* application with Web Services Security, you must define the security constraints in the IBM extension deployment descriptors and in IBM extension bindings. Sample keystore files and default binding information are provided for a sample configuration to demonstrate what IBM deployment descriptor extensions and bindings can do.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.x and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides the following sample keystores for sample configurations. These sample keystores are for testing and sample purposes only. Do not use them a in production environment.

- ${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks
  - The keystore password is client
  - Trusted certificate with alias name, soapca
  - Personal certificate with alias name, soaprequester and key password client issued by intermediary certificate authority Int CA2, which is, in turn, issued by soapca
- ${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
  - The keystore password is server
  - Trusted certificate with alias name, soapca
  - Personal certificate with alias name, soapprovider and key password server, issued by intermediary certificate authority Int CA2, which is, in turn, issued by soapca
- ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks
  - The keystore password is storepass
  - Secret key CN=Group1, alias name Group1, and key password keypass
  - Public key CN=Bob, O=IBM, C=US, alias name bob, and key password keypass
  - Private key CN=Alice, O=IBM, C=US, alias name alice, and key password keypass
- ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks
  - The keystore password is storepass
  - Secret key CN=Group1, alias name Group1, and key password keypass
  - Private key CN=Bob, O=IBM, C=US, alias name bob, and key password keypass
  - Public key CN=Alice, O=IBM, C=US, alias name alice, and key password keypass
- ${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer
  - The intermediary certificate authority is Int CA2.

Default binding (cell and server level)

WebSphere Application Server provides the following default binding information:

Trust anchors
Used to validate the trust of the signer certificate.

- SampleClientTrustAnchor is used by the response receiver to validate the signer certificate.
- SampleServerTrustAnchor is used by the request receiver to validate the signer certificate.

Collection Certificate Store
Used to validate the certificate path.

- SampleCollectionCertStore is used by the response receiver and the request receiver to validate the signer certificate path.

Key Locators
Used to locate the key for signature, encryption, and decryption.

- SampleClientSignerKey is used by the requesting sender to sign the SOAP message. The signing key name is clientsignerkey, which can be referenced in the signing information as the signing key name.
- SampleServerSignerKey is used by the responding sender to sign the SOAP message. The signing key name is serversignerkey, which can be referenced in the signing information as the signing key name.
- SampleSenderEncryptionKeyLocator is used by the sender to encrypt the SOAP message. It is configured to use the ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks keystore and the com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator keystore key locator.
- SampleReceiverEncryptionKeyLocator is used by the receiver to decrypt the encrypted SOAP message. The implementation is configured to use the ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks keystore and the

com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator keystore key locator. The implementation is configured for symmetric encryption (DES or TRIPLEDES). However, to use it for asymmetric encryption (RSA), you must add the private key CN=Bob, O=IBM, C=US, alias name bob, and key password keypass.

- SampleResponseSenderEncryptionKeyLocator is used by the response sender to encrypt the SOAP response message. It is configured to use the ${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks keystore and the com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator key locator. This key locator maps an authenticated identity (of the current thread) to a public key for encryption. By default, WebSphere Application Server is configured to map to public key alice, and you must change WebSphere Application Server to the appropriate user. The SampleResponseSenderEncryptionKeyLocator key locator also can set a default key for encryption. By default, this key locator is configured to use public key alice.

Trusted ID Evaluator

Used to establish trust before asserting to the identity in identity assertion.

SampleTrustedIDEvaluator is configured to use the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl implementation. The default implementation of com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator contains a list of trusted identities. The list is defined as properties with trustedId_* as the key and the value as the trusted identity. Define this information for the server level in the administration console by completing the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server1***.
2. Under Additional Properties, click **JAX-WS and JAX-RPC security runtime** > **Trusted ID Evaluators** > ***SampleTrustedIDEvaluator***.

For the cell level, click **Security** > **Web Services** > **Trusted ID Evaluators** > ***SampleTrustedIDEvaluator*** .

Login Mapping

Used to authenticate the incoming security token in the Web Services Security SOAP header of a SOAP message.

- The BasicAuth authentication method is used to authenticate user name security token (user name and password).
- The signature authentication method is used to map a distinguished name (DN) into a WebSphere Application Server Java Authentication and Authorization Server (JAAS) Subject.
- The IDAssertion authentication method is used to map a trusted identity into a WebSphere Application Server JAAS Subject for identity assertion.
- The Lightweight Third Party Authentication (LTPA) authentication method is used to validate a LTPA security token.

The previous default bindings for trust anchors, collection certificate stores, and key locators are for testing or sample purpose only. Do not use them for production.

A sample configuration

The following examples demonstrate what IBM deployment descriptor extensions and bindings can do. The unnecessary information was removed from the examples to improve clarity. Do not copy and paste these examples into your application deployment descriptors or bindings. These examples serve as reference only and are not representative of the recommended configuration.

Use the following tools to create or edit IBM deployment descriptor extensions and bindings:

- Use an assembly tool to create or edit the IBM deployment descriptor extensions.
- Use an assembly tool or the administrative console to create or edit the bindings file.

The following example illustrates a scenario that:

- Signs the SOAP body, time stamp, and security token.
- Encrypts the body content and user name token.
- Sends the user name token (basic authentication data).
- Generates the time stamp for the request.

For the response, the SOAP body and time stamp are signed, the body content is encrypted, and the SOAP message freshness is checked using the time stamp. The freshness of the message indicates whether the message complies with predefined time constraints.

The request sender and the request receiver are a pair. Similarly, the response sender and the response receiver are a pair.

Tip: It is recommended that you use the WebSphere Application Server variables for specifying the path to the key stores. In the administrative console, click **Environment** > **Manage WebSphere Variables**. These variables often help with platform differences such as file system naming conventions. In the following examples, `$${USER_INSTALL_ROOT}` is used for specifying the path to the key stores.

Client-side IBM deployment descriptor extension

The client-side IBM deployment descriptor extension describes the following constraints:

Request Sender

- Signs the SOAP body, time stamp and security token
- Encrypts the body content and user name token
- Sends the basic authentication token (user name and password)
- Generates the time stamp to expire in three minutes

Response Receiver

- Verifies that the SOAP body and time stamp are signed
- Verifies that the SOAP body content is encrypted
- Verifies that the time stamp is present (also check for message freshness)

The `xmi:id` statements are removed for readability. These statements must be added for this example to work.

Important: In the following code sample, lines 2 through 4 were split into three lines due to the width of the printed page.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscext:WsClientExtension xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wscext=
  http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscext.xmi">
  <serviceRefs serviceRefLink="service/myServ">
    <portQnameBindings portQnameLocalNameLink="Port1">
      <clientServiceConfig actorURI="myActorURI">
        <securityRequestSenderServiceConfig actor="myActorURI">
          <integrity>
            <references part="body"/>
            <references part="timestamp"/>
            <references part="securitytoken"/>
          </integrity>
          <confidentiality>
            <confidentialParts part="bodycontent"/>
            <confidentialParts part="usernametoken"/>
          </confidentiality>
          <loginConfig authMethod="BasicAuth"/>
          <addCreatedTimeStamp flag="true" expires="PT3M"/>
        </securityRequestSenderServiceConfig>
```

```
        <securityResponseReceiverServiceConfig>
          <requiredIntegrity>
            <references part="body"/>
            <references part="timestamp"/>
          </requiredIntegrity>
          <requiredConfidentiality>
            <confidentialParts part="bodycontent"/>
          </requiredConfidentiality>
          <addReceivedTimeStamp flag="true"/>
        </securityResponseReceiverServiceConfig>
      </clientServiceConfig>
    </portQnameBindings>
  </serviceRefs>
</com.ibm.etools.webservice.wscext:WsClientExtension>
```

Client-side IBM extension bindings

Example 2 shows the client-side IBM extension binding for the security constraints described previously in the discussion on client-side IBM deployment descriptor extensions.

The signer key and encryption (decryption) key for the message can be obtained from the keystore key locator implementation (`com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator`). The signer key is used for encrypting the response. The sample is configured to use the Java Certification Path API to validate the certificate path of the signer of the digital signature. The user name token (basic authentication) data is collected from the standard in (stdin) prompts using one of the default JAAS implementations `:javax.security.auth.callback.CallbackHandler` implementation (`com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`).

Important: In the following code sample, several lines were split into multiple lines due to the width of the printed page. See the close bracket for an indication of where each line of code ends.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscbnd:ClientBinding xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wscbnd=
  "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscbnd.xmi">
  <serviceRefs serviceRefLink="service/MyServ">
    <portQnameBindings portQnameLocalNameLink="Port1">
      <securityRequestSenderBindingConfig>
        <signingInfo>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <signingKey name="clientsignerkey" locatorRef="SampleClientSignerKey"/>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfo>
        <keyLocators name="SampleClientSignerKey" classname=
        "com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}PDM2OjEr" path=
          "${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks" type="JKS"/>
          <keys alias="soaprequester" keypass="{xor}PDM2OjEr" name="clientsignerkey"/>
        </keyLocators>
        <encryptionInfo name="EncInfo1">
          <encryptionKey name="CN=Bob, O=IBM, C=US" locatorRef=
          "SampleSenderEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfo>
        <keyLocators name="SampleSenderEncryptionKeyLocator" classname=
        "com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}LCswLTovPiws" path=
          "${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks" type="JCEKS"/>
          <keys alias="Group1" keypass="{xor}NDomLz4sLA==" name="CN=Group1"/>
        </keyLocators>
        <loginBinding authMethod="BasicAuth" callbackHandler=
        "com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler"/>
      </securityRequestSenderBindingConfig>
      <securityResponseReceiverBindingConfig>
        <signingInfos>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <certPathSettings>
            <trustAnchorRef ref="SampleClientTrustAnchor"/>
            <certStoreRef ref="SampleCollectionCertStore"/>
          </certPathSettings>
```

```
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfos>
        <trustAnchors name="SampleClientTrustAnchor">
          <keyStore storepass="{xor}PDM2OjEr" path=
          "$${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks" type="JKS"/>
        </trustAnchors>
        <certStoreList>
          <collectionCertStores provider="IBMCertPath" name="SampleCollectionCertStore">
            <x509Certificates path="$${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer"/>
          </collectionCertStores>
        </certStoreList>
        <encryptionInfos name="EncInfo2">
          <encryptionKey locatorRef="SampleReceiverEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfos>
        <keyLocators name="SampleReceiverEncryptionKeyLocator" classname=
        "com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}PDM2OjEr" path=
          "$${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks" type="JKS"/>
          <keys alias="soaprequester" keypass="{xor}PDM2OjEr" name="clientsignerkey"/>
        </keyLocators>
      </securityResponseReceiverBindingConfig>
    </portQnameBindings>
  </serviceRefs>
</com.ibm.etools.webservice.wscbnd:ClientBinding>
```

Server-side IBM deployment descriptor extension

The client-side IBM deployment descriptor extension describes the following constraints:

Request Receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi`)

- Verifies that the SOAP body, time stamp, and security token are signed.
- Verifies that the SOAP body content and user name token are encrypted.
- Verifies that the basic authentication token (user name and password) is in the Web Services Security SOAP header.
- Verifies that the time stamp is present (also check for message freshness). The freshness of the message indicates whether the message complies with predefined time constraints.

Response Sender (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi`)

- Signs the SOAP body and time stamp
- Encrypts the SOAP body content
- Generates the time stamp to expire in 3 minutes

Important: In the following code sample, several lines were split into multiple lines due to the width of the printed page. See the close bracket for an indication of where each line of code ends.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wsext=
http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsext.xmi">
  <wsDescExt wsDescNameLink="MyServ">
    <pcBinding pcNameLink="Port1">
      <serverServiceConfig actorURI="myActorURI">
        <securityRequestReceiverServiceConfig>
          <requiredIntegrity>
            <references part="body"/>
            <references part="timestamp"/>
            <references part="securitytoken"/>
          </requiredIntegrity>
          <requiredConfidentiality">
            <confidentialParts part="bodycontent"/>
            <confidentialParts part="usernametoken"/>
          </requiredConfidentiality>
          <loginConfig>
            <authMethods text="BasicAuth"/>
          </loginConfig>
```

```
      <addReceivedTimestamp flag="true"/>
    </securityRequestReceiverServiceConfig>
    <securityResponseSenderServiceConfig actor="myActorURI">
      <integrity>
        <references part="body"/>
        <references part="timestamp"/>
      </integrity>
      <confidentiality>
        <confidentialParts part="bodycontent"/>
      </confidentiality>
      <addCreatedTimestamp flag="true" expires="PT3M"/>
    </securityResponseSenderServiceConfig>
  </serverServiceConfig>
 </pcBinding>
</wsDescExt>
</com.ibm.etools.webservice.wsext:WsExtension>
```

Server-side IBM extension bindings

The following binding information reuses some of the default binding information defined either at the server level or the cell level, which depends upon the installation. For example, request receiver is referencing the `SampleCollectionCertStore` certification store and the `SampleServerTrustAnchor` trust store is defined in the default binding. However, the encryption information in the request receiver is referencing a `SampleReceiverEncryptionKeyLocator` key locator defined in the application-level binding (the same `ibm-webservices-bnd.xmi` file). The response sender is configured to use the signer key of the digital signature of the request to encrypt the response using one of the default key locator (`com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator`) implementations.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wsbnd=
  http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
  <wsdescBindings wsDescNameLink="MyServ">
    <pcBindings pcNameLink="Port1" scope="Session">
      <securityRequestReceiverBindingConfig>
        <signingInfos>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <certPathSettings>
            <trustAnchorRef ref="SampleServerTrustAnchor"/>
            <certStoreRef ref="SampleCollectionCertStore"/>
          </certPathSettings>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfos>
        <encryptionInfos name="EncInfo1">
          <encryptionKey locatorRef="SampleReceiverEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfos>
        <keyLocators name="SampleReceiverEncryptionKeyLocator" classname=
         "com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}LCswLTovPiws" path="$${USER_INSTALL_ROOT}/
           etc/ws-security/samples/enc-receiver.jceks" type="JCEKS"/>
          <keys alias="Group1" keypass="{xor}NDomLz4sLA==" name="CN=Group1"/>
          <keys alias="bob" keypass="{xor}NDomLz4sLA==" name="CN=Bob, O=IBM, C=US"/>
        </keyLocators>
      </securityRequestReceiverBindingConfig>
      <securityResponseSenderBindingConfig>
        <signingInfo>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <signingKey name="serversignerkey" locatorRef="SampleServerSignerKey"/>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfo>
        <encryptionInfo  name="EncInfo2">
          <encryptionKey locatorRef="SignerKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfo>
        <keyLocators name="SignerKeyLocator" classname=
         "com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator"/>
      </securityResponseSenderBindingConfig>
```

```
    </pcBindings>
  </wsdescBindings>
  <routerModules transport="http" name="StockQuote.war"/>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

Overview of authentication methods:

The Web Services Security implementation for WebSphere Application Server supports the following authentication methods: BasicAuth, Lightweight Third Party Authentication (LTPA), digital signature, and identity assertion.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

When the WebSphere Application Server is configured to use the BasicAuth authentication method, the sender attaches the Lightweight Third Party Authentication (LTPA) token as a BinarySecurityToken from the current security context or from basic authentication data configuration in the binding file in the SOAP message header. The Web Services Security message receiver authenticates the sender by validating the user name and password against the configured user registry. With the LTPA method, the sender attaches the LTPA BinarySecurityToken it previously received in the SOAP message header. The receiver authenticates the sender by validating the LTPA token and the token expiration time. With the Digital Signature authentication method, the sender attaches a BinarySecurityToken from a X509 certificate to the Web Services Security message header along with a digital signature of the message body, time stamp, security token, or any combination of the three. The receiver authenticates the sender by verifying the validity of the X.509 certificate and the digital signature using the public key from the verified certificate.

The identity assertion authentication method is different from the other three authentication methods. This method establishes the security credential of the sender based on the trust relationship. You can use the identity assertion authentication method, for example, when an intermediary server must invoke a service from a downstream server on behalf of the client, but does not have the client authentication information. The intermediary server might establish a trust relationship with the downstream server and then assert the client identity to the same downstream server.

Web Services Security supports the following trust modes:

- BasicAuth
- Digital signature
- Presumed trust

When you use the BasicAuth and digital signature trust modes, the intermediary server passes its own authentication information to the downstream server for authentication. The presumed trust mode establishes a trust relationship using some external mechanism. For example, the intermediary server might pass SOAP messages through a Secure Socket Layers (SSL) connection with the downstream server and transport layer client certificate authentication.

The Web Services Security implementation for WebSphere Application Server validates the trust relationship by following this procedure:

1. The downstream server validates the authentication information of the intermediary server.
2. The downstream server verifies whether the authenticated intermediary server is authorized for identity assertion. For example, the intermediary server must be in the trust list for the downstream server.

The client identity might be represented by a name string, a distinguished name (DN), or an X.509 certificate. The client identity is attached in the Web Services Security message in a UsernameToken with just a user name, DN, or in a BinarySecurityToken of a certificate. The following table summarizes the type of security token that is required for each authentication method.

Table 38. Authentication methods and their security tokens. Use the methods to authenticate the sender of a message.

Authentication method	Security token
BasicAuth	BasicAuth requires `<wsse:UsernameToken>` with `<wsse:Username>` and `<wsse:Password>`.
Signature	Signature requires `<ds:Signature>` and `<wsse:BinarySecurityToken>`.
IDAssertion	IDAssertion requires `<wsse:UsernameToken>` with `<wsse:Username>` or `<wsse:BinarySecurityToken>` with a X.509 certificate for client identity depending on `<idType>`. This method also requires other security tokens according to the `<trustMode>`: • If the `<trustMode>` is BasicAuth, IDAssertion requires `<wsse:UsernameToken>` with `<wsse:Username>` and `<wsse:Password>`. • If the `<trustMode>` is Signature, IDAssertion requires `<wsse:BinarySecurityToken>`.
LTPA	LTPA requires `<wsse:BinarySecurityToken>` with an LTPA token.

A web service can support multiple authentication methods simultaneously. The receiver side of the web services deployment descriptor can specify all the authentication methods that are supported in the `ibm-webservices-ext.xmi` XML file. The web services receiver-side, as shown in the following example, is configured to accept all the authentication methods described previously:

```
<loginConfig xmi:id="LoginConfig_1052760331326">
      <authMethods xmi:id="AuthMethod_1052760331326" text="BasicAuth"/>
      <authMethods xmi:id="AuthMethod_1052760331327" text="IDAssertion"/>
      <authMethods xmi:id="AuthMethod_1052760331336" text="Signature"/>
      <authMethods xmi:id="AuthMethod_1052760331337" text="LTPA"/>
</loginConfig>
<idAssertion xmi:id="IDAssertion_1052760331336" idType="Username" trustMode="Signature"/>
```

You can define only one authentication method in the sender-side web services deployment descriptor. A web service client can use any of the authentication methods that are supported by the particular web services application. The following example illustrates an identity assertion authentication method configuration in the `ibm-webservicesclient-ext.xmi` deployment descriptor extension of the web service client:

```
<loginConfig xmi:id="LoginConfig_1051555852697">
      <authMethods xmi:id="AuthMethod_1051555852698" text="IDAssertion"/>
</loginConfig>
<idAssertion xmi:id="IDAssertion_1051555852697" idType="Username" trustMode="Signature"/>
```

As shown in the previous example, the client identity type is `Username` and the trust mode is digital signature.

Figure 16. Security token generation and validation

The sender security handler invokes the `handle()` method of an implementation of the `javax.security.auth.callback.CallbackHandler` interface. The `javax.security.auth.callback.CallbackHandler` interface creates the security token and passes it back to the sender security handler. The sender security handler constructs the security token based on the authentication information in the callback array and inserts the security token into the Web Services Security message header.

The receiver security handler compares the token type in the message header with the expected token types configured in the deployment descriptor. If none of the expected token types are found in the Web Services Security header of the SOAP message, the request is rejected with a SOAP fault exception. Otherwise, the token type is used to map to a Java Authentication and Authorization Service (JAAS) login configuration for validating the token. If the authentication is successful, a JAAS Subject is created and associated with the running thread. Otherwise, the request is rejected with a SOAP fault exception.

Overview of token types:

Web Services Security defines the types of security tokens. The deployment descriptor extension file defines the types of tokens that the message can accept.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.1*x* and later applications.

The types of security tokens that are defined by Web Services Security are:

- User name token
- Binary security token

A user name token consists of a user name and, optionally, password information. You can include a user name token directly in the <Security> header within the message. Binary tokens, such as X.509 certificates, Kerberos tickets, Lightweight Third Party Authentication (LTPA) tokens, or other non-XML formats, require a special encoding for inclusion. The Web Services Security specification describes how to encode binary security tokens such as X.509 certificates and Kerberos tickets, and it also describes how to include opaque encrypted keys. The specification also includes extensibility mechanisms that you can use to further describe the characteristics of the credentials that are included with a message.

WebSphere Application Server Version 5.0.2 supports user name tokens, which include both user name and password for basic authentication and user name, which is used for identity assertion. The WebSphere Application Server Version 5.0.2 binary security token implementation supports both X.509 certificates and LTPA binary security. You extend the implementation to generate other types of tokens. However, Kerberos tickets are not supported in WebSphere Application Server Version 5.0.2. Each type of token is processed by a corresponding token generation and validation module. The binary token generation and validation modules are pluggable that is based on the Java Authentication and Authorization Service (JAAS) framework. For example, an arbitrary XML-based token format is supported using the JAAS pluggable framework. WebSphere Application Server Version 5.0.2 does not support an XML-based token that is used in the SecurityTokenReference.

You can define the types of tokens that the message can accept in the deployment descriptor extension file, `ibm.webservices-ext.xmi`. A message receiver might support one or more types of security tokens. The following example shows that the receiver supports four types of security tokens:

Important: In the following code sample, several lines were split into multiple lines due to the width of the printed page. See the close bracket for an indication of where each line of code ends.

```
?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.etools.webservice.wsext=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsext.xmi"
xmi:id="WsExtension_1052760331306" routerModuleName="StockQuote.war">
  <wsDescExt xmi:id="WsDescExt_1052760331306" wsDescNameLink="StockQuoteFetcher">
    <pcBinding xmi:id="PcBinding_1052760331326" pcNameLink="urn:xmltoday-delayed-quotes"
      scope="Session">
       <serverServiceConfig
         xmi:id="ServerServiceConfig_1052760331326"actorURI="myActorURI">
          <securityRequestReceiverServiceConfig
           xmi:id="SecurityRequestReceiverServiceConfig_1052760331326">
            <loginConfig xmi:id="LoginConfig_1052760331326">
              <authMethods xmi:id="AuthMethod_1052760331326" text="BasicAuth"/>
              <authMethods xmi:id="AuthMethod_1052760331327" text="IDAssertion"/>
              <authMethods xmi:id="AuthMethod_1052760331336" text="Signature"/>
              <authMethods xmi:id="AuthMethod_1052760331337" text="LTPA"/>
           </loginConfig>
<idAssertion xmi:id="IDAssertion_1052760331336" idType="Username" trustMode="Signature"/>
```

The message sender might choose one of the token types that are supported by the receiver when sending a message. You can define the type of token to be used by the sending side in the client descriptor extension file, `ibm-webservicesclient-ext.xmi`. The following example shows that the sender chooses to send a UsernameToken to the receiver:

Important: In the following code sample, several lines were split into multiple lines due to the width of the printed page. See the close bracket for an indication of where each line of code ends.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscext:WsClientExtension xmi:version="2.0"
mlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.etools.webservice.wscext=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscext.xmi"
xmi:id="WsClientExtension_1052760331496">
<ServiceRefs xmi:id="ServiceRef_1052760331506" serviceRefLink="service/StockQuoteService">
    <portQnameBindings xmi:id="PortQnameBinding_1052760331506"
portQnameLocalNameLink="StockQuote">
      <clientServiceConfig xmi:id="ClientServiceConfig_1052760331506"
actorURI="myActorURI">
          <securityRequestSenderServiceConfig
xmi:id="SecurityRequestSenderServiceConfig_1052760331506" actor="myActorURI">
          <loginConfig xmi:id="LoginConfig_1052760331506" authMethod="BasicAuth"/>
```

Username token:

The <UsernameToken> element propagates a user name and optionally propagates the password information. Use this token type to carry basic authentication information.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Both a user name and a password are used to authenticate the message. A <UsernameToken> element that contains the user name is used in identity assertion. Identity assertion establishes the identity of the user, based on the trust relationship.

The following example shows the syntax of the <UsernameToken> element:

```
<UsernameToken Id="...">
    <Username>...</Username>
    <Password Type="...">...</Password>
</UsernameToken>
```

The Web Services Security specification defines the following password types:

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText (default)
This type is the actual password for the user name.

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest
The type is the digest of the password for the user name. The value is a base64-encoded SHA1 hash value of the UTF8-encoded password.

WebSphere Application Server supports the default PasswordText type. However, it does not support password digest because most user registry security policies do not expose the password to the application software.

The following example illustrates the use of the <UsernameToken> element:

```
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
    <S:Header>
            ...
        <wsse:Security>
            <wsse:UsernameToken>
```

```
            <wsse:Username>Joe</wsse:Username>
            <wsse:Password>ILoveJava</wsse:Password>
         </wsse:UsernameToken>
      </wsse:Security>
   </S:Header>
</S:Envelope>
```

Nonce, a randomly generated token:

Nonce is a randomly generated, cryptographic token used to prevent the theft of user name tokens used with SOAP messages. Nonce is used with the basic authentication (BasicAuth) method.

Without nonce, when a UsernameToken is passed from one machine to another machine using a nonsecure transport, such as HTTP, the token might be intercepted and used in a replay attack. The same key might be reused when the username token is transmitted between the client and the server, which leaves it vulnerable to attack. The user name token can be stolen even if you use XML digital signature and XML encryption.

To help eliminate these replay attacks, the <wsse:Nonce> and <wsu:Created> elements are generated within the <wsse: usernameToken> element and used to validate the message. The request receiver or response receiver checks the freshness of the message to verify the difference between when the message is created and the current time falls within a specified time period. Also, WebSphere Application Server verifies that the receiver has not processed the token within the specified time period. These two features are used to lessen the chance that a user name token is used for a replay attack.

Binary security token:

The <ValueType> attribute identifies the type of the security token, for example, a Lightweight Third Party Authentication (LTPA) token. The EncodingType indicates how the security token is encoded, for example, Base64Binary. The <BinarySecurityToken> element defines a security token that is binary encoded. The encoding is specified using the EncodingType attribute. The value type and space are specified using the ValueType attribute. The Web Services Security implementation for WebSphere Application Server, Version 5.0.2 supports both LTPA and X.509 certificate binary security tokens.

Important: There is an important distinction between Version 5.x and Version 6 and later applications. The information in this article supports Version 5.x applications only that are used with WebSphere Application Server Version 6.0.x and later. The information does not apply to Version 6 and later applications.

A binary security token has the following attributes that are used for interpretation:

- Value type
- Encoding type

The following example depicts an LTPA binary security token in a Web Services Security message header:

```
<wsse:BinarySecurityToken xmlns:ns7902342339871340177=
                    "http://www.ibm.com/websphere/appserver/tokentype/5.0.2"
     EncodingType="wsse:Base64Binary"
     ValueType="ns7902342339871340177:LTPA">
            MIZ6LGPt2CzXBQfio9wZTo1VotWov0NW3Za6lU5K7Li78DSnIK6iHj3hxXgrUn6p4wZI
            8Xg26havepvmSJ8XxiACMihTJuh1t3ufsrjbFQJOqh5VcRvI+AKEaNmnEgEV65jUYAC9
            C/iwBBWk5U/6DIk7LfXcTT0ZPAd+3D3nCS0f+6tnqMou8EG9mtMeTKccz/pJVTZjaRSo
            msu0sewsOKfl/WPsjW0bR/2g3NaVvBy18VlTFBpUbGFVGgzHRjBKAGo+ctkl80nlVLIk
            TUjt/XdYvEpOr6QoddGi4okjDGPyyoDxcvKZnReXww5UsoqlpfXwN4KG9as=
</wsse:BinarySecurityToken></wsse:Security></soapenv:Header>
```

As shown in the example, the token is Base64Binary encoded.

XML token:

XML tokens are offered in two formats, Security Assertion Markup Language (SAML) and Extensible rights Markup Language (XrML).

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

XML-based security tokens are growing in popularity. Two well-known formats are:

- Security Assertion Markup Language (SAML)
- Extensible rights Markup Language (XrML)

Using extensibility of the <wsse:Security> header in XML-based security tokens, you can directly insert these security tokens into the header.

SAML assertions are attached to Web Services Security messages using web services by placing assertion elements inside the <wsse:Security> header. The following example illustrates a Web Services Security message with a SAML assertion token.

```
<S:Envelope xmlns:S="...">&
      <wsse:Security xmlns:wsse="...">
          <saml:Assertion
                    MajorVersion="1"
                    MinorVersion="0"
                    AssertionID="SecurityToken-ef375268"
                       Issuer="elliotw1"
                       IssueInstant="2002-07-23T11:32:05.6228146-07:00"
                     xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
                     ...
           </saml:Assertion>
      </wsse:Security>
 </S:Header>
 <S:Body>
 ...
 </S:Body>
</S:Envelope>
```

For a complete list of the supported standards and specifications, read about web services specifications and APIs.

XML digital signature:

XML-Signature Syntax and Processing (XML signature) is a specification that defines XML syntax and processing rules to sign and verify digital signatures for digital content. The specification was developed jointly by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF).

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

XML signature does not introduce new cryptographic algorithms. WebSphere Application Server uses XML signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method.

XML canonicalization (c14n) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. For example, although their octet representations are different, the following examples are identical:

- `<person first="John" last="Smith"/>`
- `<person last="Smith" first="John"></person>`

C14n is a process used to canonicalize XML information. Select an appropriate c14n algorithm because the information that is canonicalized is dependent upon this algorithm. One of the major c14n algorithms,

Exclusive XML Canonicalization, canonicalizes the character encoding scheme, attribute order, namespace declarations, and so on. The algorithm does not canonicalize white space outside tags, namespace prefixes, or data type representation.

XML signature in the Web Services Security-Core specification

The Web Services Security-Core (WSS-Core) specification defines a standard way for SOAP messages to incorporate an XML signature. You can use almost all of the XML signature features in WSS-Core except enveloped signature and enveloping signature. However, WSS-Core has some recommendations such as exclusive canonicalization for the c14n algorithm and some additional features such as `SecurityTokenReference` and `KeyIdentifier`. The `KeyIdentifier` is the value of the SubjectKeyIdentifier field within the X.509 certificate. For more information on the `KeyIdentifier`, see "Reference to a Subject Key Identifier" within the OASIS Web Services Security X.509 Certificate Token Profile documentation.

By including XML signature in SOAP messages, the following are realized:

Message integrity
: A message receiver can confirm that attackers or accidents have not altered parts of the message after these parts are signed by a key.

Authentication
: You can assume that a valid signature is *proof of possession*. A message with a digital certificate issued by a certificate authority and a signature in the message that is validated successfully by a public key in the certificate, is proof that the signer has the corresponding private key. The receiver can authenticate the signer by checking the trustworthiness of the certificate.

XML signature in the current implementation

XML signature is supported in Web Services Security, however, an application programming interface (API) is not available. The current implementation has many hardcoded behaviors and has some user-operable configuration items. To configure the client for digital signature, see Configuring the client for response digital signature verification: Verifying the message parts. To configure the server for digital signature, see Configuring the server for request digital signature verification: Verifying the message parts.

Security considerations:

In a replay attack, an attacker taps the lines, receives a signed message, and then returns the message to the receiver. In this case, the receiver receives the same message twice and might process both of them if the signatures are valid. Processing both messages can cause damage to the receiver if the message is a claim for money. If you have the signed generation time stamp and the signed expiration time in a message replay, attacks might be reduced. However, this is not a complete solution. A message must have a nonce value to prevent these attacks and the receiver must reject a message that contains a processed nonce. The current implementation does not provide a standard way to generate and check nonces in messages. In WebSphere Application Server, Version 5.1, nonce is supported in username tokens only. The username token profile contains concrete nonce usage scenarios for username tokens. Applications handle nonces (such as serial numbers) and they need to be signed.

Signing parameter configuration settings:

Use this page to configure new signing parameters.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with

WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications. Version 5.*x* applications are based on Java 2 platform, Enterprise Edition (J2EE) 1.3.

The specifications that are listed on this page for the signature method, digest method, and canonicalization method are located in the World Wide Web Consortium (W3C) document entitled, *XML Signature Syntax and Specification: W3C Recommendation 12 Feb 2002*.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Additional properties, you can access the signing information for the following bindings:
   a. For the Request sender binding, click **Web services: Client security bindings**. Under Request sender binding, click **Edit**. Under Additional properties, click **Signing information**.
   b. For the Response sender binding, click **Web services: Server security bindings**. Under Response sender binding, click **Edit**. Under Additional properties, click **Signing information**.
4. In the Request Sender Binding column, click **Edit** > **Signing Information**.

If the signing information is not available, select **None**.

If the signing information is available, select **Dedicated Signing Information** and specify the configuration in the following fields:

Signature method:

Specifies the algorithm Uniform Resource Identifiers (URI) of the signature method.

The following algorithms are supported:
- http://www.w3.org/2000/09/xmldsig#rsa-sha1
- http://www.w3.org/2000/09/xmldsig#dsa-sha1
- http://www.w3.org/2000/09/xmldsig#hmac-sha1

`You can also add custom algorithms.`

Digest method:

Specifies the algorithm URI of the digest method.

WebSphere Application Server supports the http://www.w3.org/2000/09/xmldsig#sha1 algorithm.

Canonicalization method:

Specifies the algorithm URI of the canonicalization method.

The following algorithms are supported:
- http://www.w3.org/2001/10/xml-exc-c14n#
- http://www.w3.org/2001/10/xml-exc-c14n#WithComments
- http://www.w3.org/TR/2001/REC-xml-c14n-20010315
- http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments

Key name:

Specifies the name of the key object found in the keystore file.

Key locator reference:

Specifies the name used to reference the key locator

You can configure these key locator reference options on the cell level, the server level, and the application level. The configurations that are listed in the field are a combination of the configurations on these three levels.

You can specify a key locator configuration for the following bindings on the following levels:

Table 39. Key locator binding settings. The key locator is part of the signing parameter information.

Binding name	Cell level, server level, or application level	Path
N/A	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Key locators**.
N/A	Server level	1. Click **Servers** > **Server Typ** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Key locators**.
Request sender	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. 4. Under Request sender binding, click **Edit**. 5. Under Additional properties, click **Key locators**.
Request receiver	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage module** > ***URI_name***. 3. Click **Web services: Server security bindings**. 4. Under Request receiver binding, click **Edit**. 5. Under Additional properties, click **Key locators**.
Response sender	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage module** > ***URI_name***. 3. Click **Web services: Server security bindings**. 4. Under Response sender binding, click **Edit**. 5. Under Additional properties, click **Key locators**.
Response receiver	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. 4. Under Response receiver binding, click **Edit**. 5. Under Additional properties, click **Key locators**.

Default binding:

The default binding information is defined in the `ws-security.xml` file and can be administered by either the administrative console or by scripting. Only default bindings for JAX-RPC applications are supported. Default bindings for JAX-WS applications are not supported.

Important: There is an important distinction between Version 5.x and Version 6 and later applications. The information in this article supports version 5.x applications only that are used with WebSphere Application Server Version 6.0.x and later. The information does not apply to Version 6 and later applications. Also, policy sets can only be used with JAX-WS applications. Policy sets cannot be used for JAX-RPC applications.

Certain applications can share certain binding information. This information includes truststores, keystores, and authentication methods (token validation). WebSphere Application Server provides support for default binding information. Administrators can define binding information at:

- The server level
- The cell level

Applications can refer to this binding information.

You can define the following binding information in the `ws-security.xml` file:

Trust anchors (truststore)

- **`Trust anchors`** contain key store configuration information that has the root-trusted certificates. Trust anchors are used for certificate path validation of the incoming X.509-formatted security tokens.
- The Trust Anchor Name is used in the binding file (`ibm-webservices-bnd.xmi` and `ibm-webservicesclient-bnd-xmi` when web services is running as a client) to refer to the trust anchor defined in the default binding information. The trust anchor name must be unique in the trust anchor collection.

Collection certificate store

- The **`collection certificate store`** specifies a list of untrusted, intermediate certificates and is used for certificate path validation of incoming X.509-formatted security tokens. The default provider is IBMCertPath.
- The Certificate Store Name is used in the binding file (`ibm-webservices-bnd.xmi` and `ibm-webservicesclient-bnd-xmi` when web services is running as a client) to refer to the certificate store defined in the default binding information. The Certificate Store Name must be unique to the collection certificate store collection.

Key locators

- **`Key locators`** specify implementation of the com.ibm.wsspi.wssecurity.config.KeyLocator interface. This interface is used to retrieve keys for signature or encryption. Customer implementations can extend the key locator interface to retrieve keys using other methods. WebSphere Application Server provides implementations to retrieve a key from the key store, map an authenticated identity to a key in the key store, or retrieve a key from the signer certificate (mapping and retrieving actions are used for encrypting the response).
- The Key Locator Name is used in the binding file (`ibm-webservices-bnd.xmi` and `ibm-webservicesclient-bnd-xmi` when web services is running as a client) to refer to the key locator defined in the default binding information. The Key Locator Name must be unique to the key locators collection in the default binding information.

Trusted ID evaluators

- **`Trusted ID evaluators`** are an implementation of the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface. This interface is used to make sure the identity (ID)-asserting authority is trusted. Additionally, you can extend the trusted identity evaluator to validate the trust. WebSphere Application Server provides a default implementation for validating trust based on a predefined list of identities.
- The Trusted ID Evaluator Name is used in the binding file (`ibm-webservices-bnd.xmi`) to refer to the trusted identity evaluator defined in the default binding information. The Trusted ID Evaluator Name must be unique to the Trusted ID Evaluator collection.

Login mappings

- `Login mappings` define the mapping of the authentication method to the Java Authentication and Authorization Service (JAAS) login configuration. The mappings are used to authenticate the incoming security token embedded in the Web Services Security SOAP message header. The JAAS login configuration is defined in the administrative console under **Security** > **Global security** > **Java Authentication and Authorization Service** > **Application logins**.
- WebSphere Application Server defines the following authentication methods:

  BasicAuth
  Authenticates user name and password.

  Signature
  Maps the subject distinguished name (DN) in the certificate to a WebSphere Application Server credential.

  IDAssertion
  Maps the identity to a WebSphere Application Server credential.

  LTPA Authenticates a Lightweight Third Party Authentication (LTPA) token.

  After identity authentication, the associated credential is used in the downstream call.
- This method can be extended to authenticate custom security tokens by providing a custom JAAS login configuration and by using the com.ibm.wsspi.wssecurity.auth.module.WSSecurityMappingModule to create the principal and credential required by WebSphere Application Server.
- If LoginConfig (`AuthMethod`) is defined in the IBM extension deployment descriptor (`ibm-webservices-ext.xmi`), but there are no login mapping bindings (`ibm-webservices-bnd.xmi`) defined for the `AuthMethod`, Web Services Security run time uses the login mapping defined in the default binding information.

WebSphere Application Server, Network Deployment

When the WebSphere Application Server, Network Deployment cell, the default binding file (`ws-security.xml`) of the server is added to the new cell (with other server level configuration information). If you use the cell-level default binding, the entries of the server level default binding must be removed.

There is a cell-level default binding (`ws-security.xml`) for WebSphere Application Server, Network Deployment installation. Furthermore, for WebSphere Application Server, Network Deployment installation server-level binding is optional. To navigate to the cell-level default binding in the administrative console, click **Security > Web Services**.

Figure 17. Web Services Security application-level, cell-level, and server-level default binding information

The order of the default binding information is application-level binding, server-level, and cell-level default binding.

ws-security.xml file - Default configuration for WebSphere Application Server, Network Deployment:

For JAX-RPC applications, WebSphere Application Server, Network Deployment installation uses the `ws-security.xml` file to define the default binding information for Web Services Security for an entire cell.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.x and later. The information does not apply to Version 6.0.*x* and later applications.

In the WebSphere Application Server, Network Deployment installation, the `ws-security.xml` file is at the cell level and defines the default binding information for Web Services Security for the entire cell. But each application server can have its own `ws-security.xml` file to override the cell default; similarly, each web service can override the default in its binding files. The following list contains the defaults defined in `ws-security.xml` file:

Trust anchors
Identifies the trusted root certificates for signature verification.

Collection certificate stores
Contains certificate revocation lists (CRLs) and non-trusted certificates for verification.

Key locators
Locates the keys for digital signature and encryption.

Trusted ID evaluators
Evaluates the trust of the received identity before identity assertion.

Login mappings
Contains the Java Authentication and Authorization Service (JAAS) configurations for AuthMethod token validation.

The Web Services Security run time reads the configuration from the application bindings first, then tries the server-level, and finally tries the cell level. The following figure depicts the runtime configuration process.

Figure 18. Runtime configuration

Trust anchors:

A trust anchor specifies keystores that contain trusted root certificates that validate the signer certificate. The request receiver and the response receiver use these keystores to validate the signer certificate of the digital signature.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

The request receiver (as defined in the `ibm-webservices-bnd.xmi` file) and the response receiver (as defined in the `ibm-webservicesclient-bnd.xmi` file when web services are acting as client) use these keystores to validate the signer certificate of the digital signature. The keystores are critical to the integrity of the digital signature validation. If the keysores are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these keystores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `ibm-webservicesclient-bnd.xmi` file.

The trust anchor is defined as javax.security.cert.TrustAnchor in the Java CertPath application programming interface (API). The Java CertPath API uses the trust anchor and the certificate store to validate the incoming X.509 certificate that is embedded in the SOAP message.

The Web Services Security implementation in WebSphere Application Server supports this trust anchor. In WebSphere Application Server, the trust anchor is represented as a Java keystore object. The type, path, and password of the keystore are passed to the implementation through the administrative console or by scripting.

Collection certificate store:

A *collection certificate store* is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs is used to check the signature of a digitally signed SOAP message.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* applications.

The collection certificate stores are used when processing a received SOAP message. This collection is configured in the securityRequestReceiverBindingConfig section of the binding file for servers and in the securityResponseReceiverBindingConfig section of the binding file for clients.

A collection certificate store is one kind of certificate store. A certificate store is defined as javax.security.cert.CertStore in the Java CertPath application programming interface (API). The Java CertPath API defines the following types of certificate stores:

Collection certificate store
A collection certificate store accepts the certificates and CRLs as Java collection objects.

Lightweight Directory Access Protocol certificate store
The Lightweight Directory Access Protocol (LDAP) certificate store accepts certificates and CRLs as LDAP entries.

The CertPath API uses the certificate store and the trust anchor to validate the incoming X.509 certificate that is embedded in the SOAP message.

The Web Services Security implementation in the WebSphere Application Server supports the collection certificate store. Each certificate and CRL is passed as an encoded file. This configuration is done using either the administrative console or by scripting.

Key locator:

A *key locator* (com.ibm.wsspi.wssecurity.config.KeyLocator) is an abstraction of the mechanism that retrieves the key for digital signature and encryption.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* applications.

You can use any of the following infrastructure from which to retrieve the keys depending upon the implementation:

- Java keystore file
- Database
- Lightweight Third Party Authentication (LTPA) server

Key locators search the key using some type of a clue. The following types of clues are supported:

- A string label of the key, which is explicitly passed through the application programming interface (API). The relationships between each key and its name (string label) is maintained inside the key locator.
- The execution context of the key locator; explicit information is not passed to the key locator. A key locator determines the appropriate key according to the execution context.

Current versions of key locators do not support the retrieval of verification keys because current Web Services Security implementations do not support the secret key-based signature. Because the key

locators support the public key-based signature only, the key for verification is embedded in the X.509 certificate as a `<BinarySecurityToken>` element in the incoming message.

For example, key locators can obtain the identity of the caller from the context and can retrieve the public key of the caller for response encryption.

Usage scenarios

This section describes the usage scenarios for key locators.

Signing:

The name of the signing key is specified in the Web Services Security configuration. This value is passed to the key locator and the actual key is returned. The corresponding X.509 certificate also can be returned.

Verification

As described previously, key locators are not used in signature verification.

Encryption:

The name of the encryption key is specified in the Web Services Security configuration. This value is passed to the key locator and the actual key is returned.

Decryption:

The Web Services Security specification recommends using the key identifier instead of the key name. However, while the algorithm for computing the identifier for the public keys is defined in Internet Engineering Task Force (IETF) Request for Comment (RFC) 3280, there is no agreed upon algorithm for the secret keys. Therefore, the current implementation of Web Services Security uses the identifier only when public key-based encryption is performed. Otherwise, the ordinal key name is used.

When you use public key-based encryption, the value of the key identifier is embedded in the incoming encrypted message. Then, the Web Services Security implementation searches for all the keys managed by the key locator and decrypts the message using the key whose identifier value matches the one in the message.

When you use secret key-based encryption, the value of the key name is embedded in the incoming encrypted message. The Web Services Security implementation asks the key locator for the key with the name that matches the name in the message and decrypts the message using the key.

Keys:

Keys are used for XML signature and encryption.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

There are two predominant kinds of keys used in the current Web Services Security implementation:
- Public key - such as Rivest Shamir Adleman (RSA) encryption and Digital Signature Algorithm (DSA) encryption
- Secret key - such as Data Encryption Standard (DES) encryption

In public key-based signature, a message is signed using the sender private key and is verified using the sender public key. In public key-based encryption, a message is encrypted using the receiver public key and is decrypted using the receiver private key. In secret key-based signature and encryption, the same key is used by both parties.

While the current implementation of Web Services Security can support both kinds of keys, there are a few items to note:

- Secret key-based signature is not supported.
- The format of the message differs slightly between public key-based encryption and secret key-based encryption.

Trusted ID evaluator:

The trusted ID evaluator is an abstraction of the mechanism that evaluates whether the given ID name is to be trusted. The trusted ID evaluator is typically used by the eventual receiver in a multi-hop environment.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* applications.

Depending upon the implementation, you can use various types of infrastructure to store a list of the trusted IDs, such as:

- Plain text file
- Database
- Lightweight Directory Access Protocol (LDAP) server

The Web Services Security implementation (com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator) invokes the trusted ID evaluator and passes the identity name of the intermediary as a parameter. If the identity is evaluated and deemed trustworthy, the procedure continues. Otherwise, an exception is created and the procedure is stopped.

Login mappings:

Login mappings, found in the `ibm-webservices-bnd.xmi` Extended Markup Language (XML) file, contains a mapping configuration. This mapping configuration defines how the Web Services Security handler maps the token <ValueType> element that is contained within the security token extracted from the message header, to the corresponding authentication method. The token <ValueType> element is contained within the security token extracted from a SOAP message header.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

The sender-side Web Services Security handler generates and attaches security tokens based on the <AuthMethods> element that is specified in the deployment descriptor. For example, if the authentication method is BasicAuth, the sender-side security handler generates and attaches UsernameToken (with both user name and password) to the SOAP message header. The Web Services Security run time uses the Java Authentication and Authorization Service (JAAS) javax.security.auth.callback.CallbackHandler interface as a security provider to generate security tokens on the client side (or when web services are acting as client).

The sender security handler invokes the handle() method of a javax.security.auth.callback.CallbackHandler interface implementation. This implementation creates the security token and passes the token back to the sender security handler. The senders security handler constructs the security token based on the authentication information in the callback array. The security handler then inserts the security token into the Web Services Security message header.

The CallbackHandler interface implementation that you use to generate the required security token is defined in the <loginBinding> element in the ibm-webservicesclient-bnd.xmi Web Services Security binding file. For example,

```
<loginBinding xmi:id="LoginBinding_1052760331526" authMethod="BasicAuth"
      callbackHandler="com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler"/>
```

The <loginBinding> element associates the com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler interface with the BasicAuth authentication method. WebSphere Application Server provides the following set of CallbackHandler interface implementations you can use to create various security token types:

com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
: If there is no basic authentication data defined in the login binding information (this information is not the same as the HTTP basic authentication information), the previous token type prompts for user name and password through a login panel. The implementation uses the basic authentication data defined in the login binding. Use this CallbackHandler with the BasicAuth authentication method. Do not use this CallbackHandler implementation on the server because it prompts you for login binding information.

com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
: If basic authentication data is not defined in the login binding (this information is not the same as the HTTP basic authentication information), the implementation prompts for the user name and password using standard in (stdin). The implementation uses the basic authentication data defined in the login binding. Use this CallbackHandler implementation with the BasicAuth authentication method. Do not use this CallbackHandler implementation on the server because it prompts you for login binding information.

: **Restriction:** If you have a multi-threaded client and multiple threads attempt to read from standard in at the same time, all the threads will not successfully obtain the user name and password information. Therefore, you cannot use the com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler implementation with a multi-threaded client where multiple threads might attempt to obtain data from standard in concurrently.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
: This CallbackHandler implementation does not prompt. Rather, it uses the basic authentication data defined in the login binding (this information is not the same as the HTTP basic authentication information). This CallbackHandler implementation is meant for use with the BasicAuth authentication method. You must define the basic authentication data in the login binding information for this CallbackHandler implementation. You can use this implementation when web services is running as a client and needs to send basic authentication (<wsse:UsernameToken>) to the downstream call.

com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler
: The CallbackHandler generates Lightweight Third Party Authentication (LTPA) tokens from the run as JAAS Subject (invocation Subject) of the current WebSphere Application Server security context. However, if basic authentication data is defined in the login binding information (not the HTTP basic authentication information), the implementation uses the basic authentication data and LTPA token generated. The **Token Type URI** and **Token Type Local Name** values must be defined in the login binding information for this CallbackHandler implementation. The token value type is used to validate the token to the request sender and request receiver binding configuration. The Web Services Security run time inserts the LTPA token as a binary security token

(<wsse:BinarySecurityToken>) into the message SOAP header. The value type is mandatory. (See LTPA for more information). Use this CallbackHandler implementation with the LTPA authentication method.

Figure 1 shows the sender security handler in the request sender message process.

Figure 19. Request sender SOAP message process

You can configure the receiver-side security server to support multiple authentication methods and multiple types of security tokens. The following steps describe the request sender SOAP message process:

1. After receiving a message, the receiver Web Services Security handler compares the token type (in the message header) with the expected token types configured in the deployment descriptor.
2. The Web Services Security handler extracts the security token form the message header and maps the token <ValueType> element to the corresponding authentication method. The mapping configuration is defined in the <loginMappings> element in the `ibm-webservices-bnd.xmi` XML file. For example:

```
<loginMappings xmi:id="LoginMapping_1051977980074" authMethod="LTPA"
      configName="WSLogin">
     <callbackHandlerFactory xmi:id="CallbackHandlerFactory_1051977980081"
     classname="com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl"/>
      <tokenValueType xmi:id="TokenValueType_1051977980081"
      uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2" localName="LTPA"/>
</loginMappings>
```

   The com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory interface is a factory for javax.security.auth.callback.CallbackHandler.
3. The Web Services Security run time initiates the factory implementation class and passes the authentication information from Web Services Security header to the factory class through the set methods.
4. The Web Services Security run time invokes the newCallbackHandler() method to obtain the javax.security.auth.CallbackHandler object, which generates the required security token.
5. When the security handler receives an LTPA BinarySecurityToken, it uses the WSLogin JAAS login configuration and the newCallbackHandler() method to validate the security token. If none of the expected token types are found in the SOAP message Web Services Security header, the request is rejected with an SOAP fault. Otherwise, the token type is used to map to a JAAS login configuration for token validation. If authentication is successful, a JAAS Subject is created and associated with the running thread. Otherwise, the request is rejected with a SOAP fault.

   The following table shows the authentication methods and the JAAS login configurations.

Table 40. Authentication methods and JAAS login configurations. The authentication methods map to the JAAS login configuration for token validation.

Authentication method	JAAS login configuration
BasicAuth	WSLogin
Signature	system.wssecurity.Signature
LTPA	WSLogin
IDAssertion	system.wssecurity.IDAssertion

Figure 2 shows the receiver security handler in the request receiver message process.

Figure 20. Request receiver SOAP message process

The default <LoginMapping> is defined in the following files:

- Cell-level `ws-security.xml` and server-level `ws-security.xml` files

If nothing is defined in the binding file information, the `ws-security.xml` default is used. However, an administrator can override the default by defining a new <LoginMapping> element in the binding file.

6. The client reads the default binding information in the `${install_dir}/properties/ws-security.xml` file.
7. The server runtime component loads the following files if they exist:
   - Cell-level `ws-security.xml` file and the server-level `ws-security.xml` file. The two files are merged in the run time to form one effective set of default binding information.

   On a base application server, the server run time component only loads the server-level `ws-security.xml` file. The server-side `ws-security.xml` file and the application Web Services Security binding information are managed using the administrative console. You can specify the binding information during application deployment using the administrative console. The Web Services Security policy is defined in the deployment descriptor extension (`ibm-webservicesclient-ext.xmi`) and the bindings are stored in the IBM binding extension (`ibm-webservicesclient-bnd.xmi`). However, the `${install_dir}/properties/ws-security.xml` file defines the default binding value for the client. If the binding information is not specified in the binding file, the run time reads the binding information from the default `${install_dir}/properties/ws-security.xml` file.

XML encryption:

Extensible Markup Language (XML) encryption is a specification developed by World Wide Web (WWW) Consortium (W3C) in 2002 that contains the steps to encrypt data, the steps to decrypt encrypted data, the syntax to represent XML encrypted data, the information used to decrypt the data, and a list of encryption algorithms such as triple Data Encryption Standard (DES), Advanced Encryption Standard (AES), and Rivest-Shamir-Adleman algorithm (RSA).

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

You can apply XML encryption to an XML element, XML element content, and arbitrary data, including an XML document. For example, suppose that you need to encrypt the <CreditCard> element shown in the example 1.

Example 1: Sample XML document

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Example 2: XML document with a common secret key:

Example 2 shows the XML document after encryption. The <EncryptedData> element represents the encrypted <CreditCard> element. The <EncryptionMethod> element describes the applied encryption algorithm, which is triple DES in this example. The <KeyInfo> element contains the information to retrieve a decryption key, which is a <KeyName> element in this example. The <CipherValue> element contains the ciphertext obtained by serializing and encrypting the <CreditCard> element.

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
   <EncryptionMethod
      Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
      <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
        <KeyName>John Smith</KeyName>
      </KeyInfo>
      <CipherData>
```

```
        <CipherValue>ydUNqHkMrD...</CipherValue>
      </CipherData>
  </EncryptedData>
</PaymentInfo>
```

Example 3: XML document encrypted with the public key of the recipient:

In example 2, it is assumed that both the sender and recipient have a common secret key. If the recipient has a public and private key pair, which is most likely the case, the <CreditCard> element can be encrypted as shown in example 3. The <EncryptedData> element is the same as the <EncryptedData> element found in Example 2. However, the <KeyInfo> element contains an EncryptedKey.

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <EncryptionMethod
      Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
    <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
      </EncryptedKey>
    </KeyInfo>
    <CipherData>
      <CipherValue>ydUNqHkMrD...</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

XML Encryption in the WSS-Core:

WSS-Core specification is under development by Organization for the Advancement of Structured Information Standards (OASIS). The specification describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. The message confidentiality is realized by encryption based on XML Encryption.

The WSS-Core specification supports encryption of any combination of body blocks, header blocks, their sub-structures, and attachments of a SOAP message. The specification also requires that when you encrypt parts of a SOAP message, you prepend a reference from the security header block to the encrypted parts of the message. The reference can be a clue for a recipient to identify which encrypted parts of the message to decrypt.

The XML syntax of the reference varies according to what information is encrypted and how it is encrypted. For example, suppose that the <CreditCard> element in example 4 is encrypted with either a common secret key or the public key of the recipient.

Example 4: Sample SOAP message:

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <CreditCard Limit='5,000' Currency='USD'>
        <Number>4019 2445 0277 5567</Number>
        <Issuer>Example Bank</Issuer>
        <Expiration>04/02</Expiration>
      </CreditCard>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The resulting SOAP messages are shown in Examples 5 and 6. In these example, the <ReferenceList> and <EncryptedKey> elements are used as references, respectively.

Example 5: SOAP message encrypted with a common secret key:

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <ReferenceList xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <DataReference URI='#ed1'/>
      </ReferenceList>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>John Smith</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 6: SOAP message encrypted with the public key of the recipient:

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI='#ed1'/>
        </ReferenceList>
      </EncryptedKey>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Relationship to digital signature:

The WSS-Core specification also provides message integrity, which is realized by a digital signature based on the XML-Signature specification.

A combination of encryption and digital signature over common data introduces cryptographic vulnerabilities.

Request sender:

The security handler on the request sender side of the SOAP message enforces the security constraints, located in the `ibm-webservicesclient-ext.xmi` file, and bindings, located in the `ibm-webservicesclient-bnd.xmi file`. These constraints and bindings apply both to Java Platform, Enterprise Edition (Java EE) application clients or when web services are acting as a client. The security handler acts on the security constraints before sending the SOAP message. For example, the security handler might digitally sign the message, encrypt the message, create a time stamp, or insert a security token.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

The security handler on the request sender side of the SOAP message enforces the security constraints, located in the `ibm-webservicesclient-ext.xmi` file, and the bindings, located in the `ibm-webservicesclient-bnd.xmi` file. These constraints and bindings apply both to Java Platform, Enterprise Edition (Java EE) application clients or when web services are acting as a client. The security handler acts on the security constraints before sending the SOAP message. Request sender security constraints must match the security constraint requirements defined in the request receiver. For example, the security handler might digitally sign the message, encrypt the message, create a time stamp, or insert a security token. You can specify the following security requirements for the request sender and apply them to the SOAP message:

Integrity (digital signature)
You can select multiple parts of a message to sign digitally. The following list contains the integrity options:
- Body
- Time stamp
- Security token

Confidentiality (encryption)
You can select multiple parts of a message to encrypt. The following list contains the confidentiality options:
- Body content
- Username token

Security token
You can insert only one token into the message. The following list contains the security token options:
- Basic authentication, which requires both a user name and a password
- Identity assertion, which requires a user name only
- X.509 binary security token
- Lightweight Third Party Authentication (LTPA) binary security token
- Custom token , which is pluggable and supports custom-defined tokens in the SOAP message

Timestamp
You can have a time stamp to indicate the timeliness of the message.
- Timestamp

Request receiver:

The request receiver defines the security requirement of the SOAP message. The security handler on the request receiver side of the SOAP message enforces the security specifications that are defined in the IBM extension deployment descriptor (`ibm-webservices-ext.xmi`) and bindings (`ibm-webservices-bnd.xmi`).

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

The security constraint for request sender must match the security requirement of the request receiver for the server to accept the request. If the incoming SOAP message does not meet all the security requirements defined, then the request is rejected with the appropriate fault code returned to the sender. For security tokens, the token is validated using Java Authentication and Authorization Service (JAAS) login configuration and authenticated identity is set as the identity for the downstream invocation.

For example, if there is a security requirement to have the SOAP body digitally signed by Joe Smith and if the SOAP body of the incoming SOAP message is not signed by Joe Smith, then the request is rejected.

You can define the following security requirements for the request receiver:

Required integrity (digital signature)
You can select multiple parts of a message to sign digitally. The following list contains the integrity options:
- Body
- Time stamp
- Security token

Required confidentiality (encryption)
You can select multiple parts of a message to encrypt. The following list contains the confidentiality options:
- Body content
- Token

You can have multiple security tokens. The following list contains the security token options:
- Basic authentication, which requires both a user name and a password
- Identity assertion, which requires a user name only
- X.509 binary security token
- Lightweight Third Party Authentication (LTPA) binary security token
- Custom token, which is pluggable and supports custom-defined tokens validated by the JAAS login configuration

Received time stamp
You can have a time stamp for checking the timeliness of the message.
- Time stamp

Response sender:

The response sender defines the security requirements of the SOAP response message. The security handler acts on the security constraints that are defined for the response in the IBM extension deployment descriptors.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

The IBM extension deployment descriptors are located in the `ibm-webservices-ext.xmi` file and the bindings, located in the `ibm-webservices-bnd.xmi` file. The security handler signs, encrypts, or generates the time stamp for the SOAP response message before the response is send to the caller.

Integrity constraints (digital signature)
You can select which parts of the message are digitally signed.
- Body
- Time stamp

Confidentiality (encryption)
You can encrypt the body content of the message.

Time stamp
You can have a time stamp for checking the timeliness of the message.

The security constraints that apply to the SOAP response message must match the security requirements defined in the response receiver. Otherwise, the response is rejected by the response receiver (caller).

Response receiver:

The response receiver defines the security requirements of the response received from a request to a web service. The security constraints for response sender must match the security requirements of the response receiver. If the constraints do not match, the response is not accepted by the caller or the sender.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

The security handler enforces the security constraints based on the security requirements defined in the IBM extension deployment descriptor, located in the `ibm-webservicesclient-ext.xmi` file and in the bindings, located in the `ibm-webservicesclient-bnd.xmi` file.

For example, the security requirement might have the response SOAP body encrypted. If the SOAP body of the SOAP message is not encrypted, the response is rejected and the appropriate fault code is communicated back to the caller of the web services.

You can specify the following security requirements for a response receiver:

Required integrity (digital signature)
You can select which parts of a message are digitally signed. The following list contains the integrity options:
- Body
- Time stamp

Required confidentiality (encryption)
You can encrypt the body content of the message.

Received time stamp
You can have a time stamp for checking the timeliness of the message.

Identity assertion in a SOAP message:

Identity assertion is a method for expressing the identity of the sender (for example, user name) in a SOAP message. When identity assertion is used as an authentication method, the authentication decision is performed based only on the name of the identity and not on other information, such as passwords and certificates.

Important: There is an important distinction between Version 5.x and Version 6.0.x and later applications. The information in this article supports Version 5.x applications only that are used with WebSphere Application Server Version 6.0.x and later. The information does not apply to Version 6.0.x and later applications.

Identity assertion involves:

ID type

The Web Services Security implementation in WebSphere Application Server can handle these identity types:

User name
Denotes the user name, such as the one in the local operating system (for example, `alice`). This name is embedded in the <Username> element within the <UsernameToken> element.

DN
Denotes the distinguished name (DN) for the user, such as `"CN=alice, O=IBM, C=US"`. This name is embedded in the <Username> element within the <UsernameToken> element.

X.509 certificate
Represents the identity of the user as an X.509 certificate instead of a string name. This certificate is embedded in the <BinarySecurityToken> element.

Managing trust

The intermediary host in the SOAP message itinerary can assert claimed identity of the initial sender. Two methods (called trust mode) are supported for this assertion:

Basic authentication
The intermediary adds its user name and password pair to the message.

Signature
The intermediary digitally signs the <UsernameToken> element of the initial sender.

Note: This trust mode does not support the X.509 certificate ID type.

Typical scenario

ID assertion is typically used in the multi-hop environment where the SOAP message passes through one or more intermediary hosts. The intermediary host authenticates the initial sender. The following scenario describes the process:

1. The initial sender sends a SOAP message to the intermediary host with some embedded authentication information. This authentication information might be a user name and a password pair with an Lightweight Third Party Authentication (LTPA) token.
2. The intermediary host authenticates the initial sender according to the embedded authentication information.
3. The intermediary host removes the authentication information from the SOAP message and replaces it with the <UsernameToken> element, which contains a user name.
4. The intermediary host asserts the trust according to the trust mode.
5. The intermediary host sends the updated SOAP message to the ultimate receiver.
6. The ultimate receiver checks the trust against the intermediary host information according to the configured trust mode. Also, the trusted ID evaluator is invoked.
7. If trust is established by the final receiver, the receiver invokes the web service under the authorization of the user name (that is, the initial sender) in the SOAP message.

Security token:

A security token represents a set of claims made by a client that might include a name, password, identity, key, certificate, group, privilege, and so on.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Web Services Security provides a general-purpose mechanism to associate security tokens with messages for single message authentication. A specific type of security token is not required by Web Services Security. Web services security is designed to be extensible and support multiple security token formats to accommodate a variety of authentication mechanisms. For example, a client might provide proof of identity and proof of a particular business certification.

A security token is embedded in the SOAP message within the SOAP header. The security token within the SOAP header is propagated from the message sender to the intended message receiver. On the receiving side, the WebSphere Application Server security handler authenticates the security token and sets up the caller identity on the running thread.

Pluggable token support:

Pluggable security token support provides plug-in points to support customer security token types, including token generation, token validation, and client identity mapping to a WebSphere Application Server identity that is used by the Java Platform, Enterprise Edition (Java EE) authorization engine. Moreover, the pluggable token generation and validation framework supports XML-based tokens to be inserted into the web service message header and validated on the receiver-side validation.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* applications.

You can extend the WebSphere Application Server login mapping mechanism to handle new types of authentication tokens. WebSphere Application Server provides a pluggable framework to generate security tokens on the sender-side of the message and to validate the security token on the receiver-side of the message. The framework is based on the Java Authentication and Authorization Service (JAAS) Application Programming Interfaces (APIs).

Use the javax.security.auth.callback.CallbackHandler implementation to create a new type of security token following these guidelines:

- Use a constructor that takes a user name (a string or null, if not defined), a password (a char[] or null, if not defined) and java.util.Map (empty, if properties are not defined).
- Use handle() methods that can process the following implementations:
  - javax.security.auth.callback.NameCallback
  - javax.security.auth.callback.PasswordCallback
  - com.ibm.wsspi.wssecurity.auth.callback.XMLTokenCallback
  - com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl

  If:

  1. Either the javax.security.auth.callback.NameCallback or the javax.security.auth.callback.PasswordCallback implementation is populated with data, then a <wsse:UsernameToken> element is created.
  2. com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl is populated, the <wsse:BinarySecurityToken> element is created from the com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl implementation.

3. com.ibm.wsspi.wssecurity.auth.callback.XMLTokenCallback is populated, a XML-based token is created based on the Document Object Model (DOM) element that is returned from the XMLTokenCallback.

Encode the token byte by using the security handler and not by using the javax.security.auth.callback.CallbackHandler implementation.

You can implement the com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory interface, which is a factory for instantiating the javax.security.auth.callback.CallbackHandler implementation. For your own implementation, you must provide the javax.security.auth.callback.CallbackHandler interface. The Web Service Security run time instantiates the factory implementation class and passes the authentication information from the web services message header to the factory class through the setter methods. The Web Services Security run time then invokes the newCallbackHandler() method of the factory implementation class to obtain an instance of the javax.security.auth.CallbackHandler object. The object is passed to the JAAS login configuration.

The following is an example the definition of the CallbackHandlerFactory interface:

```
public interface com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory {
       public void setUsername(String username);
       public void setRealm(String realm);
       public void setPassword(String password);
       public void setHashMap(Map properties);
       public void setTokenByte(byte[] token);
       public void setXMLToken(Element xmlToken);
       public CallbackHandler newCallbackHandler();
```

Migrating Web Services Security

You can migrate Web Services Security bindings from an older version to the latest version of WebSphere® Application Server. The product migration function handles most of the migration process, but your input and action is required for specific configurations in order to complete the migration.

Migration of JAX-WS Web Services Security bindings from Version 6.1

You can migrate Web Services Security bindings from an older version to Version 7.0 and later of WebSphere Application Server. Migration of the JAX-WS bindings in Version 6.1 Feature Pack for Web Services takes place during the product migration to Version 7.0 and later.

The product migration handles most of the WS-Security migration process, but your input and action is required for specific configurations in order to complete the migration. Following are some examples of configurations that require manual migration steps:

- Using callers on Version 6.1 Feature Pack for Web Services.

  WebSphere Application Server Version 7.0 and later supports the capability to specify a preference order for callers. In the situation where only a single caller is present in the bindings, product migration automatically assigns an order of **1** to the single caller that is present. However, if there are multiple callers, warnings are logged during migration, and you must manually assign the caller preference order. Set the order attribute for each caller using the administrative console, or the administration commands, after product migration is completed. Read about call collection and configuring the callers for general and default bindings for instructions on setting the caller order using the administrative console. See the topic "WS-Security policy and binding properties" for information on using the administration commands.
- Using multiple username tokens in the default bindings.

  During product migration for Version 6.1 default bindings, all UsernameToken generators and consumers in the bindings will be migrated. However, if multiple username token generators, or consumers, are used, a warning is logged during migration. The warning states that a maximum of two username token generators and two username token consumers are allowed, and that one of each pair of token generators or consumers must have the `com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed` property set to `true`. You must manually select which username token consumer or generator to keep, and which token has the `com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed` property, using the administrative console, or administration commands.

- To use the administrative console to set the property, access the WS-Security 6.1 default bindings as instructed in the “Policy set bindings settings for WS-Security” topic, and click the **Authentication and Protection** link. Add, remove or edit the username token consumers and generators as needed, following the instructions in the “WS-Security authentication and protection for general bindings” topic.
- To use the administration commands to set the property, and to add, delete and edit the username token generators or consumers as needed, refer to the “WS-Security policy and bindings properties” topic for instructions.

- Using multiple token generators and token consumers of the same type in Version 6.1 default bindings.

  During product migration for Version 6.1 default bindings, all token generators and token consumers for supporting tokens are migrated. In the situation where multiple token generators and token consumers of the same supporting token type are found a warning is logged during migration. This warning states that only a single token consumer and token generator for each supporting token type must be present. You must manually select which token consumer or generator to use for the repeating supporting token type, using the administrative console or administration commands. To use the administrative console to select the token, access the WS-Security 6.1 default bindings as instructed in the “Policy set bindings settings for WS-Security” topic, and select the **Authentication and Protection** link. Add, edit or remove token consumers and generators as needed, following the instructions in the “WS-Security authentication and protection for general bindings” topic.

Migrating JAX-RPC Web Services Security applications to Version 8.5 applications

Migration of a Java Platform, Enterprise Edition (Java EE) Version 1.3 application that uses Web Services Security to a Java EE Version 1.4 application is possible.

Before you begin

You can install Java Platform, Enterprise Edition (Java EE) Version 1.3 applications that use Web Services Security on a WebSphere Application Server Version 8.5 server. However, if you want Java EE Version 1.3 applications to use the Web Services Security (WSS) Version 1.0 or 1.1 specifications and the other new features added in Version 8.5, you must migrate the Java EE Version 1.3 applications to Java EE Version 1.4.

About this task

Complete the following steps to migrate a Version 1.3 application, along with the Web Services Security configuration information, to a Version 8.5 application:

Procedure

1. Save the original Java EE Version 1.3 application. You need the Web Services Security configuration files of the Java EE Version 1.3 application to recreate the configuration in the new format for the Java EE Version 1.4 application.
2. Use the Java Platform, Enterprise Edition (Java EE) Migration Wizard in an assembly tool to migrate the Java EE Version 1.3 application to Java EE Version 1.4.

   Important: After you migrate to Java EE Version 1.4 using the Java EE Migration Wizard, you cannot view the Java EE Version 1.3 extension and binding information within an assembly tool. You can view the Java EE Version 1.3 Web Services Security extension and binding information using a text editor. However, do not edit the extension and binding information using a text editor. The Java EE Migration Wizard does not migrate the Web Services Security configuration files to the new format in the Java EE Version 1.4 application. Rather the wizard is used to migrate your files from Java EE Version 1.3 to Version 1.4.

   To access the Java EE Migration Wizard, complete the following steps:

   a. Right-click the name of your application.
   b. Click **Migrate** > **Java EE Migration Wizard**.

3. Manually delete all of the Web Services Security configuration information from the binding and extension files of the application that is migrated to Java EE Version 1.4.
   a. Delete the `<securityRequestReceiverServiceConfig>` and `<securityResponseSenderServiceConfig>` sections from the server-side `ibm-webservices-ext.xmi` extension file.
   b. Delete the `<securityRequestReceiverBindingConfig>` and `<securityResponseSenderBindingConfig>` sections from the server-side `ibm-webservices-bnd.xmi` binding file.
   c. Delete the `<securityRequestSenderServiceConfig>` and `<securityResponseReceiverServiceConfig>` sections from the client-side `ibm-webservicesclient-ext.xmi` extension file.
   d. Delete the `<securityRequestSenderBindingConfig>` and `<securityResponseReceiverBindingConfig>` sections from client-side `ibm-webservicesclient-bnd.xmi` binding file.
4. Recreate the Web Services Security configuration information in the new Java EE Version 1.4 format. At this stage, because the application is already migrated to the Java EE Version 1.4, use an assembly tool to configure the original Web Services Security information in the new Version 8.5 format. For more information on assembly tools, see the related information.

Results

This task provides general information about how to migrate Java EE Version 1.3 applications to Java EE Version 1.4.

What to do next

The following articles contain some general scenarios that map some of the basic Web Services Security information specified in a Java EE Version 1.3 application to a Java EE Version 1.4 application and specify this information using an assembly tool. The Web Services Security configuration information is contained in four configuration files: two server-side configuration files and two client-side configuration files. The migration of all of the configuration information is divided into four sections; one for each configuration file. When you recreate the Web Services Security information in the new Java EE Version 1.4 format, it is recommended that you configure the extensions and binding files in the following order:
1. Configure the `ibm-webservices-ext.xmi` server-side extensions file. For more information, see "Migrating the JAX-RPC server-side extensions configuration."
2. Configure the `ibm-webservicesclient-ext.xmi` client-side extensions file. For more information, see "Migrating the client-side extensions configuration" on page 366.
3. Configure the `ibm-webservices-bnd.xmi` server-side bindings file. For more information, see "Migrating the server-side bindings file" on page 368.
4. Configure the `ibm-webservicesclient-bnd.xmi` client-side bindings file. For more information, see "Migrating the client-side bindings file" on page 370.

Migrating the JAX-RPC server-side extensions configuration:

You can migrate the Web Services Security server-side extensions configuration for a Java Platform, Enterprise Edition (Java EE) Version 1.3 application to a Java EE Version 1.4 application for the JAX-RPC programming model.

About this task

The following table lists the mappings for the top-level sections under the server-side **Security Extensions** tab within an assembly tool from a Java EE Version 1.3 application to a Java EE Version 1.4 application.

Table 41. The mapping of the configuration sections. Use the extensions configuration information for migration.

Java EE Version 1.3 extensions configuration	Java EE Version 1.4 extensions configuration
Request Receiver Service Configuration Details	Request Consumer Service Configuration Details
Response Sender Service Configuration Details	Response Generator Service Configuration Details

For information about the assembly tools that are available for WebSphere Application Server Version 6.0.*x*, see the assembly tools information.

Consider the following steps to migrate the server-side extensions from a Java EE Version 1.3 application to a Java EE Version 1.4 application. These steps are dependent upon your specific configuration.

Procedure

- Import the Java EE Version 1.3 application into an assembly tool and identify all the message parts that are required to be signed and encrypted. The message parts are listed in the Required Integrity and Required Confidentiality sections under the Request Receiver Service Configuration Details section. In a Java EE Version 1.4 application, these message parts map to the Message parts field of the **Required integrity** and **Required confidentiality** dialogs windows within the assembly tool.

  To specify these message parts within an assembly tool, complete the following steps in the Web Services Editor. The steps are based on typical scenarios, but the steps are not all-inclusive.

  1. Click the **Extensions** tab.
  2. Navigate to the Required integrity subsection within the Request Consumer Service Configuration Details section.
  3. Specify each message part to be signed in the Message Parts field.

  For example, if the message part in the Java EE Version 1.3 application is `body`, you need to specify `body` in the Message parts keyword field. Similarly, on the **Extensions** tab, configure the message parts to be encrypted using the Required Confidentiality dialog. Also, for all the message parts that are migrated from a Java EE Version 1.3 application, you must select **http://www.ibm.com/websphere/webservices/wssecurity/dialect-was** in the Message parts dialect field and **Required** in the Usage type field.
- Optional: Configure the Required Security Token and Caller Part sections on the **Extensions** tab if the authentication method of BasicAuth is configured under the Login Config section of the Java EE Version 1.3 application. When you configure the Required Security Token section, select **Username** in the name field and **Required** in the Usage type field within the Required Security Token Dialog window. The following table shows how the authentication method values for a Java EE Version 1.3 application map to the token type values within the Java EE Version 1.4 application.

Table 42. Authentication method to token type mappings. Use the extensions configuration information for migration.

Login Config Authentication method values in the Java EE Version 1.3 extensions configuration	Token type values in the Java EE Version 1.4 extensions configuration
`BasicAuth`	`UsernameToken`
`Signature`	`X509 certificate`
`LTPA`	`LTPAToken`

If the authentication method value is IDAssertion within the Login Config section, the token type that you must specify in the Java EE Version 1.4 application depends upon the IDType value within the IDAssertion section. The following table shows how the IDType values for Java EE Version 1.3 application map to the token type values in the Java EE Version 1.4 application.

Table 43. IDType values to token type mappings. Use the extensions configuration information for migration.

IDType values in the Java EE Version 1.3 application extensions configuration	Token type values in the Java EE Version 1.4 application extensions configuration
`X509Certificate`	`X509 certificate`
`Username`	`Username`

- Select the appropriate token type in the Name field of the Call Part Dialog window based on the previous two tables. Select the `Username` token type when you are configuring the caller part for the basic authentication method. Configuring the other token types in the Caller part dialog is similar to configuring token types in the Required Security Token dialog. If you need to map the IDAssertion authentication method from a Java EE Version 1.3 application to a Java EE Version 1.4 application, select the **Use IDAssertion** option and configure the ID assertion section of the Caller Part Dialog window. The Trust Mode field under the IDAssertion section maps to the Trust method name field of the Trust method property section in the Caller Part Dialog window. If Signature is selected for the Trust method, specify the Required Integrity part that specifies the signature of the trusted intermediary certificate.
- Configure a nonce in the Version 8.5 Binding Configurations section if nonce is specified in the Add Authentication Method dialog under Login Config within the Java EE Version 1.3 application extensions configuration.

  Important: Nonce is configured in the bindings for a Java EE Version 1.4 application and not in the extensions.

  To configure a nonce on the Binding Configurations tab, set the `com.ibm.wsspi.wssecurity.token.username.verifyNonce` property in the Token Consumer configuration for the Username token.
- Configure the Add Timestamp section to migrate the time stamp information if the <addReceivedTimestamp> element is configured in the Java EE Version 1.3 extensions. To migrate the Response Sender Service Configuration Details section in the Java EE Version 1.3 extensions, identify all of the message parts listed within the Integrity and Confidentiality sections. Configure these message parts using the Integrity and Confidentiality dialogs under the Response Generator Service Configuration details section. This configuration is similar to the configuration for Required Integrity and Required Confidentiality, with the exception of the Order field in the Integrity Dialog. The value of this Order field specifies the order in which the message parts specified in the Message Parts field are digitally signed or encrypted in the SOAP message. For example, the extensions contain the following information:
  - One integrity entry called `int_part1` with a value of `1` in the Order field
  - One confidentiality entry called `conf_part1` with a value of `2` in the Order field

  In this example, the message parts that are specified by the `int_part1` integrity entry are signed before the message parts specified by the `conf_part1` confidentiality entry are encrypted. The same rule for the order attribute applies for multiple integrity or confidentiality elements.

Results

These steps describe the types of information that you need to migrate the Web Services Security server-side extensions for a Java EE Version 1.3 application to a Java EE Version 1.4 application.

What to do next

Migrate the client-side extensions for a Java EE Version 1.3 application to a Java EE Version 1.4 application. For more information, see "Migrating the client-side extensions configuration."

Migrating the client-side extensions configuration:

You can migrate the Web Services Security client-side extensions configuration for a Java Platform, Enterprise Edition (Java EE) Version 1.3 application to a Java EE Version 1.4 application.

About this task

The following table lists the mappings of the top-level sections under the client-side **Security Extensions** tab for Web Services Security from a Java EE Version 1.3 application to a Java EE Version 1.4 application.

Table 44. The mapping of the configuration sections. Use the extensions configuration information for migration.

Java EE Version 1.3 security extensions for Web Services Security	Java EE Version 1.4 extensions for Web Services Security
Request Sender Configuration	Request Generator Configuration
Response Receiver Configuration	Response Consumer Configuration

Consider the following steps to migrate the client-side extensions configuration from a Java EE Version 1.3 application to a Java EE Version 1.4 application. These steps are dependent upon your specific configuration. The steps are based on typical scenarios, but the steps are not all-inclusive.

Procedure

- Migrate the message parts that you need to sign or encrypt from the Integrity and Confidentiality sections in the Java EE Version 1.3 application to the Integrity and Confidentiality sections on the **WS Extensions** tab in an assembly tool for a Java EE Version 1.4 application.
- Configure the Security Token section under the Request Generator Configuration on the **WS Extensions** tab if Login Config section is configured in the Java EE Version 1.3 extensions configuration. When you configure the security token, select the token type in the Token type field that matches the authentication method value of the Login Config in the Java EE Version 1.3 application. For example, if the authentication method in the Java EE Version 1.3 extensions configuration is BasicAuth, then select **Username** in the Token type field within the assembly tool. For more information on how the authentication methods for Web Services Security map from a Java EE Version 1.3 application to a Java EE Version 1.4 application, see Table 42 on page 365. If the authentication method is IDAssertion, there is no action required because in a Java EE Version 1.4 application the identity assertion configuration is not required in the client-side extensions configuration. In a Java EE Version 1.4 application, the identity assertion configuration is specified in the server-side extensions configuration and in the client-side bindings configuration.
- Migrate the Required Integrity and Required Confidentiality sections by configuring the Required Integrity and Required Confidentiality sections in an assembly tool. Migrating the Response Receiver Configuration section is similar to migrating the Request Receiver Service Configuration Details section of the server-side extensions configuration. For more information, see “Migrating the JAX-RPC server-side extensions configuration” on page 364.
- Migrate the nonce configuration in the Login Config section in a Java EE Version 1.3 extensions configuration for Web Services Security to a Java EE Version 1.4 application.

  Important: Nonce is not configured in a Java EE Version 1.4 extension file for Web Services Security. Rather, it is configured in the binding file for Web Services Security.

  To configure a nonce in the binding file, define the com.ibm.wsspi.wssecurity.token.username.addNonce property in the token generator of the username token.
- Configure the Add Timestamp section under the Request Generator Configuration in the assembly tool if the **Add Created Time Stamp** option is configured in the Java EE Version 1.3 extensions.

Results

This set of steps describe the types of information that you need to migrate the client-side extensions configuration for Web Services Security for a Java EE Version 1.3 application to a Java EE Version 1.4 application.

What to do next

Migrate the server-side bindings configuration for a Java EE Version 1.3 application to a Java EE Version 1.4 application. For more information, see "Migrating the server-side bindings file."

Migrating the server-side bindings file:

You can migrate the server-side bindings configuration for a Java Platform, Enterprise Edition (Java EE) Version 1.3 application to a Java EE Version 1.4 application.

About this task

The following table lists the mappings of the top-level sections under the server-side **Binding Configurations** tab from a Java EE Version 1.3 application to a Java EE Version 1.4 application.

Table 45. The mapping of the configuration sections. Use the binding configuration information for migration.

Java EE Version 1.3 Binding Configurations	Java EE Version 1.4 Binding Configurations
Request Receiver Binding Configuration Details	Request Consumer Service Binding Configuration Details
Response Sender Binding Configuration Details	Response Generator Binding Configuration Details

Consider the following steps to migrate the server-side bindings from Java EE Version 1.3 to Java EE Version 1.4. These steps are dependent upon your specific configuration. The steps are based on typical scenarios, but the steps are not all-inclusive.

Procedure

- Migrate the configuration information under the Request Receiver Binding Configuration Details section of a Java EE Version 1.3 application.
  1. Migrate any trust anchor information that is specified in the Java EE Version 1.3 application to Java EE Version 1.4 using the Trust Anchor dialog.
  2. Migrate the information under the certificate store list that is specified in the Java EE Version 1.3 application to Java EE Version 1.4 by configuring the Certificate Store List section in the Java EE Version 1.4 application.
  3. Configure the key locator and token consumer information that is referenced from the Key Information dialog window. The configuration of the key locator and the token consumer depends upon the key information type. For example, if an X.509 certificate that is embedded in the `<wsse:Security>` security header is used for digital signature, complete the following steps:
     a. For configuring the key locator, specify the `com.ibm.wsspi.wssecurity.keyinfo.X509TokenKeyLocator` class as the key locator class and do not specify a key store.
     b. For configuring the token consumer, select the `com.ibm.wsspi.wssecurity.token.509TokenConsumer` class, specify `X509 certificate token` for the value type Uniform Resource Identifier (URI), and specify `system.wssecurity.X509BST` in the jaas.config.name field. Also, you must specify the certificate path settings (the trust anchor reference and the certificate store reference) as part of the token consumer configuration.
  4. Explicitly specify the key information type in the Key Information Dialog window. In a Java EE Version 1.3 application, the key information type, such as the security token reference and the key identifier, is not explicitly specified. The key information type is implied by the configuration. In a Java EE Version 1.4 application, you must specify the key information type explicitly using the Key Information Dialog when you have digital signature or encryption information in the binding file. Before you configure the key information, make sure that you have configured the key locator and token consumer information that is referenced from the Key Information dialog.

     When you configure the key information for either digital signature or encryption, you need to specify the correct key information type. The value of the key information type depends upon the type of mechanism that is used to reference the security token that is used for digitally signing or

encrypting. The following information describes the Security token reference (or Direct reference) and the Key identifier, which are the most common, recommended key information types that are used for digitally signing and encrypting:

Security token reference (or Direct reference)
The security token is directly referenced using the Uniform Resource Identifiers (URIs). The following `<KeyInfo>` element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo>
   <wsse:SecurityTokenReference>
      <wsse:Reference URI="#mytoken" />
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Key identifier
The security token is referenced using an opaque value that uniquely identifies the token. The algorithm that is used for generating the `KeyIdentifier` value depends upon the token type. For example, a hash of the important elements of the security token is used for generating the `KeyIdentifier` value. The following `<KeyInfo>` element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   <wsse:SecurityTokenReference>
      <wsse:KeyIdentifier ValueType="wsse:X509v3">/62wXO...</wsse:KeyIdentifier>
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

In the Key Information Dialog window, specify the names of the key locator and the token consumer that you configured previously. The Key name field is optional for the consumer side.

5. Migrate the information in the Signing Information section by configuring the Signing Information, Part References, and Transforms sections.
   - Specify the Signature method and Canonicalization method algorithms in the Signing Information Dialog window.
   - Specify the Digest method algorithm in the Part Reference Dialog window.
6. Migrate the information under the Encryption Information section. In the Encryption Information Dialog window, select the name of the Key Information element that is configured for encryption, and specify the `RequiredConfidentiality` part. Verify that the value for the selected `RequiredConfidentiality` part is the same name as the Required Confidentiality part that is configured in the extension file.

   The Login Mapping section in the Java EE Version 1.3 application maps to the Token Consumer configuration for the type of token that is specified by the authentication method. For example, to migrate a Login Mappings configuration that uses the `BasicAuth` authentication method, configure a token consumer for the username token. To configure a token consumer for a username token, complete the following steps:
   a. Select the `com.ibm.wsspi.wssecurity.UsernameTokenConsumer` token consumer class.
   b. Specify the name of the Required Security Token configuration from the Extensions within in the **Security Token** field.
   c. Select **Username Token** for value type.
   d. Specify the `system.wssecurity.UsernameToken` value in the **jaas.config.name** field.

- Migrate the configuration information in the Response Sender Binding Configuration Details section of the Java EE Version 1.3 bindings file to the Response Generator Binding Configuration Details section of the Java EE Version 1.4 application. Configuring the Response Generator section is very similar to configuring the Request Consumer section.
  1. Migrate the information from the Key Locators section by using the Key Locator Dialog window in an assembly tool.
  2. Configure a token generator, which is referenced in the Key Information Dialog window. You must configure a token generator for every security token that is generated in the SOAP message. If the token generator is for an X.509 certificate that is used for digital signature or encryption, complete the following steps:

a. For configuring the key locator, specify the `com.ibm.wsspi.wssecurity.keyinfo.X509TokenKeyLocator` class as the key locator class and do not specify a key store.
b. For configuring the token generator, select the `com.ibm.wsspi.wssecurity.X509TokenGenerator` class and specify `X509 certificate token` for the value type Uniform Resource Identifier (URI). The key store information that is specified for the token generator is the same information that is used for configuring the key locator. Therefore, the keystore information from the Key Locators configuration in a Java EE Version 1.3 application is used to configure the key locator and the token generator in a Java EE Version 1.4 application.
c. In the Token Generator Dialog window, specify the key store information that is required by the callback handler to obtain the key information that is required for generating the token.
d. For the callback handler, select the `com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler` class.

3. Specify the names of the key locator and the token generator in the Key Information Dialog window that you configured previously. The Key name is required for the generator side. The key that is specified in the Key Information Dialog window must exist in the list of keys that is specified in the key locator configuration. Also, migrating the Signing Information and the Encryption Information configurations is similar to migrating the Signing Information and the Encryption Information configurations for the Request Receiver Binding Configuration section. Configuring the key information for the response generator section is similar to configuring the key information for the request consumer section.

Results

This set of steps describes the types of information that you need to migrate the server-side bindings configuration for a Java EE Version 1.3 application to a Java EE Version 1.4 application.

What to do next

Migrate the client-side binding configuration for a Java EE Version 1.3 application to a Java EE Version 1.4 application. For more information, see "Migrating the client-side bindings file."

Migrating the client-side bindings file:

You can migrate the Web Services Security client-side binding configuration for a Java Platform, Enterprise Edition (Java EE) Version 1.3 application to a Java EE Version 1.4 application.

About this task

The following table lists the mapping of the top-level sections under the client-side **Port Bindings** tab within a Java EE Version 1.3 application to a Java EE Version 1.4 application.

Table 46. The mapping of the configuration sections. Use the binding configuration information for migration.

Java EE Version 1.3 binding configuration for Web Services Security	Java EE Version 1.4 binding configuration for Web Services Security
Security Request Sender Binding Configuration	Security Request Generator Binding Configuration
Security Response Receiver Binding Configuration	Security Response Consumer Binding Configuration

Consider the following steps to migrate the client-side binding configuration from a Java EE Version 1.3 application to a Java EE Version 1.4 application. These steps are dependent upon your specific configuration. The steps are based on typical scenarios, but the steps are not all-inclusive.

Procedure

- Migrate the information in the Security Request Sender Binding Configuration section in a Java EE Version 1.3 application to a Java EE Version 1.4 application. The migrations process for the Security

Request Sender Binding Configuration section is similar to the process for the Response Sender Binding Configuration Details section in the server-side binding configuration. For more information, see "Migrating the server-side bindings file" on page 368.

- Migrate the information in the Key Locators, Signing Information, and the Encryption Information sections of the Java EE Version 1.3 application to a Java EE Version 1.4 application. The migration process for these elements on the client side is similar to migration process on the server side. For more information, see "Migrating the server-side bindings file" on page 368.
- Migrate the information in the Login Bindings section in a Java EE Version 1.3 application to a Java EE Version 1.4 application. The migration of the Login Bindings section depends upon the value of the authentication method. If the authentication method is `BasicAuth` or `IDAssertion`, configure a token generator for the username token. If the authentication method is LTPA, select the `com.ibm.wsspi.wssecurity.token.LTPATokenGenerator` class as the token generator class. If the client-side bindings for the web service uses IDAssertion, complete the following steps:
  1. Configure a token generator for the authentication token of the original client.
  2. Define the `com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed` property and set its value to `true` in the Token Generator Dialog window within an assembly tool. If the original client is using a username token for authentication and if the target web service is using `BasicAuth` for authentication, configure the following token generators in the client-side binding file:
     – The username token of the original client. You must set the `com.ibm.wsspi.wssecurity.token.IDAssertion.isUsed` property in the token generator of the original client.
     – The username token of the intermediary web service.
- Migrate the Security Response Receiver Binding Configuration section from a Java EE Version 1.3 application to a Java EE Version 1.4 application. Migrating the Security Response Receiver Binding Configuration section is similar to migrating the Request Receiver Binding Configuration Details section of the server-side bindings configuration. Migrate this information under the Security Response Consumer Binding Configuration section. For more information, see "Migrating the server-side bindings file" on page 368.

  To configure a nonce in the binding file, define the `com.ibm.wsspi.wssecurity.token.username.addNonce` property in the token generator of the username token.

Results

This set of steps describe the types of information that you need to migrate the Web Services Security client-side bindings configuration for a Java EE Version 1.3 application to a Java EE Version 1.4 application.

What to do next

Verify that you have migrated both the server-side and the client-side extension and binding configurations for a Java EE Version 1.3 application to a Java EE Version 1.3 application. For more information, see "Migrating JAX-RPC Web Services Security applications to Version 8.5 applications" on page 363.

View web services client deployment descriptor:

Use this page to view your client deployment descriptor.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

Before you begin this task, the web services application must be installed.

By completing this task, you can gather information that enables your to maintain or configure binding information. After the web services application is installed, you can view the web services deployment descriptors.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules >** ***URI_name*** .
3. Under Web Services Properties, click **View web services client deployment descriptor extension**.

The information in the following implementation indicates how to configure your application-level bindings. If the web server is acting as a client, the default bindings are used. To configure the server-level bindings, which are the defaults, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. To configure the cell-level bindings, click **Security > Web services**.

If you are using any of the following configurations, verify that the deployment descriptor is configured properly:

- “Request signing”
- “Request encryption”
- “BasicAuth authentication” on page 373
- “Identity (ID) Assertion authentication with BasicAuth TrustMode” on page 373
- “Identity (ID) Assertion authentication with the Signature TrustMode” on page 373
- “Response digital signature verification” on page 374
- “Response decryption” on page 374

Request signing

If the integrity constraints (digital signature) are specified, verify that you configured the signing information in the binding files.

To configure the signing parameters, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules >** ***URI_name***.
3. Under Web Services Security properties, click **Web Services: Client security bindings**.
4. In the Response receiver binding column, click **Edit > Signing information > New**.

To configure the key locators, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Key locators**.

Request encryption

If the confidentiality constraints (encryption) are specified, verify that you configured the encryption information in the binding files.

To configure the encryption parameters, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Client security bindings**.
4. In the Response receiver binding column, click **Edit > Encryption Information > New**.

To configure the key locators, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers > *server_name***.
2. Under Additional properties, click **Web Services: Default bindings for Web Services Security > Key locators**.

BasicAuth authentication

If BasicAuth authentication is configured as the required security token, specify the callback handler in the binding file to collect the basic authentication data. The following list contains the CallBack support implementations:

com.ibm.wsspi.wssecurity.auth.callback.GuiPromptCallbackHandler
This implementation prompts for basic authentication information, the user name and password, in an interface.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This implementation reads the basic authentication information from the binding file.

com.ibm.wsspi.wssecurity.auth.callback.StdPromptCallbackHandler
This implementation prompts for a user name and password using the standard in (stdin) prompt.

To configure the login binding information, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Client security bindings**.
4. Under Request sender bindings, click **Edit > Login binding**.

Identity (ID) Assertion authentication with BasicAuth TrustMode

Configure a login binding in the bindings file with a com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler implementation. Specify a BasicAuth user name and password that a trusted ID evaluator on a downstream server trusts.

To configure the login binding information, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Client security bindings**.
4. Under Request sender bindings, click **Edit > Login binding**.

Identity (ID) Assertion authentication with the Signature TrustMode

Configure the signing information in the bindings file with a signing key pointing to a key locator. The key locator contains the X.509 certificate that is trusted by the downstream server.

To configure ID assertion, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers > *server_name***.

2. Under Additional properties, click **JAX-WS and JAX-RPC security runtime > Login mappings > IDAssertion**.

To configure the login binding information, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Client security bindings**.
4. Under Request sender bindings, click **Edit > Login binding**.

Response digital signature verification

If the integrity constraints, which require a signature, are defined, verify that you configured the signing information in the binding files.

To configure the signing parameters, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Client security bindings**.
4. In the Response receiver binding column, click **Edit > Signing information > New**.

To configure the trust anchors, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Trust anchors > New**.

To configure the collection certificate store, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Collection certificate store > New**.

Response decryption

If the confidentiality constraints (encryption) are specified, verify that you defined the encryption information.

To configure the encryption information, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Client security bindings**.
4. In the Response receiver binding column, click **Edit > Encryption information > New**.

To configure the key locators, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Key locators**.

View web services server deployment descriptor:

Use this page to view your server deployment descriptor settings.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

Before you begin this task, the web services application must be installed.

By completing this task, you can gather information that enables you to maintain or configure binding information. After the web services application is installed, you can view the web services deployment descriptors.

To view this administrative console page, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Properties, click **View web services server deployment descriptor**.

WebSphere Application Server, Network Deployment has three levels of bindings: application-level, server-level, and cell-level. The information in the following implementation descriptions indicate how to configure your application-level bindings. To configure the server-level bindings, which are the defaults, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

To configure the cell-level bindings, click **Security** > **JAX-WS and JAX-RPC security runtime**.

- “Request digital signature verification”
- “Request decryption” on page 376
- “Basic authentication” on page 376
- “Identity (ID) assertion authentication with the BasicAuth TrustMode” on page 377
- “Identity (ID) assertion authentication with the signature TrustMode” on page 377
- “Response signing” on page 378
- “Response encryption” on page 378

Request digital signature verification

If the integrity constraints, which require a signature, are defined, verify that you configured the signing information in the binding files.

To configure the signing parameters, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Properties, click **Web services: Server security bindings**.

4. Under Request receiver binding, click **Edit > Signing information**.

To configure the trust anchor, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Trust anchors**.

To configure the collection certificate store, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Collection certificate store**.

To configure the key locators, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Key locators**.

Request decryption

If the confidentiality constraints (encryption) are specified, verify that the encryption information is defined.

To configure the encryption information parameters, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Server security bindings**.
4. Under Request receiver binding, click **Edit** > **Encryption information**.

To configure the key locators, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Key locators**.

Basic authentication

If BasicAuth authentication is configured as the required security token, specify the callback handler in the binding file to collect the basic authentication data. The following list contains callback support implementations:

com.ibm.wsspi.wssecurity.auth.callback.GuiPromptCallbackHandler
The implementation prompts for BasicAuth information (user name and password) in an interface panel.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This implementation reads the BasicAuth information from the binding file.

com.ibm.wsspi.wssecurity.auth.callback.StdPromptCallbackHandler
This implementation prompts for a user name and password using the standard in (stdin) prompt.

To configure the login mapping information, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Login mappings**.

Identity (ID) assertion authentication with the BasicAuth TrustMode

Configure a login binding in the bindings file with a `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler` implementation. Specify a user name and password for basic authentication that a TrustedIDEvaluator on a downstream server trusts.

To configure the login mapping information, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Login mappings**.

Identity (ID) assertion authentication with the signature TrustMode

Configure the signing information in the bindings file with a signing key that points to a key locator. The key locator contains the X.509 certificate that is trusted by the downstream server.

To configure the login mapping information, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Login mappings**.

The Java Authentication and Authorization Service (JAAS) uses `WSLogin` as the name of the login configuration. To configure JAAS, complete the following steps:

1. Click **Security** > **Global security**.
2. Under Authentication, click **Java Authentication and Authorization Service** > **Application logins**.

The value of the <TrustedIDEvaluatorRef> tag in the binding must match the value of the <TrustedIDEvaluator> name.

To configure the trusted ID evaluators, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Trusted ID evaluators**.

Response signing

If the integrity constraints (digital signature) are defined, verify that you have the signing information configured in the binding files.

To specify the signing information, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Server security bindings**.
4. In the Request receiver binding column, click **Edit** > **Signing information**.

To configure the key locators, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Key locators**.

Response encryption

If the confidentiality constraints (encryption) are specified, verify that the encryption information is defined.

To specify the encryption information, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Server security bindings**.
4. Under Request receiver binding, click **Edit** > **Encryption information**.

To configure the key locators, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Key locators**.

Developing applications that use Web Services Security

The Web Services Security specification provides a flexible framework for building secure web services to implement message content integrity and confidentiality. The Web Services Security service programming model supports this flexible framework by providing extension points to integrate new token formats, and

methods to obtains keys needed for message protection. The application server programming model provides Web Services Security programming application programming interfaces (WSS API) for securing SOAP messages.

Configuring HTTP basic authentication for JAX-RPC web services programmatically

You can configure HTTP basic authentication for Java API for XML-based RPC (JAX-RPC) web services by programmatically modifying HTTP properties.

Before you begin

This task is one of three ways that you can configure HTTP basic authentication. You can also configure HTTP basic authentication with an assembly tool or with the administrative console.

If you programmatically configure HTTP basic authentication, the properties are configured in the Stub or Call instance. If you choose to configure HTTP basic authentication with the administrative console or an assembly tool, the Web Services Security binding information is modified. The values that are set programmatically take precedence over the values defined in the binding.

About this task

The HTTP basic authentication that is discussed in this topic is orthogonal to WS-Security and is distinct from basic authentication that WS-Security supports. WS-Security supports basic authentication token, not HTTP basic authentication.

Configure HTTP basic authentication programmatically with the following steps.

Procedure

Set the properties in the Stub or Call instance for a Web service or a web service client. You can set properties with the following constant names:

```
javax.xml.rpc.Call.USERNAME_PROPERTY
javax.xml.rpc.Call.PASSWORD_PROPERTY
javax.xml.rpc.Stub.USERNAME_PROPERTY
javax.xml.rpc.Stub.PASSWORD_PROPERTY
```

Example

The following code enables you to configure basic authentication programmatically:

```
Properties prop = new Properties();
InitialContext ctx = new InitialContext(prop);
Service service = (Service)ctx.lookup("java:comp/env/service/StockQuoteService");
QName portQname = new QName("http://httpchannel.test.wsfvt.ws.ibm.com", "StockQuoteHttp");
StockQuote sq = (StockQuote)service.getPort(portQname, StockQuote.class);
((javax.xml.rpc.Stub) sq)._setProperty(javax.xml.rpc.Stub.USERNAME_PROPERTY, "myUser");
((javax.xml.rpc.Stub) sq)._setProperty(javax.xml.rpc.Stub.PASSWORD_PROPERTY, "myPwd");
```

Developing message-level security for JAX-WS web services

JAX-WS applications can be secured with Web Services Security in one of two ways. The application can be secured using policy sets, or through the use of the Web Services Security API (WSS API). The WSS API can only be used to secure a JAX-WS client application. The Web Services Security service programming interface (WSS SPI) provides additional programming interfaces for securing web services.

Web Services Security API programming model:

The application server programming model provides Web Services Security programming application programming interfaces (WSS API) for securing SOAP messages.

The API programming model is an interface-based programming model that is based on Web Services Security Version 1.1 standards, but the design also includes support for Web Services Security Version 1.0

for securing SOAP messages. The WSS API programming model implementation is a simplified version, which is based on an early draft proposal of JSR-183, which is the JSR for defining Java API binding for Web Services Security. By design, because the application code is programmed to the interface, any application code that is programmed with the open source implementation should be able to run on the WebSphere Application Server with minimal changes or no changes at all.

The configuration model for web services has also been redesigned from a deployment descriptor model to a policy set model. Web Services Security can be enabled by either using a policy set that is configured by using the administrative console, or by using the WSS API for configuration. The functions provided by the policy set configurations are the same as the functions supported by the WSS API for the Web Services Security run time. However, the security policy that is defined using policy sets has a higher priority over the WSS API. When the WSS API and the policy set are both used in the application, the default behavior is for the security policy from the policy set to be enforced and the WSS API to be ignored. To use the WSS API in the application, you must make sure that there is no policy set attached to the application or to the application resources, or make sure there is no security policy in the attached policy set.

You can still use your existing JAX-RPC applications with Web Services Security; however, those applications cannot take advantage of the Web Services Security Version 1.1 functions, such as configuring the security policy using a policy set, OM filter performance improvements, WSS API, Web Services Secure Conversation (WS-SecureConversation), Kerberos token and the associated SHA-1 key for message protection and identity propagation, and Web Services Trust (WS-Trust) features.

In order to take advantage of the Web Services Security Version 1.1 functions, you must rewrite an existing JAX-RPC application as a JAX-WS application, manually re-configure the security constraints to a policy set, and perform code migration of the DOM-based SPIs to the OM-based SPIs.

For example, when using the JAX-WS programming model, the improved design of the pluggable token framework allows the same security implementation to be used for both the API and policy sets. The framework uses the JAAS Login Module and JAAS Callback Handler for token creation and token validation.

The following diagrams illustrate differences between the programming models.

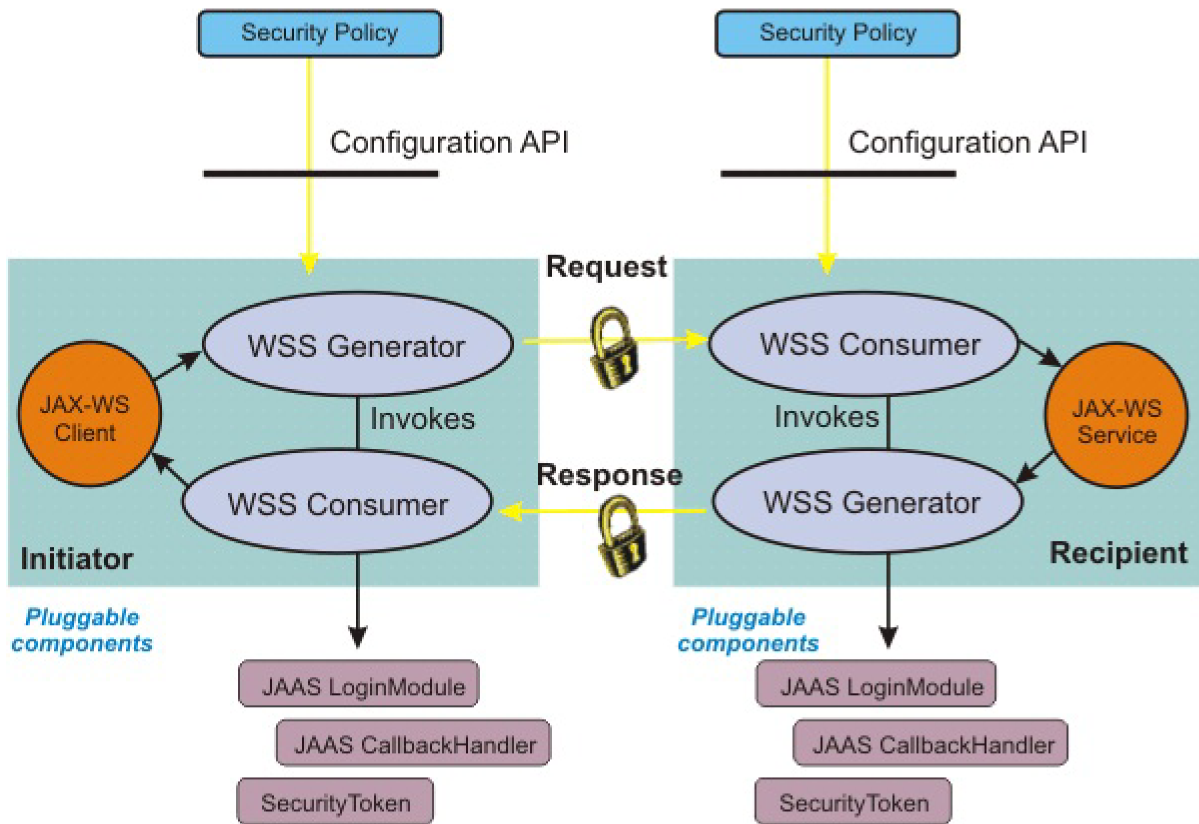


Figure 21. Pluggable token architecture using the JAX-WS programming model

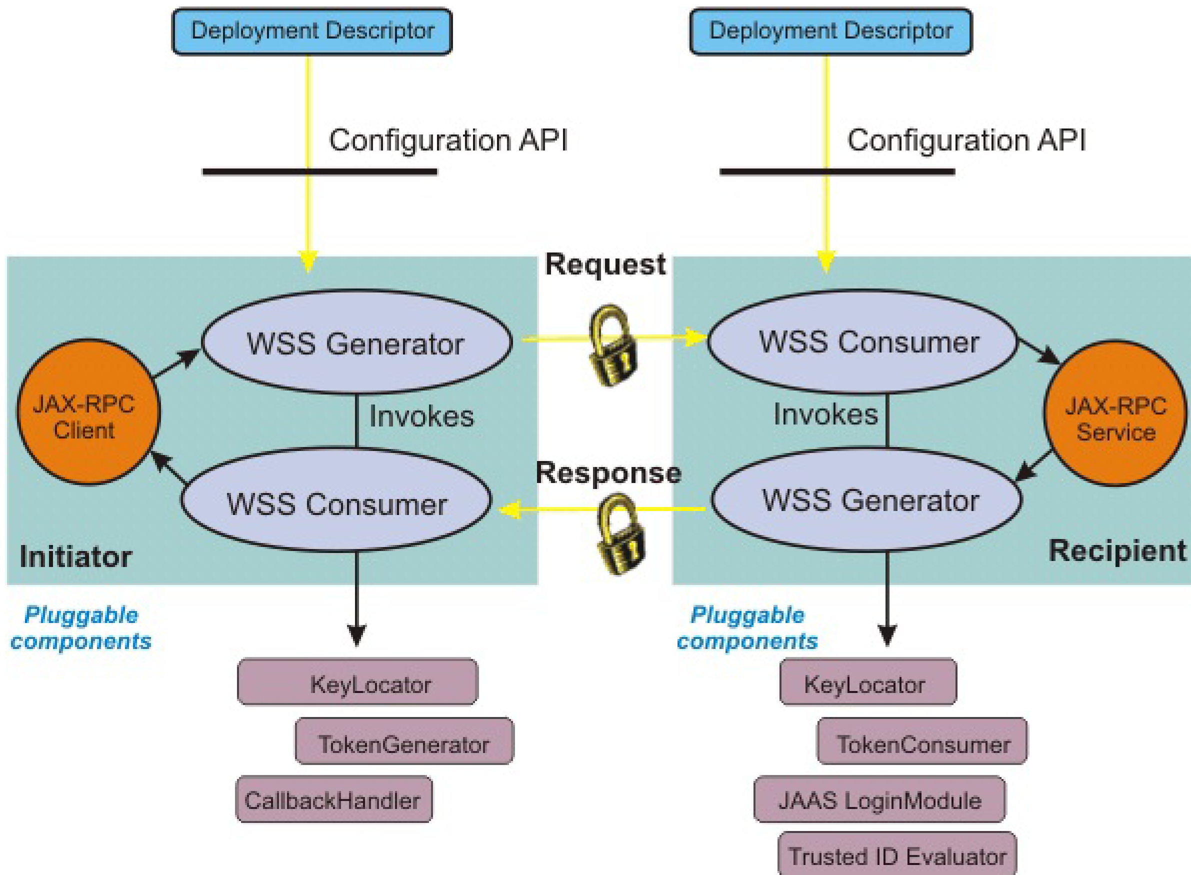


Figure 22. Pluggable token architecture using the JAX-RPC programming model

What is supported when using the WSS APIs

The WSS API can only be used on the client. You can use the Java SE 6 client, the J2EE Application client, or a server client (a service provider acting as client) using the API to secure SOAP message with message-level security.

You should have Web Services Security (WSS) knowledge to use the WSS APIs. Before using the WSS API, keep in mind that the WSS API:

- Are Java-based interfaces.
- Are implemented by using a factory model (WSSFactory).
- Supports the WS-Security Version 1.0 and 1.1 standards, which include the Username and X.509 token profiles, Versions 1.0 and 1.1.
- Are very XML centric.
- Include an object-oriented design which simplifies the APIs.
- Are task oriented and allow common usage scenarios, such as: signing the body and encrypting the SOAP message body content.
- Are flexible and extensible, and they let you to extend the token type support.
- Are based on the provider framework and allow the use of different data models to be used, such as: AXIOM or DOM.
- Provides application programmer with better control and flexibility in applying WSS in their applications.

The default values for the WSS API are predefined and are part of the Web Services Security run time. Default values are provided for:

- The duration of the timestamp
- The signing algorithm, canonicalization algorithm, digest method, transform algorithm, security token reference method and signed parts such as the SOAP body, Web Services Addressing headers and the time stamp.
- The key encryption algorithm, data encryption algorithm, security token reference method, and encrypted parts such as the SOAP body content.

The signature validation has similar default values as the signature (signing information). Similarly, decryption has similar default values as encryption.

What is not supported when using the WSS APIs

The WSS API provided with the application server does not support the following function:
- The application programming model is JAX-WS, meaning JAX-RPC (JSR-109) applications are not supported.
- The WSS API is available in the synchronous message exchange of the JAX-WS client application. However, the WSS API are not supported for the asynchronous client.
- WSS API support is available only for the requester and not for the provider.
- The identity assertion semantic programming model is not supported in the WSS API because identity assertion is not part of the Web Services Security Version 1.0 standard. However, you can use the WSS API to add Identity Assertion semantic in the token processing.

WS-Trust and WS-SecureConversation scenarios

There are several ways to secure the WS-Trust SOAP messages:
- Using the bootstrap policy defined in the policy set.
- Using the WSS API, which supports WS-SecureConversation.
- Enabling dynamic policy for the provider so that the client can retrieve the provider-side policy at run time.

An application would use the WSS API to acquire a security context token for programmatic API-based secure conversation. The WebSphere Application Server trust service provides an application the ability to request a security token for access to a service. The scope and focus of the trust service is only for a WebSphere Application Server Security Context Token (SCT) for WS-SecureConversation.

The WS-SecureConversation and WS-Trust scenarios focus on the inter-operability functions, such as the configuration and runtime interaction of various components. You would use the WSS API to secure the bootstrap RST and RSTR to acquire the security context token from the trust service. After acquiring the security context token, a Derived Key Token is created by using the WSS API. Then the Derived Key Token can be used for signature and encryption.

There are two conditions when using the WSS API to secure the SOAP message with Web Services Security:
- Generation of the secure SOAP message, which is in the request generator application code.
- Consuming of the secured SOAP message, which is in the response consumer application code.

In both cases, a Java exception class com.ibm.websphere.wssecurity.wssapi.WSSException is provided if an error is encountered.

Web services client security context

When the JAX-WS client invokes web services, the current security context that is constructed by the security handler is stored in the RequestContext object. By default, the security context in the JAX-WS

web services client runtime environment is reconstructed for the next web services request invocation. You can preserve the security context for subsequent web services invocations. An example of this is a scenario where the security policy requires the client to send a username security token with the user name and password. When the client sends the first request to invoke the service, you are prompted to enter the required user name and password. The user name and password is saved in a Username SecurityToken token in a Subject in the security context. To avoid being prompted to enter the same user name and password again in subsequent request invocations, you can preserve the security context. There are two methods to preserve the security context: 1) configure the client run time to automatically preserve the client security context for subsequent request invocations; or 2) preserve the security context manually.

To configure the JAX-WS client run time environment to automatically preserve the security context, set the Java system property `com.ibm.websphere.wssecurity.context.management` to `true`. When this system property is true, the JAX-WS client run time copies the security context constructed by the security handler to the RequestContext automatically, and the context is used for subsequent request invocations.

To manually preserve the security context, use the following sample code:

```
// First request
Service svc = Service.create(...);
svc.addPort(...);
Dispatch<String> dispatch = svc.createDispatch(...);
Map<String, Object> requestContext = dispatch.getRequestContext();
String response = dispatch.invoke(body.toString());

Object securityContext = requestContext.get(com.ibm.wsspi.websvcs.Constants.WEBSPHERE_SECURITY_CONTEXT);

// Subsequent request

Dispatch<String> dispatch = svc.createDispatch(...);
Map<String, Object> requestContext = dispatch.getRequestContext();
Object securityContext = requestContext.put(com.ibm.wsspi.websvcs.Constants.WEBSPHERE_SECURITY_CONTEXT, securityContext);
```

Service Programming Interfaces (SPI):

The Web Services Security service programming interface (WSS SPI) provides programming interfaces for securing Web Services Security.

The Web Services Security specification provides a flexible framework for building secure web services to implement message content integrity and confidentiality. The specification does not define specific token formats, but instead associates separate profile documents that define various security token formats and semantics for using those tokens. The Web Services Security service programming model supports the flexible framework by providing extension points to integrate with new token formats, and with methods to obtains keys needed for message protection. Web Services Security uses this programming model to implement support for the standard X.509 token profile, the Username token profile, and the Kerberos token profiles. The programming model is also used to implement support for the LTPA security token, and for new security token types.

The Web Service Security run time token generation and token consuming Service Programming Interfaces (SPI) have been redesigned so that the same security token interface and JAAS Login Module implementation can be used for both the WSS API and the SPI. The WSS SPI for the service provider extends the security token types and provides keys and deriving keys for signing, signature verification, encryption and decryption.

The Web Services Security service programming model provides mechanisms to process custom security tokens, to use custom token in signing and encryption, and to retrieve encryption and signing keys. The Web Services Security service programming interfaces for the JAX-RPC run time, and for the JAX-WS run time, are similar, but not identical.

JAX-RPC run time

The plug-in programming interfaces for the JAX-RPC run time consist of the TokenGenerator, KeyLocator, and JAAS CallbackHandler for outbound message processing, and the TokenConsumer, KeyLocator, and JAAS LoginModule for inbound message processing.

Token Generator, KeyLocator, and Callback Handler
: The TokenGenerator class is responsible for formatting the security token to the XML element. This class calls the CallbackHandler class that is specified in the TokenGeneratorConfig object, which obtains the security token input data, and then stores the resulting security token in the Subject object private credentials.

Token Consumer, KeyLocator and JAAS LoginModule
: The KeyLocator class is responsible for obtaining the required key for signing and encrypting SOAP message elements from a key store that is specified by the KeyStoreConfig and the KeyLocatorConfig configuration. The TokenConsumer class extracts the token data from the XML security token representation, and stores it in the JAAS Subject using a JAAS LoginModule. The specified KeyLocator class is invoked to find the required key for verifying the digital signature and decrypting the SOAP message elements.

JAX-WS run time

The plug-in programming interfaces for the JAX-WS run time are based on the JAAS programming model for both inbound and outbound SOAP message processing. The JAAS LoginModule and CallbackHandler are responsible for processing the security tokens in SOAP messages. The Login Module and Callback Handler both retrieve and generate tokens, and store the SecurityToken objects in the run time. They replace the functionality of the TokenGenerator, TokenConsumer, and KeyLocator interfaces.

Due to the differences in the programming models, any WebSphere Application Server or custom SPI implementation from the Web Services Security Version 6.1 run time is not supported to run on the Web Services Security run time with the Version 6.1 Feature Pack for Web Services, or the Version 7.0 and later Web Services Security runtime. However, the Web Services Security Version 6.1 run time is supported simultaneously with the Version 6.1 Feature Pack for Web Services, meaning the Version 6.1 SPI implementations are still supported through the original run time. Before using the new Web Services Security run time, a code migration is required to reprogram the Version 6.1 DOM-based SPIs to the AXIOM-based SPIs in the Feature Pack for Web Services, before the SPI can be used.

Developing SAML applications:

Use the SAML library application programming interface (API), the SAMLTokenFactory, to configure token parameters, create a SAML token, and bind the created token to a service request. The SAML trust client API provides helper functions that send WS-Trust SOAP requests to the specified external Security Token Service (STS).

About this task

The SAMLTokenFactory API creates SAML tokens through various method signatures. The API also instantiates runtime configuration objects related to the SAML token requester, as well as the recipient.

The WS-Trust Client API for SAML includes the WSSTrustClient class, the WSSTrustClientValidateResult class, and other configuration utility classes.

The following topics provide more information about developing SAML applications using the APIs.

WS-Trust client API:

The WS-Trust client application programming interface (API) includes the WSSTrustClient class, the WSSTrustClientValidateResult class, and other configuration utility classes. The WSSTrustClient class

provides helper functions that send WS-Trust SOAP requests to the specified external Security Token Service (STS) so that the STS can issue or validate one or more SAML assertions and other types of security tokens.

Overview

WebSphere Application Server includes WS-Trust client function, implemented through the WSSTrustClient class, that sends WS-Trust SOAP requests to a specified external Security Token Service (STS). Using the trust requests, the STS can issue one or more SAML assertions or other types of security tokens. The WSSTrustClient class supports the OASIS WS-Trust Version 1.3 specification, and also the WS-Trust Version 1.2 specification. In addition, the SOAP Version 1.1 and SOAP Version 1.2 specifications are supported by the function.

The sample code which follows demonstrates how a web services client uses the WSSTrustClient API to request a SAML bearer token. In the explanatory text which precedes the code sample, the term **SAML token** is used interchangeably with the term **SAML assertion**.

The WSSTrustClient class

You can copy the sample code into an assembly tool application, such as Rational Application Developer, and start using the code after completing the configuration steps. Use the WSSTrustClient class, together with other SAML APIs, to build useful SAML functions. Refer to the SAML API Javadoc for more information.

The WSSTrustClient class is an abstract class and has two concrete implementations: a WS-Trust Version 1.3 implementation and a WS-Trust v1.2 implementation. On line 50 of the code sample, the SAMLWSTrustClientExample web services client code invokes the WSSTrustClient.getInstance(ProviderConfig) method to retrieve the WS-Trust v1.3 implementation. The getInstance() method takes a single ProviderConfig object, which specifies configuration data that are relevant to the SAML token issuer. A ProviderConfig object is also instantiated in the sample code on line 32. The client code sends WS-Trust Version 1.3 request messages to a target STS endpoint. In the sample, the endpoint is https://MyCompany/Trust/13/UsernameMixed. To use the sample code, replace this example STS endpoint with the specific STS endpoint you plan to use.

Note: Starting with WebSphere Application Server Release 8, you can use the `com.ibm.websphere.wssecurity.wssapi.token.SAMLToken` class in Web Services Security (WSS) application programming interface (API). When there is no concern of confusion we use the term `SAMLToken` instead of using its complete package name. You can use WSS API to request SAMLToken processing from an external Security Token Service (STS), to propagate SAMLTokens in SOAP request messages, and to use a symmetric or asymmetric key identified by SAMLTokens to protect SOAP messages.

The WSS API SAML support complements the `com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory` and `com.ibm.websphere.wssecurity.wssapi.trust.WSSTrustClient` interfaces. SAMLTokens that are generated using the `com.ibm.websphere.wssecurity.wssapi.WSSFactory newSecurityToken()` method can be processed by the `SAMLTokenFactory` and `WSSTrustClient` programming interfaces. Conversely, SAMLTokens that are generated by `SAMLTokenFactory` or returned by `WSSTrustClient` can be used in WSS API. Deciding which API to use in your application depends on your specific needs. WSS API SAML support is self contained in the sense that it provides functionality equivalent to that of the `SAMLTokenFactory` and `WSSTrustClient` interfaces as far as web services client applications are concerned. The `SAMLTokenFactory` interface has additional functions to validate SAMLTokens and to create the JAAS Subject that represents authenticated SAMLTokens. This validation is useful for the Web services provider side. When you develop applications to consume SAMLTokens, the `SAMLTokenFactory` programming interface is more suitable for you.

Example: Web services client code that uses the WSSTrustClient class

```
1.  package sample;
2.
3.   import com.ibm.websphere.wssecurity.wssapi.WSSException;
4.   import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
5.   import com.ibm.websphere.wssecurity.wssapi.trust.WSSTrustClient;
6.   import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
7.   import com.ibm.websphere.wssecurity.wssapi.XMLStructure;
8.
9.
10.  import com.ibm.wsspi.wssecurity.core.token.config.RequesterConfiguration;
11.  import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.Namespace;
12.  import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.TokenType;
13.  import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.WST13;
14.  import com.ibm.wsspi.wssecurity.trust.config.ProviderConfig;
15.  import com.ibm.wsspi.wssecurity.trust.config.RequesterConfig;
16.  import com.ibm.wsspi.wssecurity.wssapi.OMStructure;
17.
18.  import org.apache.axiom.om.OMElement;
19.  import org.apache.axis2.util.XMLPrettyPrinter;
20.
21.  import java.util.List;
22.  import java.io.ByteArrayOutputStream;
23.  import java.io.InputStream;
24.  import java.io.BufferedReader;
25.  import java.io.InputStreamReader;
26.  import java.io.IOException;
27.
28.  public class WSSTrustClientExample {
29.
30.    public static void main(String[] args) {
31.          try {
32.          ProviderConfig providerConfig = WSSTrustClient.newProviderConfig(Namespace.WST13, https://MyCompany.com/Trust/13/UsernameMixed );
33.
34.          showProviderConfigDefaultValue(providerConfig);
35.
36.          providerConfig.setPolicySetName("Username WSHTTPS default");
37.          providerConfig.setBindingName("SamlTCSample");
38.          providerConfig.setBindingScope("domain");
39.
40.
41.          RequesterConfig requesterConfig = WSSTrustClient.newRequesterConfig(Namespace.WST13);
42.
43.          showRequestConfigDefaultValue(requesterConfig);
44.
45.          requesterConfig.put(RequesterConfiguration.RSTT.APPLIESTO_ADDRESS, "https://user.MyCompany:9443/WSSampleSei/EchoService12");
46.          requesterConfig.put(RequesterConfiguration.RSTT.TOKENTYPE, TokenType.SAML11);
47.          requesterConfig.put(RequesterConfiguration.RSTT.KEYTYPE, WST13.KEYTYPE_BEARER);
48.          requesterConfig.setSOAPNamespace(Namespace.SOAP12);
49.
50.          WSSTrustClient client = WSSTrustClient.getInstance(providerConfig);
51.          List<SecurityToken> securityTokens = client.issue(providerConfig, requesterConfig);
52.
53.          // Process SAML token
54.          if (securityTokens != null && !securityTokens.isEmpty()) {
55.            System.out.println("Number of tokens returned = " + securityTokens.size());
56.            SecurityToken token = securityTokens.get(0);
57.            if (token instanceof SAMLToken) {
58.              showSAMLToken((SAMLToken)token);
59.            } else {
60.              System.out.println("Returned token is not an SAMLToken");
61.            }
62.          } else {
63.            System.out.println("No securityToken obtained.");
64.          }
65.
66.      } catch (SoapSecurityException ex) {
67.        System.out.println("Caught exception: " + ex.getMessage());
68.        ex.printStackTrace();
69.      }
70.    }
71.
72.    private static void showProviderConfigDefaultValue(ProviderConfig providerConfig) {
73.      System.out.println("providerConfig.getApplicationName() = " + providerConfig.getApplicationName());
74.      System.out.println("providerConfig.getBindingName() = " + providerConfig.getBindingName());
75.      System.out.println("ProviderConfig.getBindingScope() = " + providerConfig.getBindingScope());
76.      System.out.println("providerConfig.getIssuerURI() = " + providerConfig.getIssuerURI());
77.
78.      System.out.println("providerConfig.getPolicySetName() = " + providerConfig.getPolicySetName());
79.      System.out.println("ProviderConfig.getPortName() = " + providerConfig.getPortName());
80.      System.out.println("providerConfig.getProvider() = " + providerConfig.getProvider());
81.      System.out.println("ProviderConfig.getServiceName() = " + providerConfig.getServiceName());
82.      System.out.println("providerConfig.getWSTrustNamespace() = " + providerConfig.getWSTrustNamespace());
83.      System.out.println("ProviderConfig.toString() = " + providerConfig.toString());
84.    }
85.
86.    private static void showRequestConfigDefaultValue(RequesterConfig requesterConfig) {
87.      System.out.println("requesterConfig.getRSTTProperties() = " + requesterConfig.getRSTTProperties());
88.      System.out.println("requesterConfig.getSecondaryParameters() = " + requesterConfig.getSecondaryParameters());
```

```
89.      System.out.println("requesterConfig.getSOAPNamespace() = " + requesterConfig.getSOAPNamespace());
90.      System.out.println("requesterConfig.getWSAddressingNamespace() = " + requesterConfig.getWSAddressingNamespace());
91.
92.      System.out.println("requesterConfig.getMessageID() = " + requesterConfig.getMessageID());
93.      System.out.println("requesterConfig.toString() = " + requesterConfig.toString());
94.    }
95.
96.    private static void showSAMLToken(SAMLToken samlToken){
97.      System.out.println("samlToken.getAssertionQName() = " + samlToken.getAssertionQName());
98.      System.out.println("samlToken.getAudienceRestriction() = " + samlToken.getAudienceRestriction());
99.      System.out.println("samlToken.getAuthenticationMethod() = " + samlToken.getAuthenticationMethod());
100.     System.out.println("samlToken.getConfirmationMethod() = " + samlToken.getConfirmationMethod());
101.     System.out.println("samlToken.getId() = " + samlToken.getId());
102.     System.out.println("samlToken.getKeyIdentifier() = " + samlToken.getKeyIdentifier());
103.     System.out.println("samlToken.getKeyIdentifierEncodingType() = " + samlToken.getKeyIdentifierEncodingType());
104.     System.out.println("samlToken.getKeyIdentifierValueType() = " + samlToken.getKeyIdentifierValueType());
105.     System.out.println("samlToken.getKeyName() = " + samlToken.getKeyName());
106.     System.out.println("samlToken.getPrincipal() = " + samlToken.getPrincipal());
107.     System.out.println("samlToken.getProperties() = " + samlToken.getProperties());
108.     System.out.println("samlToken.getReferenceURI() = " + samlToken.getReferenceURI());
109.     System.out.println("samlToken.getSAMLAttributes() = " + samlToken.getSAMLAttributes());
110.     System.out.println("samlToken.getSamlCreated() = " + samlToken.getSamlCreated());
111.     System.out.println("samlToken.getSamlExpires() = " + samlToken.getSamlExpires());
112.     System.out.println("samlToken.getSamlID() = " + samlToken.getSamlID());
113.     System.out.println("samlToken.getSAMLIssuerName() = " + samlToken.getSAMLIssuerName());
114.     System.out.println("samlToken.getSAMLNameID() = " + samlToken.getSAMLNameID());
115.     System.out.println("samlToken.getStringAttributes() = " + samlToken.getStringAttributes());
116.     System.out.println("samlToken.getSubjectDNS() = " + samlToken.getSubjectDNS());
117.     System.out.println("samlToken.getSubjectIPAddress() = " + samlToken.getSubjectIPAddress());
118.     System.out.println("samlToken.getThumbprint() = " + samlToken.getThumbprint());
119.     System.out.println("samlToken.getThumbprintEncodingType() = " + samlToken.getThumbprintEncodingType());
120.     System.out.println("samlToken.getThumbprintValueType() = " + samlToken.getThumbprintValueType());
121.     System.out.println("samlToken.getTokenQname() = " + samlToken.getTokenQname());
122.     System.out.println("samlToken.getValueType() = " + samlToken.getValueType());
123.
124.    XMLStructure samlXmlStructure = samlToken.getXML();
125.     if (samlXmlStructure != null && samlXmlStructure instanceof OMStructure) {
126.     OMStructure samlOMStructure = (OMStructure) samlXmlStructure;
127.     System.out.println("((OMStructure)samlToken.getXML()).getNode()formatted = " + formatXML(samlOMStructure.getNode()));
128.    }
129.
130.    try {
131.      InputStream is = samlToken.getXMLInputStream();
132.      if (is != null) {
133.        try {
134.          BufferedReader reader = new BufferedReader(new InputStreamReader(is));
135.          StringBuilder sb = new StringBuilder();
136.          String line = null;
137.          while ((line = reader.readLine()) != null) {
138.            sb.append(line + "\n");
139.          }
140.          System.out.println(sb.toString());
141.        } catch (Exception ex) {
142.          System.out.println("Caught exception reading from InputStream: " + ex.getMessage());
143.          ex.printStackTrace();
144.        } finally {
145.          try {
146.            is.close();
147.          } catch (IOException e) {
148.            e.printStackTrace();
149.          }
150.        }
151.      }
152.    } catch (WSSException wex) {
153.      System.out.println("Caught exception getXMLInputStream(): " + wex.getMessage());
154.      wex.printStackTrace();
155.    }
156.  }
157.
158.  private static String formatXML(OMElement omInput) {
159.    ByteArrayOutputStream out = new ByteArrayOutputStream();
160.    String output = "";
161.
162.    try {
163.          XMLPrettyPrinter.prettify(omInput, out);
164.          output = out.toString();
165.    } catch (Throwable e) {
166.        try {
167.           output = omInput.toString();
168.        } catch (Throwable e2) {
169.               System.out.println("Caught exception: " + e2.getMessage());
170.        e2.printStackTrace();
171.        }
172.        }
173.    return output;
174.  }
175.
176.  }
```

WSSTrustClient class support for policy sets and bindings

The WS-Trust client function supports both application-specific bindings and general bindings for use with the trust client policy set and binding documents. In addition, general bindings and default bindings are supported if the application is running in the application server environment. General bindings are supported in the thin client environment, but default bindings are not.

Managing the policy set and bindings for the WS-Trust client API is similar to managing a policy set and bindings for a web services client. However, differences exist that are unique to the WS-Trust client. One difference is that the WS-Trust client does not use policy set attachments. Instead, the policy set name and binding name are specified in a ProviderConfig object, as shown in line 36 and line 37 of the sample code.

When the WS-Trust client looks for bindings, the way the client manages the search scope differs from the web services client. If you do not specify the wstrustClientBindingScope property for the trust client binding, the system first searches the application for an application-specific binding with the binding name that you specified. If a binding is found, it is used for the trust client request. If no application-specific binding is found, the system searches the available general bindings for a binding with the name that you specified. If a general binding is found, it is used for the trust client request. If no bindings with the specific name are found, then default bindings are used in a server environment. Default bindings are only used in a server environment. If the binding scope is specified, only that scope is used for the binding search.

Line 38 of the sample code, `providerConfig.setBindingScope("domain")`, indicates that the example uses general bindings. You can also set the binding scope to `application` to indicate that the sample code uses application-specific bindings. The example uses the general binding named **SamlTCSample**. Both application-specific and general bindings are supported in the application server and the thin client environment. For more information about configuring the SamlTCSample bindings when the application is installed on the application server, read about configuring policy sets and bindings to communicate with STS.

The showProviderConfigDefaultValue(providerConfig) code on line 34 of the sample code shows the default settings. The sample code includes a utility method that prints out the contents of providerConfig.

Line 51 of the sample code, List<SecurityToken> securityTokens = client.issue(providerConfig, requesterConfig), sends an issue WS-Trust request. The second parameter on this line specifies the RequesterConfig object, and this parameter determines the content of the issue request. The code on line 41, RequesterConfig requesterConfig = WSSTrustClient.newRequesterConfig(Namespace.WST13), instantiates a RequesterConfig object that is used to construct the trust request using the WS-Trust Version 1.3 namespace. A utility function is shown on line 43: showRequestConfigDefaultValue(requesterConfig). This function displays the default settings for the RequesterConfig object. The code between lines 45 and 48 initializes the RequesterConfig to request a Version 1.1 SAML bearer token. This token is used to access the service endpoint using the SOAP 1.2 namespace. In the example, the service endpoint is https://user.MyCompany.com:9443/WSSampleSei/EchoService12.

JVM arguments support

Before executing the sample code, you must set up several Java Virtual Machine (JVM) arguments. The sample code implements the Username WSHTTPS default policy set, which has two requirements: 1) a Username token is sent to the STS; and 2) messages are protected using Secure Sockets Layer (SSL). To set up the environment to meet these requirements, first configure the `ssl.client.props` file to define a truststore. For step-by-step instructions, read about running an unmanaged web services JAX-WS client.

To meet the second requirement regarding SSL message protection, obtain a copy of the STS SSL X.509 certificate and inset it into the truststore. To do this, follow the steps in the topic, Using the retrieveSigners command in SSL, to enable server-to-server trust. Alternately, you can accept the STS certificate when

you send the first trust request to the STS if the com.ibm.ssl.enableSignerExchangePrompt property in the profile_home/properties/ssl.client.props file is set to true. For more information about this option, read about changing the signer auto-exchange prompt at the client.

In addition, you must specify the client JAAS configuration file so that the client runtime environment can locate the Username token LoginModule JAAS login configuration. Specify the parameter using this code: -Djava.security.auth.login.config="%WAS_HOME%\properties\wsjaas_client.conf. You must also include the thin client jar, for example com.ibm.jaxws.thinclient_8.0.0.jar, in the classpath. For more information, read about running an unmanaged web services JAX-WS client application.

Sample code execution

A prerequisite to executing the sample code is to set up an external STS endpoint to issue a SAML 1.1 bearer token for the specified web services as defined by the RequesterConfiguration.RSTT.APPLIESTO_ADDRESS property.

Executing the sample code generates a WS-Trust issue request message, as shown in the example:

```
177.  <?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
178.      <soapenv:Header>
179.          <wsa:To xmlns:wsa="http://www.w3.org/2005/08/addressing">https://user.MyCompany.com/Trust/13/UsernameMixed</wsa:To>
180.          <wsa:MessageID xmlns:wsa="http://www.w3.org/2005/08/addressing">urn:uuid:4951B6775950CAC92A1252458259166</wsa:MessageID>
181.          <wsa:Action xmlns:wsa="http://www.w3.org/2005/08/addressing">http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue</wsa:Action>
182.      </soapenv:Header>
183.      <soapenv:Body>
184.          <wst:RequestSecurityToken xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
185.              <wst:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1</wst:TokenType>
186.              <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue</wst:RequestType>
187.              <wst:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer</wst:KeyType>
188.              <wsp:AppliesTo xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
189.                  <wsa:EndpointReference xmlns:wsa="http://www.w3.org/2005/08/addressing">
190.                      <wsa:Address>https://user.MyCompany.com:9443/WSSampleSei/EchoService12</wsa:Address>
191.                  </wsa:EndpointReference>
192.              </wsp:AppliesTo>
193.          </wst:RequestSecurityToken>
194.      </soapenv:Body>
195.  </soapenv:Envelope>
```

To view the WS-Trust request message, you must enable a client-side trace. Set the following JVM properties:

- -DtraceSettingsFile=MyTraceSettings.properties
- -Djava.util.logging.manager=com.ibm.ws.bootstrap.WsLogManager
- -Djava.util.logging.configureByServer=true

For more information about these properties, read about enabling trace on client and stand-alone applications. In addition to setting the JVM properties, you must also specify the trace setting, com.ibm.ws.wssecurity.*=all=enabled, in the MyTraceSettings.properties file. Look for Trust Client outgoing request: in the trace log file.

SAML token return

The code on line 51 of the sample code, List<SecurityToken> securityTokens = client.issue(providerConfig, requesterConfig), returns a SAML token if the WS-Trust issue request is processed successfully. The code between lines 54 and 64 processes the returned SAML token. The utility function shown on line 58, showSAMLToken((SAMLToken)token), displays the content of the received SAML token. The showSAMLToken() routine shows the SAML token as an XML document. An example of this XML document is provided in line 196 to line 233 of the sample code.

```
196.  <?xml version="1.0" encoding="UTF-8"?>
197.  <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion" MajorVersion="1" MinorVersion="1"
198.   AssertionID="_f7f65d28-fbb1-4e10-8ddf-f4b6ed0c8277" Issuer="http://MyCompany.com/Trust"
199.   IssueInstant="2009-09-09T01:04:41.144Z">
200.      <saml:Conditions NotBefore="2009-09-09T01:04:41.141Z" NotOnOrAfter="2009-09-09T11:04:41.141Z">
201.          <saml:AudienceRestrictionCondition>
202.              <saml:Audience>https://user.MyCompany.com:9443/WSSampleSei/EchoService12</saml:Audience>
203.          </saml:AudienceRestrictionCondition>
204.      </saml:Conditions>
205.      <saml:AuthenticationStatement AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
206.       AuthenticationInstant="2009-09-09T01:04:41.131Z">
207.          <saml:Subject>
208.              <saml:SubjectConfirmation>
```

```
209.                 <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:bearer</saml:ConfirmationMethod>
210.             </saml:SubjectConfirmation>
211.         </saml:Subject>
212.     </saml:AuthenticationStatement>
213.     <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
214.         <ds:SignedInfo>
215.             <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
216.             <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
217.             <ds:Reference URI="#_f7f65d28-fbb1-4e10-8ddf-f4b6ed0c8277">
218.                 <ds:Transforms>
219.                     <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
220.                     <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
221.                 </ds:Transforms>
222.                 <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
223.                 <ds:DigestValue>AQ6e7YQqKgcg/B/ebBj8/DF+uWg=</ds:DigestValue>
224.             </ds:Reference>
225.         </ds:SignedInfo>
226.         <ds:SignatureValue>SuccIOniR . . . . yjTh9iQs=</ds:SignatureValue>
227.         <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
228.             <X509Data>
229.                 <X509Certificate>MIIB3zCCAUi . . . . itzymqg3</X509Certificate>
230.             </X509Data>
231.         </KeyInfo>
232.     </ds:Signature>
233. </saml:Assertion>
```

SAML token library APIs:

The SAML token library application programming interfaces (APIs) provide methods you can use to create, validate, parse, and extract SAML tokens.

Overview

The library implementation for SAML Version 1.1 and SAML Version 2.0 provides three types of subject confirmation: holder-of-key (HoK), bearer, and sender-vouches. You can use the SAML token library APIs to create, validate, and extract the attributes of a SAML HoK or bearer token. SAML token propagation from web services SOAP messages is also discussed. Sample code is provided to demonstrate the use of the APIs.

WebSphere Application Server with SAML provides default policy sets to support the bearer and HoK subject confirmation.

These sections discuss creating a SAML token using the SAML token library APIs:

1. “Configuration of token creation parameters” on page 392
2. “SAML token factory instance creation” on page 393
3. “SAML token creation” on page 393
4. “Sample code” on page 394

The SAMLTokenFactory API is the primary SAML token library programming interface. SAMLTokenFactory supports creating, parsing, and validating both SAML 1.1 and SAML 2.0 tokens. Using the SAMLTokenFactory API, you can create ProviderConfid, RequesterConfig, and ConsumerConfig configuration objects to define the required SAML token characteristics. Read the API documentation for more details.

You can perform additional operations on a SAML token after it is created, including:

- “SAML token validation” on page 393
- “SAML token identity mapped to a subject” on page 393
- “Parse assertion elements” on page 394
- “SAML token attributes extraction” on page 394

Note: Starting with WebSphere Application Server Release 8, you can use the `com.ibm.websphere.wssecurity.wssapi.token.SAMLToken` class in Web Services Security (WSS) application programming interface (API). When there is no concern of confusion we use the term `SAMLToken` instead of using its complete package name. You can use WSS API to request

SAMLToken processing from an external Security Token Service (STS), to propagate SAMLTokens in SOAP request messages, and to use a symmetric or asymmetric key identified by SAMLTokens to protect SOAP messages.

The WSS API SAML support complements the `com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory` and `com.ibm.websphere.wssecurity.wssapi.trust.WSSTrustClient` interfaces. SAMLTokens that are generated using the `com.ibm.websphere.wssecurity.wssapi.WSSFactory newSecurityToken()` method can be processed by the `SAMLTokenFactory` and `WSSTrustClient` programming interfaces. Conversely, SAMLTokens that are generated by `SAMLTokenFactory` or returned by `WSSTrustClient` can be used in WSS API. Deciding which API to use in your application depends on your specific needs. WSS API SAML support is self contained in the sense that it provides functionality equivalent to that of the `SAMLTokenFactory` and `WSSTrustClient` interfaces as far as web services client applications are concerned. The `SAMLTokenFactory` interface has additional functions to validate SAMLTokens and to create the JAAS Subject that represents authenticated SAMLTokens. This validation is useful for the Web services provider side. When you develop applications to consume SAMLTokens, the `SAMLTokenFactory` programming interface is more suitable for you.

Configuration of token creation parameters

When you configure the token creation parameters, the configuration information relates to either the requesting entity, the issuing entity, or the receiving entity. In this example, configuration information is defined for the requesting and the issuing entities. For each type of supported subject confirmation, the SAML token library provides pre-configured attributes for the requesting entity. These attributes are used during the creation of the self-issued SAML token by the WebSphere runtime environment. A self-issued SAML token is one that is generated locally, instead of one that is requested from a Security Token Service (STS). If you need to customize the attributes for a default parameter, use the RequesterConfig parameter. For more information, read about the RequesterConfig parameter in the SAMLTokenFactory API topic.

First, set up the requestor configuration information:

```
// Setup the requester's configuration information (parameters needed
// to create the token specified as configuration properties).
// in this case we are using the configuration information to create a
// SAML token that contains a symmetric holder of key subject
// confirmation.
RequesterConfig requesterData =
     samlFactory.newSymmetricHolderOfKeyTokenGenerateConfig();
```

Next, set the recipient public key alias and optionally, the authentication method:

```
// Set recipient's public key alias
// (in this example we use SOAPRecipient), so the provider can encrypt secret
// key for the receiving end.
requesterData.setKeyAliasForAppliesTo("SOAPRecipient");

// Set the authentication method that took place. This is an optional
// parameter.
reqData.setAuthenticationMethod("Password");
```

Then set the issuer configuration attributes:

```
// Set issuer information by instantiating a default ProviderConfig.
// See javadocs for the SAMLTokenFactory class on the details of the
// default values and how to modify them.
ProviderConfig samlIssuerCfg =
  samlFactory.newDefaultProviderConfig("WebSphereSelfIssuer");
```

SAML token factory instance creation

Use the SAMLTokenFactory class, specifying the SAML token type, either Version 1.1 or Version 2.0. Set additional parameters for creating the SAML token.

Use the SAMLTokenFactory class with the SAML token type:

```
// Instantiate a token factory based on the version level of the token
// to use. In this example we use the SAML v1.1 token factory.
SAMLTokenFactory samlFactory =
SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);
```

Set additional parameters in the CredentialConfig object using the caller subject or the runAsSubject:

```
// Retrieve the caller subject or the runAsSubject (depending on your
// scenario) then use the Subject to get a CredentialConfig object
// using the SAML token library.
// This invocation requires the
// wssapi.SAMLTokenFactory.newCredentialConfig" Java Security
// permission.
CredentialConfig cred = samlFactory.newCredentialConfig(runAsSubject);
```

SAML token creation

Create the SAML token using the token factory:

```
// Now create the SAML token. This invocation requires the
// "wssapi.SAMLTokenFactory.newSAMLToken" Java Security permission.
SecurityToken samlToken =
                samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

SAML token validation

An entity that receives a SAML token, such as a business service, can use the SAML token library API to validate the token before using it. For example, the service needs to validate the token before extracting the SAML attributes from the requester. An existing SAML assertion document can be validated using the configuration data from the consumer.

The following API code validates the token:

```
ConsumerConfig consumerConfig = samlFactory.newConsumerConfig();
 XMLStructure xml =
 try {
    SAMLToken token = samlFactory.newSAMLToken( consumerConfig, XMLStructure xml );
    // token successfully validated
 } catch(WSSException e){
  // token failed validation
 }
```

SAML token identity mapped to a subject

A SAML token can be used to create a subject. The name identifier in the SAML token is mapped to a user in the user registry to obtain the principal name for the subject.

```
Subject subject;
SAMLToken aSAMLToken = ...;
  try {
        subject = samlFactory.newSubject(aSAMLToken);
  } catch(WSSException e) {
  }
```

Parse assertion elements

The recipient of a SAML token can parse and extract assertion elements from the SAML token using the SAMLToken APIs, which are included in the SAML token library API. For example, the token creation time can be extracted using this code:

```
Date dateCreated = samlToken.getSamlCreated();
```

Extract the name of the token issuer and the confirmation method as follows:

```
String confirmationMethpo = samlToken.getConfirmationMethod();
String issuerName = samlToken.getSAMLIssuerName();
```

If the extracted subject confirmation method is returned as holder-of-key confirmation, then you can use the following API to retrieve the bytes for the key material:

```
byte[] hokBytes = samlToken.getHolderOfKeyBytes();
```

For more information about all the SAML APIs, read the API documentation for the SAMLToken interface.

SAML token attributes extraction

Extract SAML attributes from the initiating entity (service requester) using the SAMLToken API, as shown in the following code snippets.

```
// Get all attributes
List<SAMLAttribute> allAttributes =
             ((SAMLToken) samlToken).getSAMLAttributes();

// Iterate over the attribute and process accordingly
Iterator<SAMLAttribute> iter = allAttributes.iterator();
while (iter.hasNext())
{
    SAMLAttribute anAttribute = iter.next();

    // Handle attributes
    String attributeName = anAttribute.getName();
    String[] attributeValues = anAttribute.getStringAttributeValue();

}
```

Sample code

The sample code demonstrates how to use the SAML token library APIs to accomplish some of the operations previously described. A JVM property that points to the location of the SAML properties file is a prerequisite for running this code. The SAML properties file, `SAMLIssuerConfig.properties`, must contain configuration attributes related to the issuer (provider) of the SAML token.

The default location of the SAMLIssuerConfig.properties file for the cell level is: *app_server_root*/profiles/*$PROFILE*/config/cells/*$CELLNAME*/sts.

For the server level, the default location is: *app_server_root*/profiles/*$PROFILE*/config/cells/*$CELLNAME*/nodes/*$NODENAME*/servers/*$SERVERNAME*.

```
IssuerURI=WebSphere
TimeToLive=3600000
KeyStorePath=c:/samlsample/saml-provider.jceks
KeyStoreType=jceks
KeyStorePassword=myissuerstorepass
KeyAlias=samlissuer
KeyName=CN=SAMLIssuer, O=IBM, C=US
KeyPassword=xxxxxxxxx
TrustStorePath=c:/samlsample/saml-provider.jceks
TrustStoreType=jceks
TrustStorePassword=yyyyyyyy

package samlsample;
```

```
import java.util.List;
import java.util.Iterator;
import java.util.ArrayList;
import javax.security.auth.Subject;

// Import methods from the SAML token library
import com.ibm.wsspi.wssecurity.saml.data.SAMLAttribute;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.wsspi.wssecurity.saml.config.ProviderConfig;
import com.ibm.wsspi.wssecurity.saml.config.RequesterConfig;
import com.ibm.wsspi.wssecurity.saml.config.CredentialConfig;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.RequesterConfiguration;

public class SamlAPIsample {

public void testSAMLTokenLibrary() throws Exception {

try {
  // Get an instance of the SAML v1.1 token factory
  SAMLTokenFactory samlFactory = SAMLTokenFac
  tory.getInstance(SAMLTokenFactory.WssSamlV11Token11);

  // Generate default requester data for a subject confirmation of
  // type holder-of-key (secret key).
  RequesterConfig requesterData =
       samlFactory.newSymmetricHolderOfKeyTokenGenerateConfig();

  // Set the recipient's key alias, so that the issuer can encrypt
  // the secret key for recipient as part of the subject confirmation.
  requesterData.setKeyAliasForAppliesTo("SOAPRecipient");

  // Set the authentication method that took place.
  requesterData.setAuthenticationMethod("Password");

  System.out.println("default holder of key confirmation key type is: "+
    Requester
  Data.getRSTTProperties().get(RequesterConfiguration.RSTT.KEYTYPE));
  Requester
  Data.getRSTTProperties().put(RequesterConfiguration.RSTT.KEYTYPE,
      "http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey");

  requesterData.getRSTTProperties().put(
                    RequesterConfiguration.RSTT.APPLIESTO_ADDRESS,

                    "http://localhost:9080");

  requesterData.setConfirmationMethod("holder-of-key");

  // Set the recipient's key alias so that token infomation such as
  // the secret HoK can be encrypted by the issuer and decrypted by the
  // recipient.
  requesterData.setKeyAliasForAppliesTo("SOAPRecipient");
  requesterData.setAuthenticationMethod("Password");
  requesterData.getRSTTProperties().put(
  RequesterConfiguration.RSTT.ENCRYPTIONALGORITHM,
                     "http://www.w3.org/2001/04/xmlenc#aes128-cbc");
  requester
  Data.getRSTTProperties().put(RequesterConfiguration.RSTT.TOKENTYPE,
      "http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
       1.1#SAMLV1.1");
  requesterData.setRequesterIPAddress("9.53.52.65");

  // Print requester configuration items
  System.out.println("authentication method for requester is: "+
             requesterData.getAuthenticationMethod());
  System.out.println("confirmation method for requester is: "+
                    requesterData.getConfirmationMethod());
  System.out.println("key alias for requester is: "+
            requesterData.getKeyAliasForRequester());
  System.out.println("key alias for recipient as set in requester config is "+
          requesterData.getKeyAliasForAppliesTo());
  System.out.println("holder of key confirmation key type is: "+
             Requester
  Data.getRSTTProperties().get(RequesterConfiguration.RSTT.KEYTYPE));

  // Get an instance of the Credential config object
  CredentialConfig cred = samlFactory.newCredentialConfig();
  cred.setRequesterNameID("Alice");

  // Set some user attributes
  ArrayList<SAMLAttribute> userAttrs = new ArrayList<SAMLAttribute>();
  SAMLAttribute anAttribute = new SAMLAttribute("EmployeeInfo",
      new String[] {"GreenRoofing","JohnDoe", "19XY981245"},
                null, "WebSphere Namespace", null, "JohnDoeInfo " );
  userAttrs.add(anAttribute);
  cred.setSAMLAttributes(userAttrs);

  // Get default provider configuration
```

```
  ProviderConfig samlIssuerCfg =
    samlFactory.newDefaultProviderConfig("WebSphereSelfIssuer");
  System.out.println("time to live from the default provider config: "+
            samlIssuerCfg.getTimeToLive());
  System.out.println("keyStore path from default provider config: "+
                  samlIssuerCfg.getKeyStoreConfig().getPath());
  System.out.println("keyStore type from default provider config: "+
                 samlIssuerCfg.getKeyStoreConfig().getType());
  System.out.println("key alias from default provider config: "+
                 samlIssuerCfg.getKeyInformationConfig().getAlias());

  // Generate the SAML token
  SecurityToken samlToken =
        samlFactory.newSAMLToken(cred, requesterData, samlIssuerCfg);
  System.out.println("token's creation Date is:
              "+((SAMLToken)samlToken).getSamlCreated().toString());
  System.out.println("token's expiration Date is:
             "+((SAMLToken)samlToken).getSamlExpires().toString());
  System.out.println("token's subject confirmation method is:
                "+((SAMLToken)samlToken).getConfirmationMethod());


  // Create a Subject, mapping the name identifier in the token to a user
  // in the user registry to obtain the Principal name
  Subject subject = samlFactory.newSubject((SAMLToken)samlToken);

  // Retrieve attributes from the token
  List<SAMLAttribute> allAttributes =
                      ((SAMLToken)samlToken).getSAMLAttributes();

  // Iterate through the attributes and process accordingly
  Iterator<SAMLAttribute> iter = allAttributes.iterator();
  while (iter.hasNext()) {
   SAMLAttribute attribute = iter.next();
   String attributeName = attribute.getName();
   String[] attributeValues = attribute.getStringAttributeValue();
   System.out.println("attribute name = "+ attributeName +
                     "  attribute value = ["+
                     attributeValues[0]+ ",
                     "+attributeValues[1]+ ", "+
                     attributeValues[2]+"]");
  }
  } catch(Exception e) {
    e.printStackTrace();
  }
 }
}
```

Sample code output

```
default holder of key confirmation key type is: http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey
authentication method for requester is: Password
confirmation method for requester is: holder-of-key
key alias for requester is: null
key alias for recipient as set in requester config is SOAPRecipient
holder of key confirmation key type is: http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey
time to live from the default provider config: 3600000
keyStore path from default provider config: C:/saml/samlsample/saml-provider.jceks
keyStore type from default provider config: jceks
key alias from default provider config: samlissuer
token's creation Date is: Mon Sep 14 15:49:00 CDT 2009
token's expiration Date is: Mon Sep 14 16:49:00 CDT 2009
token's subject confirmation method is: urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
attribute name = EmployeeInfo  attribute value = [GreenRoofing, JohnDoe, 19XY981245]
```

Creating a SAML bearer token using the API:

Use the SAML library API to create a SAML bearer token.

About this task

This library allows you to create a SAML bearer token. You can use the SAML library API to create required SAML configuration objects, then use those configuration objects to generate a bearer SAML token.

Procedure

1. Create a SAMLTokenFactory instance using the SAML token version as a parameter.
   a. Use the following line of code to import the method:

```
import com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory;
```

b. Use one of these lines of code to create the instance, depending on the token version.
   - Add the following line of code to create a SAMLTokenFactory instance for a version 1.1 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);
```

   - Add the following line of code to create a SAMLTokenFactory instance for a version 2.0 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token20);
```

2. After you create the instance, the SAMLTokenFactory is used to create a RequesterConfig instance, which determines how the token will be generated, according the authentication requirements of the requester. Use this line of code to create the RequesterConfig instance for the bearer token:

```
RequesterConfig reqData = samlFactory.newBearerTokenGenerateConfig();
```

   The default RequestConfig instance is sufficient to generate a simple bearer token, but additional assertions can be included in the SAML token by customizing the RequesterConfig instance. For example, to include password authentication information in the token, use setAuthenticationMethod:

```
reqData.setAuthenticationMethod("password");
```

   To disable signing in a SAML assertion, use the setAssertionSignatureRequired method, for example:

```
reqData.setAssertionSignatureRequired(false);
```

   For more information, read about the SAML bearer assertion.

3. Use the SAMLTokenFactory to create a ProviderConfig instance, which describes the token issuer. The ProviderConfig instance specifies the SAML issuer name, as well as keystore and truststore information, which identifies the key for SAML encryption and signing. The ProviderConfig instance is created using property values from a property file. The property file specifies the default value of the ProviderConfig object. In a Java client environment, this property file is defined by a JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath.

   In the WebSphere Application Server runtime environment, the property file name is SAMLIssuerConfig.properties. The file can be located either under the server level configuration directory, or the cell level directory, in that order of precedence. An example of the server level path follows:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/nodes/$NODENAME/servers/$SERVERNAME/SAMLIssuerConfig.properties
```

   An example of the cell level path follows:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/sts/SAMLIssuerConfig.properties
```

   The JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath, is ignored if the property is defined in server runtime environment. For a detailed description of all the properties, read about configuration of a SAML token during token creation.

   Use the following line of code to create a default ProviderConfig instance:

```
ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("any issuer name");
```

   The issuer name is optional. If the issuer name is specified, the Issuer name appears in the SAML assertion. If the issuer name is not specified, a default issuer name property from the SAMLIssuerConfig.properties is used as the issuer name.

4. Optional: When creating a new SAML token, the SAMLTokenFactory uses either a JAAS subject or a CredentialConfig instance to populate the new SAML token. To use a JAAS subject to populate the token, use the com.ibm.websphere.security.auth.WSSubject getCallerSubject() API or the getRunAsSubject() API to obtain a JAAS subject that represents the requesting client, or the identity of the execution thread.

   When you use the JAAS subject to create a new SAML token, the SAMLTokenFactory searches for a SAMLToken object in the subject PrivateCredentials list. If a SAMLToken object exists, the NameId or NameIdentifier are copied to the new SAML token. The SAMLTokenFactory also copies the SAML attributes and AuthenticationMethod from the existing SAML token to the new SAML token. The new

SAML token includes a new issuer name, new signing certificate, confirmation method, new KeyInfo for the holder-of-key confirmation method, and new NotBefore and NotOnAfter conditions. These token settings are determined by the configuration parameters in the ProviderConfig and RequesterConfig objects.

If there is no SAMLToken object in the subject, only the WSPrincipal principal name is copied from the subject to the new SAML token. No other attributes in the subject are copied into the new SAML token. Similarly, the issuer name, signing certificate, confirmation method, KeyInfo for the holder-of-key, and NotBefore and NotOnOrAfter conditions are determined by configuration parameters in ProviderConfig and RequesterConfig objects.

Alternately, you can use the RunAsSubject method on the execution thread to create the SAML token. When using this method, do not pass the JAAS subject or the CredentialConfig object to the SAMLTokenFactory to create the SAML token. Instead, the content of the existing SAML token is copied to the new SAML token, as described previously.

Another method of creating a SAML token is to use a CredentialConfig object to populate the SAML NameId and Attributes programmatically. Use this method in the following circumstances:

- Custom SAML attributes must be included in the new SAML token.
- The SAML token is created manually instead of using the SAMLTokenFactory to populate the SAML token from a JAAS subject automatically.
- There is no existing SAML token in the subject.
- There is no JAAS subject available.

To create a CredentialConfig object without using the JAAS subject, use this line of code:

```
CredentialConfig cred = samlFactory.newCredentialConfig ();
```

There is no initial value provided for this CredentialConfig object, so you must use setter methods to populate the CredentialConfig object.

To populate the SAML NameIdentifier or NameID, use the following line of code:

```
cred.setRequesterNameID("any name");
```

The value of the *any name* variable is used as the principal name in the SAML token. The name appears in the assertion as the NameIdentifier in a SAML Version 1.1 token, or NameId in the SAML Version 2.0 token. For example, if the value of *any name* is `Alice`, the following assertion is generated in a SAML Version 1.1 token:

```
<saml:NameIdentifier>Alice</saml:NameIdentifier>
```

The following assertion is generated in a SAML Version 2.0 token:

```
<saml2:NameID>Alice</saml2:NameID>
```

To include SAML attributes in the <AttributeStatement> portion of an assertion, use this code:

```
SAMLAttribute samlAttribute = new SAMLAttribute("email" /* Name*/, new String[] {"joe@websphere"}
/*Attribute Values*/, null, "IBM WebSphere namespace" /* namespace*/, "email" /* format*/, "joe" /*friendly name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(samlAttribute)
sattribute = new SAMLAttribute("Membership", new String[] {"Super users", "Gold membership"}, null, null /* format*/, null, null );
al.add(samlAttribute );
cred.setSAMLAttributes(al);
```

This sample code generates the following <Attribute> assertions:

```
<saml:Attribute AttributeName="email" NameFormat="email" AttributeNamespace="IBM WebSphere namespace">
<saml:AttributeValue>joe@websphere</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute AttributeName="Membership">
<saml:AttributeValue>Super users</saml:AttributeValue><saml:AttributeValue>Gold membership</saml:AttributeValue>
</saml:Attribute>
```

5. Generate a SAML bearer token using this line of code:

```
SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

This method requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken.

Complete code samples using lines of code from the previous steps are included in the Example section.

Example

Use this sample code to create a SAML version 1.1 bearer token from the subject:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11)

RequesterConfig reqData = samlFactory.newBearerTokenGenerateConfig();

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("WebSphere Server");

Subject subject =  com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();

SAMLToken samlToken = samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Use this sample code to create a SAML version 1.1 bearer token without using the subject:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);

RequesterConfig reqData = samlFactory.newBearerTokenGenerateConfig();
reqData.setAuthenticationMethod("Password"); //Authentication method for Assertion

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig(Self issuer);

CredentialConfig cred = samlFactory.newCredentialConfig ();
cred.setRequesterNameID("Alice");   // SAML NameIdentifier
//SAML attributes:
SAMLAttribute attribute = new SAMLAttribute
   ("email" /* Name*/, new String[] {"joe@websphere"}
   /*Attribute Values in String*/,null
   /*Attribute Values in XML */, "WebSphere" /* Namespace*/, "email" /* format*/, "joe" /*Friendly_name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(attribute);
attribute = new SAMLAttribute("Membership", new String[] {"Super users", "My team"}, null, null, null, null  );
al.add(attribute);
cred.setSAMLAttributes(al);

SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

Creating a SAML holder-of-key token using the API:

The SAML holder-of-key token extends the security token public interface in WebSphere Application Server, and can be used as a protection token. WebSphere Application Server provides a SAML library API for SAML holder-of-key token creation.

About this task

SAML token creation requires three parameters:

- com.ibm.wsspi.wssecurity.saml.config.RequesterConfig
- com.ibm.wsspi.wssecurity.saml.config.ProviderConfig
- com.ibm.wsspi.wssecurity.saml.config.CredentialConfig

Follow the steps to create an instance for each of the parameters and then create a SAML holder-of-key token. As an alternative to CredentialConfig, you can also use javax.security.auth.Subject. For more information, read the API documentation.

Procedure

1. Create a com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory instance using the SAML token version as a parameter. The supported SAMLToken versions are http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1 and http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0. The SAMLTokenFactory instance is a singleton and therefore is thread-safe. Use one of these lines of code to create the instance, depending on the token version.
   - Use the following line of code to create a com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory instance for a version 1.1 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);
```

- Use the following line of code to create a com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory instance for a version 2.0 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token20);
```

2. The SAMLTokenFactory instance is used to create a RequesterConfig instance, which determines how the token is generated, according the authentication requirements of the requester. Use one of these lines of code to create the RequesterConfig instance, depending on whether you want the token to use a secret key (symmetric key) or a public key:
   - Use the following line of code to create a default RequesterConfig for the SAML holder-of-key token using a secret key (symmetric key), which is included in the SubjectConfirmation:

```
RequesterConfig reqData = samlFactory. newSymmetricHolderOfKeyTokenGenerateConfig ();
```

You must also set the key alias for the target service so the provider can encrypt the secret key for the service:

```
reqData.setKeyAliasForAppliesTo("Soap Recipient");
```

   - Use the following line of code to create a default RequesterConfig for the SAML holder-of-key token using a public key, which is included in the SubjectConfirmation:

```
RequesterConfig reqData = samlFactory. newAsymmetricHolderOfKeyTokenGenerateConfig ();
```

You must also set the key alias for the requester so the provider can extract the public key from the requester, and include the key in the SubjectConfirmation:

```
reqData.setKeyAliasForRequester("SOAP Initiator");
```

The default RequestConfig instance is sufficient to generate a simple holder-of-key token, but additional assertions can be included in the SAML token by customizing the RequesterConfig instance. For example, to include password authentication information in the token, use setAuthenticationMethod:

```
reqData.setAuthenticationMethod("password");
```

3. Use the SAMLTokenFactory to create a ProviderConfig instance, which describes the token issuer. The ProviderConfig instance specifies the SAML issuer name, as well as keystore and truststore information, which identifies the key for SAML encryption and signing. The ProviderConfig instance is created using property values from a property file. The property file specifies the default value of the ProviderConfig object. In a Java client environment, this property file is defined by a JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath.

   In the WebSphere Application Server runtime environment, the property file name is SAMLIssuerConfig.properties. The file can be located either under the server level configuration directory, or the cell level directory, in that order of precedence. An example of the server level path follows:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/nodes/$NODENAME/servers/$SERVERNAME/SAMLIssuerConfig.properties
```

   An example of the cell level path follows:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/sts/SAMLIssuerConfig.properties
```

   The JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath, is ignored if the property is defined in server runtime environment. For a detailed description of all the properties, read about configuration of a SAML token during token creation.

   Use the following line of code to create a default ProviderConfig instance:

```
ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("any issuer name");
```

   The issuer name is optional. If the issuer name is specified, the Issuer name appears in the SAML assertion. If the issuer name is not specified, a default issuer name property from the SAMLIssuerConfig.properties is used as the issuer name.

4. Optional: When creating a new SAML token, the SAMLTokenFactory uses either a JAAS subject or a CredentialConfig instance to populate the new SAML token. To use a JAAS subject to populate the

token, use the com.ibm.websphere.security.auth.WSSubject getCallerSubject() API or the getRunAsSubject() API to obtain a JAAS subject that represents the requesting client, or the identity of the execution thread.

When you use the JAAS subject to create a new SAML token, the SAMLTokenFactory searches for a SAMLToken object in the subject PrivateCredentials list. If a SAMLToken object exists, the NameId or NameIdentifier are copied to the new SAML token. The SAMLTokenFactory also copies the SAML attributes and AuthenticationMethod from the existing SAML token to the new SAML token. The new SAML token includes a new issuer name, new signing certificate, confirmation method, new KeyInfo for the holder-of-key confirmation method, and new NotBefore and NotOnAfter conditions. These token settings are determined by the configuration parameters in the ProviderConfig and RequesterConfig objects.

If there is no SAMLToken object in the subject, only the WSPrincipal principal name is copied from the subject to the new SAML token. No other attributes in the subject are copied into the new SAML token. Similarly, the issuer name, signing certificate, confirmation method, KeyInfo for the holder-of-key, and NotBefore and NotOnOrAfter conditions are determined by configuration parameters in ProviderConfig and RequesterConfig objects.

Alternately, you can use the RunAsSubject method on the execution thread to create the SAML token. When using this method, do not pass the JAAS subject or the CredentialConfig object to the SAMLTokenFactory to create the SAML token. Instead, the content of the existing SAML token is copied to the new SAML token, as described previously.

Another method of creating a SAML token is to use a CredentialConfig object to populate the SAML NameId and Attributes programmatically. Use this method in the following circumstances:

- Custom SAML attributes must be included in the new SAML token.
- The SAML token is created manually instead of using the SAMLTokenFactory to populate the SAML token from a JAAS subject automatically.
- There is no existing SAML token in the subject.
- There is no JAAS subject available.

To create a CredentialConfig object without using the JAAS subject, use this line of code:

```
CredentialConfig cred = samlFactory.newCredentialConfig ();
```

There is no initial value provided for this CredentialConfig object, so you must use setter methods to populate the CredentialConfig object.

To populate the SAML NameIdentifier or NameID, use the following line of code:

```
cred.setRequesterNameID("any name");
```

The value of the *any name* variable is used as the principal name in the SAML token. The name appears in the assertion as the NameIdentifier in a SAML Version 1.1 token, or NameId in the SAML Version 2.0 token. For example, if the value of *any name* is `Alice`, the following assertion is generated in a SAML Version 1.1 token:

```
<saml:NameIdentifier>Alice</saml:NameIdentifier>
```

The following assertion is generated in a SAML Version 2.0 token:

```
<saml2:NameID>Alice</saml2:NameID>
```

To include SAML attributes in the <AttributeStatement> portion of an assertion, use this code:

```
SAMLAttribute samlAttribute = new SAMLAttribute("email" /* Name*/, new String[] {"joe@websphere"}
/*Attribute Values*/, null, "IBM WebSphere namespace" /* namespace*/, "email" /* format*/, "joe" /*friendly name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(samlAttribute)
sattribute = new SAMLAttribute("Membership", new String[] {"Super users", "Gold membership"}, null, null /* format*/, null, null );
al.add(samlAttribute );
cred.setSAMLAttributes(al);
```

This sample code generates the following <Attribute> assertions:

```
<saml:Attribute AttributeName="email" NameFormat="email" AttributeNamespace="IBM WebSphere namespace">
<saml:AttributeValue>joe@websphere</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute AttributeName="Membership">
<saml:AttributeValue>Super users</saml:AttributeValue><saml:AttributeValue>Gold membership</saml:AttributeValue>
</saml:Attribute>
```

5. Generate a SAML holder-of-key token using this line of code:

```
SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

This method requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Complete code samples using lines of code from the previous steps are included in the Example section.

Example

Use this sample code to create a SAML version 1.1 holder-of-key token using a secret key (symmetric key) from the subject.

```
import com.ibm.wsspi.wssecurity.saml.config.RequesterConfig;
import com.ibm.wsspi.wssecurity.saml.config.ProviderConfig;
import com.ibm.wsspi.wssecurity.saml.config.CredentialConfoig ;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory

SAMLTokenFactory samlFactory =  SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);

RequesterConfig reqData  =  samlFactory.newSymmetricHolderOfKeyTokenGenerateConfig();

//Map "AppliesTo" to key alias, so library knows how to encrypt the Symmetric Key
reqData.setKeyAliasForAppliesTo("SOAPRecipient");

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig(IsserUri);

Subject subject =  com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();

SAMLToken samlToken = samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Use this sample code to create a SAML version 2.0 holder-of-key token using a public key from the subject:

```
//User expression on how SAML should be created, default provided
RequesterConfig reqData =  samlFactory.newAsymmetricHolderOfKeyTokenGenerateConfig();

//Choose a public key to be included in SAML
reqData.setKeyAliasForRequester("SOAPInitiator");

//Get issuer key store so can sign or encrypt assertion, issuer name
ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("any_issuer");

//Get JAAS Subject so the factory can populate principal and attributes to SAML
Subject subject =  com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();

SAMLToken samlToken = samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Use this sample code to create a SAML version 2.0 holder-of-key token using a secret key (symmetric key):

```
SAMLTokenFactory samlFactory =  SAMLTokenFactory.getInstance (SAMLTokenFactory.WssSamlV20Token11);

RequesterConfig reqData = samlFactory. newSymmetricHolderOfKeyTokenGenerateConfig ();
//Map "AppliesTo" to key alias so library knows how to encrypt the Symmetric Key
reqData.setKeyAliasForAppliesTo("SOAPRecipient");

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig(null);

CredentialConfig cred = samlFactory.newCredentialConfig ();
cred.setRequesterNameID("any_name");

SAMLToken samlToken =  samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Creating a SAML sender-vouches token using the API:

Use the SAML library API to create a SAML sender-vouches token, which includes the sender-vouches confirmation method. The sender-vouches confirmation method is used when a server needs to propagate the client identity or behavior of the client.

About this task

When SAML function is installed on a WebSphere server, a SAML library API is provided. Use the library to create a SAML sender-vouches token. You can use the SAML library API to create required SAML configuration objects. Then, use those configuration objects to generate a SAML sender-vouches token.

Procedure

1. Create a SAMLTokenFactory instance using the SAML token version as a parameter.
   a. Use the following line of code to import the method:

```
import com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory;
```

   b. Use one of these lines of code to create the instance, depending on the token version.
      - Add the following line of code to create a SAMLTokenFactory instance for a version 1.1 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);
```

      - Add the following line of code to create a SAMLTokenFactory instance for a version 2.0 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token20);
```

2. After you create the instance, the SAMLTokenFactory is used to create a RequesterConfig instance, which determines how the token will be generated, according the authentication requirements of the requester. Use this line of code to create the RequesterConfig instance for the sender-vouches token:

```
RequesterConfig reqData = samlFactory.newSenderVouchesTokenGenerateConfig();
```

   The default RequestConfig instance is sufficient to generate a simple sender-vouches token, but additional assertions can be included in the SAML token by customizing the RequesterConfig instance. For example, to include password authentication information in the token, use the method, setAuthenticationMethod:

```
reqData.setAuthenticationMethod("password");
```

   The trust validation for a sender-vouches assertion is the responsibility of the sender, not the issuer, so the Enveloped-Signature element is not required in the assertion. To remove the Enveloped-Signature element from the SAML assertion, use the setAssertionSignatureRequired method; for example:

```
reqData.setAssertionSignatureRequired(false);
```

3. Use the SAMLTokenFactory API to create a ProviderConfig instance, which describes the token issuer. The ProviderConfig instance specifies the SAML issuer name, as well as keystore and truststore information, which identifies the key for SAML encryption and signing. The ProviderConfig instance is created using property values from a property file. The property file specifies the default value of the ProviderConfig object. In a Java client environment, this property file is defined by a JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath.

   In the WebSphere Application Server runtime environment, the property file name is SAMLIssuerConfig.properties. The file can be located either under the server level configuration directory, or the cell level directory, in that order of precedence. See the following example of the server-level path:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/nodes/$NODENAME/servers/$SERVERNAME/SAMLIssuerConfig.properties
```

   See the following example of the cell-level path:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/sts/SAMLIssuerConfig.properties
```

   The JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath, is ignored if the property is defined in server runtime environment. For a detailed description of all the properties, read about configuration of a SAML token during token creation.

   Use the following line of code to create a default ProviderConfig instance:

```
ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("any issuer name");
```

The issuer name is optional. If the issuer name is specified, the Issuer name appears in the SAML assertion. If the issuer name is not specified, a default issuer name property from the SAMLIssuerConfig.properties is used as the issuer name.

4. Optional: When creating a new SAML token, the SAMLTokenFactory uses either a Java Authentication and Authorization Service (JAAS) subject or a CredentialConfig instance to populate the new SAML token. To use a JAAS subject to populate the token, use the com.ibm.websphere.security.auth.WSSubject getCallerSubject() API or the getRunAsSubject() API to obtain a JAAS subject that represents the requesting client, or the identity of the execution thread.
   - When you use the JAAS subject to create a new SAML token, the SAMLTokenFactory API searches for a SAMLToken object in the subject PrivateCredentials list. If a SAMLToken object exists, the NameId or NameIdentifier objects are copied to the new SAML token. The SAMLTokenFactory also copies the SAML attributes and AuthenticationMethod method from the existing SAML token to the new SAML token. The new SAML token includes a new issuer name, new signing certificate, confirmation method, new KeyInfo for the holder-of-key confirmation method, and new NotBefore and NotOnAfter conditions. These token settings are determined by the configuration parameters in the ProviderConfig and RequesterConfig objects.

     If there is no SAMLToken object in the subject, only the WSPrincipal principal name is copied from the subject to the new SAML token. No other attributes in the subject are copied into the new SAML token. Similarly, the issuer name, signing certificate, confirmation method, KeyInfo for the holder-of-key, and NotBefore and NotOnOrAfter conditions are determined by configuration parameters in ProviderConfig and RequesterConfig objects.

     Alternately, you can use the RunAsSubject method on the execution thread to create the SAML token. When using this method, do not pass the JAAS subject or the CredentialConfig object to the SAMLTokenFactory to create the SAML token. Instead, the content of the existing SAML token is copied to the new SAML token, as described previously.
   - Another method of creating a SAML token is to use a CredentialConfig object to populate the SAML NameId and Attributes programmatically. Use this method in the following circumstances:
     – Custom SAML attributes must be included in the new SAML token.
     – The SAML token is created manually instead of using the SAMLTokenFactory to populate the SAML token from a JAAS subject automatically.
     – There is no existing SAML token in the subject.
     – There is no JAAS subject available.

   a. To create a CredentialConfig object without using the JAAS subject, use this line of code:

```
CredentialConfig cred = samlFactory.newCredentialConfig ();
```

   There is no initial value provided for this CredentialConfig object, so you must use setter methods to populate the CredentialConfig object.

   b. To populate the SAML NameIdentifier or NameID, use the following line of code:

```
cred.setRequesterNameID("any name");
```

   The value of the parameter passed to the setRequesterNameID() method is used as the principal name in the SAML token. The name appears in the assertion as the NameIdentifier in a SAML Version 1.1 token, or NameId in the SAML Version 2.0 token. For example, if the value of the parameter passed to the setRequesterNameID() method is `Alice`, the following assertion is generated in a SAML Version 1.1 token:

```
<saml:NameIdentifier>Alice</saml:NameIdentifier>
```

   The following assertion is generated in a SAML Version 2.0 token:

```
<saml2:NameID>Alice</saml2:NameID>
```

   c. To include SAML attributes in the <AttributeStatement> portion of an assertion, use this code:

```
SAMLAttribute samlAttribute = new SAMLAttribute("email" /* Name*/, new String[] {"joe@websphere"}
/*Attribute Values*/, null, "IBM WebSphere namespace" /* namespace*/, "email" /* format*/, "joe" /*friendly name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(samlAttribute)
sattribute = new SAMLAttribute("Membership", new String[] {"Super users", "Gold membership"}, null, null /* format*/, null, null );
al.add(samlAttribute );
cred.setSAMLAttributes(al);
```

This sample code generates the following <Attribute> assertions:

```
<saml:Attribute AttributeName="email" NameFormat="email" AttributeNamespace="IBM WebSphere namespace">
<saml:AttributeValue>joe@websphere</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute AttributeName="Membership">
<saml:AttributeValue>Super users</saml:AttributeValue><saml:AttributeValue>Gold membership</saml:AttributeValue>
</saml:Attribute>
```

5. Generate a SAML sender-vouches token using this line of code:

```
SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

This method requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken.

Complete code samples using lines of code from the previous steps are included in the Example section.

Example

Use this sample code to create a SAML version 1.1 sender-vouches token from the subject:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11)

RequesterConfig reqData = samlFactory.newSenderVouchesTokenGenerateConfig();

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("WebSphere Server");

Subject subject =  com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();

SAMLToken samlToken = samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Use this sample code to create a SAML version 1.1 sender-vouches token without using the subject:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);

RequesterConfig reqData = samlFactory.newSenderVouchesTokenGenerateConfig();
reqData.setAuthenticationMethod("Password"); //Authentication method for Assertion

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig(Self issuer);

CredentialConfig cred = samlFactory.newCredentialConfig ();
cred.setRequesterNameID("Alice");   // SAML NameIdentifier
//SAML attributes:
SAMLAttribute attribute = new SAMLAttribute
   ("email" /* Name*/, new String[] {"joe@websphere"}
   /*Attribute Values in String*/,null
   /*Attribute Values in XML */, "WebSphere" /* Namespace*/, "email" /* format*/, "joe" /*Friendly_name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(attribute);
attribute = new SAMLAttribute("Membership", new String[] {"Super users", "My team"}, null, null, null, null  );
al.add(attribute);
cred.setSAMLAttributes(al);

SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

Propagation of SAML tokens using the API:

The SAML propagation function is useful for applications that interact across multiple servers. The propagation feature communicates token information from the originating server downstream to other servers.

You can propagate SAML tokens using administrative commands, or programmatically using the SAML application programming interface (API). Propagation through administrative commands is discussed in the topics Propagating SAML tokens and SAML token propagation methods.

Programmatic propagation of SAML tokens is achieved through a combination of explicit programming and use of the Web Services Security runtime environment. For example, you can extract the SAMLToken from the org.apache.axis2.jaxws.BindingProvider object. The token is then used for outbound calls. In this

example, since WebSphere security is not required, programmatically propagating the SAML token allows you to exploit SAML security at the application level. Furthermore, the SAML token can be communicated downstream using any protocol.

Use the following sample code to extract the SAMLToken on the client side after the first request is completed.

Create a Dispatch object and invoke the request:

```
javax.xml.ws.Dispatch dispatch = ...;
dispatch.invoke();
```

Obtain a response context and extract the SAMLToken:

```
Map<String, Object> responseContext = dispatch.getResponseContext();
   SAMLToken samlToken =
       (SAMLToken ) responseContext.get(com.ibm.wsspi.wssecurity.saml.config.SamlConstants.
    SAMLTOKEN_OUT_MESSAGECONTEXT);
```

The following sample code shows how to reuse a SAMLToken for subsequent web services requests.

The web services client program creates a dispatch instance to invoke a service:

```
javax.xml.ws.Dispatch dispatch = ...;
```

The web services client then uses this code to pass a SAMLToken to the Web Services Security handler:

```
Map<String, Object> requestContext = dispatch.getRequestContext();
     requestContext.put(com.ibm.wsspi.wssecurity.saml.config.SamlConstants.
     SAMLTOKEN_IN_MESSAGECONTEXT, samlToken);
```

The web services provider (receiver) can use the following code to extract a SAMLToken from an incoming web services request.

Extract a SAMLToken from the requestContext:

```
Subject subject = (Subject) context.get(com.ibm.wsspi.wssecurity.core.Constants.WSSECURITY_TOKEN_WSSSUBJECT);
 SAMLToken samlToken = null;
     try
     {
      samlToken = (SAMLToken) AccessController.doPrivileged(
        new java.security.PrivilegedExceptionAction() {
         public Object run() throws
             java.lang.Exception
         {
          final java.util.Iterator authIterator =
                subject.getPrivateCredentials(SAMLToken.class)
                .iterator();
          if ( authIterator.hasNext() ) {
           final SAMLToken token = (SAMLToken)
                  authIterator.next();

           return token;
          }
          return null;
         }
        });
 } catch (Exception ex) {
  // Error handling
```

Extract the SAML attributes:

```
List<SAMLAttribute> allAttributes;
allAttributes = ((SAMLToken) samlToken).getSAMLAttributes();
```

The web services client runtime environment can cache the SAML token. On subsequent client requests within the application, the security runtime environment retrieves the SAML token from the cache for use with the target.

Web services client token cache for SAML:

When a SAML token is initially requested, the web services runtime environment automatically caches the SAMLToken. As a result of this automatic client token caching function, subsequent web services requests can use the SAMLToken from the previous request.

The web services client token cache for SAML enables web services clients to reuse SAML tokens when accessing business services. Reusing valid SAML tokens reduces traffic to the Security Token Service (STS) and also reduces the performance impact of sending WS-Trust request messages. There are several requirements for a token to be considered valid and therefore available for caching and reuse.

In order for a SAML token to be reused, the expiration time of the token must be equivalent to, or greater than, the current time. A cache cushion is added to the current time when comparing the token expiration time with the current time so that the token does not expire immediately after it is sent.

In addition, a token is valid only if it is sent again to the same business service. The SAML function in WebSphere Application Server does not verify the AudienceRestriction condition for the SAML token. Therefore, a practical way to ensure that the SAML token is reused for the right audience is to reuse the token only for the same web service that originally used the token. If an assertion contains the OneTimeUse assertion, the SAML token is not cached.

To take advantage of the SAMLToken cache, the application and SAMLToken must meet the following requirements:

- The SAMLToken must have a relatively long expiration time, with at least 5 minutes remaining in the token lifetime after the first request is completed. The WS-Security runtime environment validates the cached SAMLToken expiration time against a predefined cache cushion. The cached token is valid only if the remaining token lifetime is greater than the cushion value. The default cushion value is 5 minutes. This value can be configured using the custom property, cacheCushion. To override the default cache cushion, edit the CallbackHandler custom property for the SAMLToken generator. Add the cacheCushion property and set the cache cushion value in milliseconds. If the cached SAMLToken lifetime is within the cache cushion limit, a new SAMLToken is requested. For example, you can change the cache cushion to 3 minutes or 180000 milliseconds.

Custom property name	Value
cacheCushion	180000

- The SAML token cannot contain the OneTimeUse assertion.
- If the SAML token is encrypted, make sure that the STS communicates the token expiration time outside the encrypted token, and that the SAML token does not include the OneTimeUse assertion.

When you do not want to reuse the same SAMLToken for subsequent requests, you can disable the client side SAMLToken cache with the cacheToken custom property. To disable the client side SAMLToken cache, modify the custom property in the CallbackHandler for the SAMLToken generator. Add the cacheToken property and set the value to `false`.

Custom property name	Value
cacheToken	false

Securing web services applications using the WSS APIs at the message level:

Standards and profiles address how to provide protection for messages that are exchanged in a web service environment. Web Services Security is a message-level standard that is based on securing SOAP messages through XML digital signature, confidentiality through XML encryption, and credential propagation through security tokens.

Before you begin

To secure web services, you must consider a broad set of security requirements, including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, delegation, and auditing across a spectrum of application and business topologies. One of the key requirements for the security model in today's business environment is the ability to interoperate between formerly incompatible security technologies in heterogeneous environments. The complete Web Services Security protocol stack and technology roadmap is described in the web services roadmap.

About this task

The Organization for the Advancement of Structured Information Standards (OASIS) Web Services Security: SOAP Message Security Version 1.1 specification is the basic messaging transport for all web services. SOAP 1.2 adds extensions to the existing SOAP 1.1 extensions so that you can build secure web services. Attachments can be added to SOAP messages by using Message Transmission Optimization Mechanism (MTOM) and XML-binary Optimized Packaging (XOP) instead of the SOAP with Attachments (SWA) profile.

The OASIS Web Services Security (WS-Security) Version 1.1 specification is the building block that is used in conjunction with other web service and application-specific protocols to accommodate a wide variety of security models. Web Services Security for WebSphere Application Server is based on specific standards that are included in the OASIS Web Services Security Version 1.1 specification and profiles.

The Version 1.1 specification defines additional facilities for protecting the integrity and confidentiality of a message. The Version 1.1 specification also provides the mechanisms for associating security-related claims with the message. The Web Services Security Version 1.1 standards that are supported by WebSphere Application Server include the signature confirmation, encrypted header elements, the Username Token Profile and the X.509 Token Profile. The Username Token Profile and the X.509 Token Profile have been updated as Version 1.1 profiles. For the X.509 Certificate Token Profile, one new type of security token reference is the Thumbprint reference, which is specified in the binding.

XML Schema, Part 1 and Part 2 are specifications that explain how schemas are organized in XML documents. The two WS-Security Version 1.0 schemas have been updated to the Version 1.1 specifications plus a new Version 1.1 schema has been added. Note that the Version 1.1 schema does not replace the Version 1.0 schema but instead builds upon it by defining an additional set of capabilities within a Version 1.1 namespace.

You can use the following methods to configure Web Services Security and to define policy types to secure the SOAP messages:

- **Use the administrative console to configure policy sets.**

  This method uses the bootstrap policy that is defined in the policy set. You can use policy sets, or assertions about how services are defined, to simplify your security configuration for web services. You can use the administrative console to create, modify, and delete custom policy sets. A set of default policy sets are available.

  For example, you can define the bootstrap policy in the policy set to secure the Web Services Trust (WS-Trust) SOAP messages.

  You can also use the administrative console to perform policy set management tasks and to secure web services using encryption, signing information, and security tokens.

  The following steps high-level steps describe how to configure WebSphere Application Server to use WS-Security and to secure the SOAP messages using the administrative console. The generator and consumer tasks that are discussed in the following steps use WS-Security Versions 1.0 and 1.1.

  - Create and configure the application policy sets or the system policy sets for trust service.
  - Define the policy types to be used to secure the SOAP messages when creating and configuring the policy sets.

- Configure the policy set binding. Select either the symmetric or asymmetric binding assertion to describe the token type and the algorithm to be used for message protection.
- Assemble your Web Services Security-enabled application by using an assembly tool.

• **Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)**

WebSphere Application Server uses a new API programming model. In addition to the existing JAX-RPC programming model, a new programming model, Java API for XML Web Services (JAX-WS), has been added. The JAX-WS programming standard aligns with the document-centric messaging model and replaces the remote procedure call programming model defined by the Java API for XML-based RPC (JAX-RPC) specification.

For example, an application could create system policy sets and then use the WebSphere Application Server WSS API to acquire the security context token for programmatic API-based Web Services Secure Conversation (WS-SecureConversation).

You can also use the administrative console to perform the encryption, signing, and token configuration tasks that the WSS APIs perform to secure web services.

The following high-level steps describe how to configure WebSphere Application Server to use WS-Security and to secure the SOAP messages using the WSS APIs. The generator and consumer tasks that are discussed in the following steps use WS-Security Versions 1.0 and 1.1.

- Use the WSSSignature API to configure the signing information for the request generator (client side) binding.

  Different message parts can be specified in the message protection for a request on the generator side. The default required parts are BODY, ADDRESSING_HEADERS and TIMESTAMP.

  The WSSSignature API also specifies the different algorithm methods to be used with the signature for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.
- Use the WSSSignPart API if you want to change the digest method and the transform method.

  The default signed parts are WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS and WSSSignature.TIMESTAMP.

  The WSSSignPart API also specifies the different algorithm methods to be used if you added or changed the signed parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N. For example, use the WSSSignPart API if you want to generate the signature for the SOAP message using the SHA256 digest method instead of the default value of SHA1.
- Use the WSSEncryption API to configure the encryption information on the request generator side.

  The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The default targets of encryption are BODY_CONTENT and SIGNATURE.

  The WSSEncryption API also specifies the different algorithm methods to be used to protect message confidentiality. The default data encryption method is AES128. The default key encryption method is KW_RSA_OAEP.
- Use the WSSEncryptPart API if you want to set the transform method only.

  For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

  No algorithm methods are required for encrypted parts.
- Use the WSS API to configure the token on the generator side.

  The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for creating the security token on the generator side. Different stand-alone tokens can be sent in request and response. The default token is the X509Token. The other token that can be used for signing is the DerivedKeyToken, which is used only with Web Services Secure Conversation (WS-SecureConversation).
- Use the WSSVerification API to verify the signature for the response consumer (client side) binding.

Different message parts can be specified in the message protection for a response on the consumer side. The required targets for verification are BODY, ADDRESSING_HEADERS and TIMESTAMP.

The WSSVerification API also specifies the different algorithm methods to be used for verifying the signature and for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.

- Use the WSSVerifyPart API to change the digest method and the transform method. The required verify parts are WSSVerification.BODY, WSSVerification.ADDRESSING_HEADERS and WSSVerification.TIMESTAMP.

  The WSSVerifyPart API also specifies the different algorithm methods to be used if you added or changed the verification parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N.

- Use the WSSDecryption API to configure the decryption information for the response consumer (client side) binding.

  The decryption information on the consumer side is used for decrypting an incoming SOAP message. The targets of decryption are BODY_CONTENT and SIGNATURE. The default key encryption method is KW_RSA_OAEP.

  No algorithm methods are required for decryption.

- Use the WSSDecryptPart API if you want to set the transform method only.

  For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

  No algorithm methods are required for decrypted parts.

- Use the WSS API to configure the token on the consumer side.

  The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for validating (authenticating) the security token on the consumer side. Different stand-alone tokens can be sent in request or response.

  The WSS API adds the information for the candidate token that is used for decryption. The default token is X509Token.

- **Use the wsadmin administrative scripting tool to configure policy sets.**

  This method allows you to create, manage, and delete policy sets from the command-line or to create scripts to automate your tasks. You can use the wsadmin tool and the PolicySetManagement command group to manage default policy sets, create custom policy sets, configure policies, and manage attachments and bindings. For more information, use the policy set scripting topics in the information center.

To secure web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. You must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

Results

After completing these high-level steps for WebSphere Application Server, you have secured web services by configuring policy sets and by using the WSS API to configure encryption and decryption, the signature and signature verification information, and the consumer and generator tokens.

Securing messages at the request generator using WSS APIs:

You can secure SOAP messages by configuring signing information, encryption, and generator tokens to protect message integrity, confidentiality, and authenticity, respectively. This request (client-side) generator configuration defines the Web Services Security requirements for the outgoing SOAP message request.

Before you begin

To secure web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. Therefore, in addition to securing messages at the request generator level, you must also secure messages at the response consumer level.

About this task

The request (client-side) generator configuration requirements involve generating a SOAP message request that uses a digital signature, incorporates encryption, and attaches security tokens.

To secure web service applications, you must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

You can use the following interfaces to configure Web Services Security and to define policy types to secure the SOAP messages:
- Use the administrative console to configure policy sets.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

The following high-level steps use the WSS APIs:

Procedure
- Configure generator signing to protect message integrity.
- Configure encryption to protect message confidentiality.
- Attach generator tokens to protect message authenticity.
- Propagate self-issued SAML bearer tokens using WSS APIs.
- Propagate self-issued SAML sender-vouches tokens with message protection using WSS APIs.
- Propagate self-issued SAML sender-vouches tokens with transport protection using WSS APIs.
- “Sending self-issued SAML holder-of-key tokens with symmetric key using WSS APIs” on page 514.
- “Sending self-issued SAML holder-of-key tokens with asymmetric key using WSS APIs” on page 516.

Results

After completing these procedures, you have secured messages at the request generator level.

What to do next

Next, if not already configured, secure messages with signature verification, decryption, and consumer tokens at the response consumer (client-side) level.

Configuring encryption to protect message confidentiality using the WSS APIs:

You can configure encryption information for the client-side request generator (sender) bindings. Encryption information is used to specify how the generators (senders) encrypt outgoing SOAP messages. To configure encryption, specify which message parts to encrypt and specify which algorithm methods and security tokens are to be used for encryption.

Before you begin

Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. Prior to

configuring encryption, familiarize yourself with XML encryption.

About this task

For encryption, you must specify the following:
- Which parts of the message are to be encrypted.
- Which encryption algorithms to specify.

To configure encryption and encrypted parts on the client side, use the WSSEncryption and WSSEncryptPart APIs, or configure policy sets using the administrative console.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

WebSphere Application Server uses encryption information for the default generator to encrypt parts of the SOAP message. The WSSEncryption API configures the following required parts as encrypted parts.

Table 47. Required encrypted parts. Use encrypted parts to increase the confidentiality of SOAP messages.

Encryption parts	Description
Keywords	Keywords are used to add the encrypted parts to the SOAP message.
XPath expression	An XPath expression is used to add the encrypted parts to the SOAP message.
WSSEncryptPart object	This object adds the encrypted parts to the SOAP message.
WSSSignature object	This object adds the signature component as an encrypted part.
Header	This part adds the header in the SOAP header, specified by QName, as an encryption part.
Security token object	This object adds the security token as an encryption part.

Web Services Security API (WSS API) supports symmetric encryption, by using a shared key, only when Web Services Secure Conversation (WS-SecureConversation) is used.

The WSS APIs allow the use of either keywords or an XPath expression to specify the parts of the message that are to be encrypted. WebSphere Application Server supports the use of the following keywords:

Table 48. Supported encryption keywords. Use keywords to specify encrypted parts.

Keyword	References
BODY_CONTENT	The keyword for the contents of the SOAP message body as an encryption target.
SIGNATURE	The keyword for the signature component as an encryption target.

If configuring using the WSS APIs, the WSSEncryption and WSSEncryptPart APIs complete these high-level steps:

Procedure

1. Use the WSSEncryption API to configure encryption. The WSSEncryption API performs these tasks by default:
   a. Generates the callback handler.
   b. Generates the generator security token object.
   c. Adds the security token reference type.
   d. Adds the signature component.
   e. Adds the WSSEncryptPart object.
   f. Adds the parts to be encrypted. Adds the default parts as targets of encryption by using keywords and XPath expressions.
   g. Adds the header in the SOAP message, specified by QName.

h. Sets the default data encryption method.
i. Specifies whether the key is to be encrypted using a Boolean value.
j. Sets the default key encryption method.
k. Selects a part reference.
l. Sets the MTOM optimization Boolean value.
2. Use the WSSEncryptPart API to configure encrypted parts or add a transform method. The WSSEncryptPart API performs these tasks by default:
   a. Sets the encrypted parts specified by using keywords or an XPath expression.
   b. Sets the encrypted parts specified by an XPath expression.
   c. Sets the signature component object, WSSSignature.
   d. Sets the header in the SOAP message, specified by QName.
   e. Sets the generator security token.
   f. Adds the transform method, if needed.
3. Change from the default values for algorithm or message parts, as needed. For example: you could change one or more of the following items:
   - Change the data encryption algorithm from the default value of AES 128.
   - Change the key encryption algorithm from the default value of KW_RSA_OAEP.
   - Specify to not encrypt the key (false).
   - Change the security token type from default of X.509 token.
   - Change the security token reference type from the default value of SecurityToken.REF_STR.
   - Only use BODY_CONTENT as an encryption part and not use SIGNATURE also.
   - Turn MTOM optimization on (true).

Results

The encryption information is configured for the generator binding.

Example

The following is an example of the WSSEncryption API:

```
WSSFactory factory = WSSFactory.getInstance();
WSSGenerationContext gencont = factory.newWSSGenerationContext();

X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);
WSSEncryption enc = factory.newWSSEncryption(token);

gencont.add(enc);
```

What to do next

You must configure similar decryption information for the client-side response consumer (receiver) bindings, if you have not already configured the information.

Next, review the WSSEncryption API process.

Encrypting the SOAP message using the WSSEncryption API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the client for request encryption on the generator side, use the WSSEncryption API to encrypt the SOAP message. The WSSEncryption API specifies which request SOAP message parts to encrypt when configuring the client.

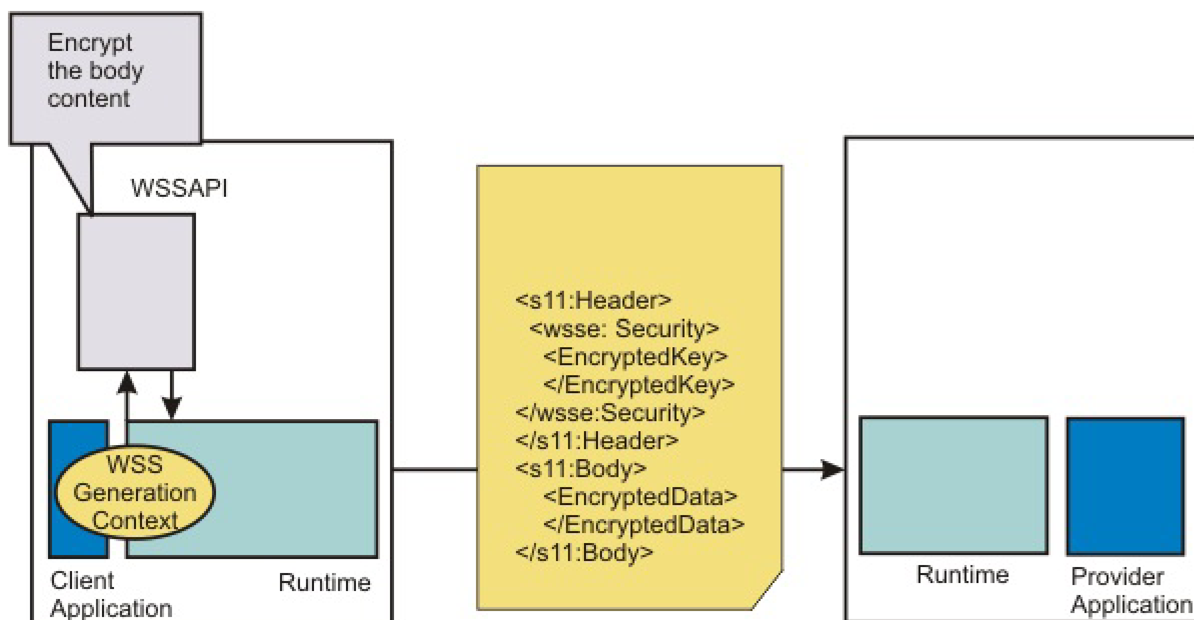
Before you begin

You can use the WSS API or use policy sets on the administrative console to enable encryption and add generator security tokens in the SOAP message. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:

- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts, as needed, using the WSSEncryptPart API.

About this task

The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The client generator configuration must match the configuration for the provider consumer.



Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and which message parts to attach encrypted Nonce and timestamp elements to.

The following encryption parts can be configured:

Table 49. Encryption parts. Use the encryption parts to enable encryption in messages.

Encryption parts	Description
part	Adds the WSSEncryptPart object as a target of the encryption part.
keyword	Adds the encryption parts using keywords. WebSphere Application Server supports the following keywords: • BODY_CONTENT • SIGNATURE
xpath	Adds the encryption part using an XPath expression.
signature	Adds the WSSignature component as a target of the encrypted part.
header	Adds the SOAP header, specified by QName, as a target of the encrypted part.
securityToken	Adds the SecurityToken object as a target of the encrypted part.

For encryption, certain default behaviors occur. The simplest way to use the WSSEncryption API is to use the default behavior (see the example code).

WSSEncryption provides defaults for the key encryption algorithm, the data encryption algorithm, the security token reference method, and the encryption parts such as the SOAP body content and the signature. The encryption default behaviors include:

Table 50. Encryption decisions. Use encryption default behavior to secure the message body content and signature.

Encryption decisions	Default behavior
Which SOAP message parts to encrypt using keywords	Sets the encryption parts that you can add using keywords. The default encryption parts are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • WSSEncryption.BODY_CONTENT • WSSEncryption.SIGNATURE
Which data encryption method to choose (algorithm)	Sets the data encryption method. Both data and key encryption methods can be specified. The default data encryption algorithm method is AES 128. WebSphere Application Server supports these data encryption methods: • WSSEncryption.AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc • WSSEncryption.AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc • WSSEncryption.AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc • WSSEncryption.TRIPLE_DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc
Whether to encrypt the key (isEncrypt)	Specifies whether to encrypt the key. The values are `true` or `false`. The default value is to encrypt the key (`true`).
Which key encryption method to choose (algorithm)	Sets the key encryption method. Both data and key encryption methods can be specified. The default key encryption algorithm method is key wrap RSA OAEP. WebSphere Application Server supports these key encryption methods: • WSSEncryption.KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128 • WSSEncryption.KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192 • WSSEncryption.KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256 • WSSEncryption.KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p • WSSEncryption.KW_RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5 • WSSEncryption.KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes
Which security token to specify (securityToken)	Sets the SecurityToken. The default security token type is the X509Token. WebSphere Application Server provides the following pre-configured consumer token types: • Derived key token • X.509 tokens
Which token reference to use (refType)	Sets the type of the security token reference. The default token reference is SecurityToken.REF_KEYID. WebSphere Application Server supports the following token reference types: • SecurityToken.REF_KEYID • SecurityToken.REF_STR • SecurityToken.REF_EMBEDDED • SecurityToken.REF_THUMBPRINT
Whether to use MTOM (mtomOptimize)	Sets Message Transmission Optimization Mechanism (MTOM) optimization for the encrypted part.

Procedure

1. To encrypt the SOAP message using the WSSEncryption API, first ensure that the application server is installed.
2. The WSS API process for encryption performs these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory used for encryption.
   d. Creates WSSEncryption from the WSSFactory instance using the SecurityToken. The default behavior of WSSEncryption is to encrypt the body content and the signature.
   e. Adds a new part to be encrypted in WSSEncryption if the existing part is not appropriate. After addEncryptPart(), addEncryptHeader(), or addEncryptPartByXPath() is called, the default part is cleared.
   f. Calls the encryptKey(false) if the key is not to be encrypted.
   g. Sets the data encryption method if the default method is not appropriate.

h. Sets the key encryption method if the default method is not appropriate.
i. Sets the token reference if the default token reference is not appropriate.
j. Adds WSSEncryption to WSSConsumingContext.
k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

If there is an error condition during encryption, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample code using methods that are defined in WSSEncryption:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance (step: b)
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
X509GenerateCallbackHandler callbackHandler = new
        X509GenerateCallbackHandler(
     "",
     "enc-sender.jceks",
     "jceks",
     "storepass".toCharArray(),
     "bob",
     null,
     "CN=Bob, O=IBM, C=US",
     null);

// Generate the security token used for encryption (step: c)
SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);

// Generate WSSEncryption instance (step: d)
WSSEncryption enc = factory.newWSSEncryption(token);

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part specified by the keyword (step: e)
   enc.addEncryptPart(WSSEncryption.BODY_CONTENT);

// Set the part in the SOAP Header specified by QName (step: e)
   enc.addEncryptHeader(new QName("http://www.w3.org/2005/08/addressing",
                             "MessageID"));

// Set the part specified by WSSSignature (step: e)
   SecurityToken sigToken = getSecurityToken();
      WSSSignature sig = factory.newWSSSignature(sigToken);
   enc.addEncryptPart(sig);

// Set the part specified by SecurityToken (step: e)
   UNTGenerateCallbackHandler untCallbackHandler =
         new UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
                                              untCallbackHandler);
   enc.addEncryptPart(unt, false);

// sSt the part specified by XPath expression (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
   enc.addEncryptPartByXPath(sb.toString());

// Set whether the key is encrypted (step: f)
// DEFAULT: true
   enc.encryptKey(true);

// Set the data encryption method (step: g)
```

```
// DEFAULT: WSSEncryption.AES128
   enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method (step: h)
// DEFAULT: WSSEncryption.KW_RSA_OAEP
   enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Set the token reference (step: i)
// DEFAULT: SecurityToken.REF_KEYID
  enc.setTokenReference(SecurityToken.REF_STR);

// Add the WSSEncryption to the WSSGenerationContext (step: j)
  gencont.add(enc);

// Process the WS-Security header (step: k)
gencont.process(msgcontext);
```

Note: The X509GenerationCallbackHandler does not need the key password because the public key is used for encryption. You do not need a password to obtain the public key from the Java keystore.

What to do next

If you have not previously specified which encryption methods to choose, use the WSS API or configure the policy sets using the administrative console to choose the data and key encryption algorithm methods.

Choosing encryption methods for generator bindings:

To configure the client for request encryption for the generator binding, you must specify which encryption methods to use when the client encrypts the SOAP messages.

Before you begin

Prior to completing these steps, read the XML encryption information to become familiar with encrypting and decrypting SOAP messages.

To specify which algorithm methods are to be used when the client encrypts the SOAP messages, complete the following tasks:

- Use the WSSEncryption API to configure the data encryption algorithm and the key encryption algorithm methods.
- Use the WSSEncryptPart API to configure a transform algorithm method, if needed. The default is no transform algorithm.

About this task

Some of the encryption-related definitions are based on the XML-Encryption specification. The following information defines some data encryption-related terms:

Data encryption method algorithm
: Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. This algorithm encrypts and decrypts data in fixed size, multiple octet blocks.

  By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

  For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Key encryption method algorithm
: Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the method to encrypt the key that is used to encrypt data. The algorithm represents public key encryption algorithms that are specified for encrypting and decrypting keys.

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property.

The property name is: com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

Table 51. Encryption usage types. The encryption usage types describe encryptions methods.

Usage types	Description
Data encryption	Specifies the algorithm URI that is used for both encrypting and decrypting data. Encrypts and decrypts data in fixed size, multiple octet blocks.
Key encryption	Specifies the algorithm URI that is used for encrypting and decrypting the encryption key.

Data encryption

WebSphere Application Server supports the following pre-configured data encryption algorithms:

Table 52. Data encryption algorithms. These pre-configuring encryption algorithms are supported by WebSphere Application Server.

Data encryption name	Algorithm URI
WSSEncryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSEncryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSEncryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSEncryption.TRIPLE_DES	A URI of data encryption algorithm, 3DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc

Key encryption

WebSphere Application Server supports the following pre-configured key encryption algorithms:

Table 53. Key encryption algorithms. These pre-configured encryption algorithms are supported by WebSphere Application Server.

Key encryption name	Algorithm URI
WSSEncryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128

Table 53. Key encryption algorithms (continued). These pre-configured encryption algorithms are supported by WebSphere Application Server.

Key encryption name	Algorithm URI
WSSEncryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSEncryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256
WSSEncryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSEncryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSEncryption.KW_TRIPLE_DES	http://www.w3.org/2001/04/xmlenc#kw-tripledes

To configure the encryption and encrypted part algorithm methods, use the WSSEncryption API, or configure policy sets using the administrative console.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints that use policy sets.

The WSS API process completes the following high-level steps to specify which encryption methods to use when configuring the client for request encryption:

Procedure

1. Using the WSSEncryption API, adds the required data encryption algorithm. The data encryption algorithm is used for encrypting or decrypting parts of a SOAP message. Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method.

   The client generator configuration must match the configuration for the provider consumer.

   The default data encryption algorithm is AES 128. The data encryption name is AES128, and the URI of the data encryption algorithm, is http://www.w3.org/2001/04/xmlenc#aes128-cbc. WebSphere Application Server supports the following pre-configured data encryption algorithms:

   - AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc

     The AES 128 algorithm is the default data algorithm method.
   - AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc

     Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

     To use this AES 192-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - TRIPLEDES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc
2. As needed, changes the WSSEncryption API method to specify another data encryption algorithm. For example, you might add the following code to change from the default AES 128 algorithm to the Triple DES algorithm:

```
// Default data encryption algorithm: AES128
WSSEncryption enc = factory.newWSSEncryption(x509t);
   enc.setEncryptionMethod(EncryptionMethod.TRIPLEDES_CBC);
gencont.add(enc);
```

3. Using the WSSEncryption API, adds the required key encryption algorithm. The key encryption algorithm is used for encrypting the key that is used for encrypting the message parts within the SOAP

message. If the encryption key, which is the key that is used for encrypting the message parts, is not encrypted, then the decryption API selects `false` to match the encryption key.

The client generator configuration must match the configuration for the provider consumer.

The default key encryption algorithm value is key wrap RSA OAP. The key encryption name is KW_RSA_OAEP, and the URI of the key encryption algorithm is http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p. WebSphere Application Server supports the following pre-configured key encryption algorithms:

- KW AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128
- KW AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192

  To use this key wrap AES 192 algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

  Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).KW AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256

  To use this key wrap AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
- KW RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

  The KW RSA OAEP algorithm is the default key algorithm method.

  When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this algorithm. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5. See more information at http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
- KW RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5
- KW TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

Note: For Web Services Secure Conversation, the WSSEncryption API might specify addition key-related information, such as the:
- algorithmName
- keyLength

Results

If there is an error condition, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample WSS API code using WSSEncryption.setEncryptionMethod() and WSSEncryption.setKeyEncryptionMethod().

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
                                 "",
                                 "enc-sender.jceks",
                                 "jceks",
                                 "storepass".toCharArray(),
                                 "bob",
```

```
                            null,
                            "CN=Bob, O=IBM, C=US",
                            null);

// Generate the security token used for encryption
   SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);

// Generate WSSEncryption instance
   WSSEncryption enc = factory.newWSSEncryption(token);

// Set the data encryption method
// DEFAULT: WSSEncryption.AES128
      enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
      enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Add the WSSEncryption to the WSSGenerationContext
   gencont.add(enc);

// Generate the WS-Security header
gencont.process(msgcontext);
```

What to do next

Next, if you want to add a transform algorithm, review the WSSEncryptPart API process task.

Encryption methods:

For request generator binding settings, the encryption methods include specifying the data and key encryption algorithms to use to encrypt the SOAP message. The WSS API for encryption (WSSEncryption) specifies the algorithm name and the matching algorithm uniform resource identifier (URI) for the data and key encryption methods. If the data and key encryption algorithms are specified, only elements that are encrypted with those algorithms are accepted.

Data encryption algorithms

The data encryption algorithm is used to encrypt parts of the SOAP message, including the body and the signature. Data encryption algorithms specify the algorithm uniform resource identifier (URI) for each type of data encryption algorithms.

The following pre-configured data encryption algorithms are supported:

Table 54. Data encryption algorithms. The algorithms are used to encrypt SOAP messages.

Data encryption algorithm name	Algorithm URI
WSSEncryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSEncryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSEncryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSEncryption.TRIPLE_DES	A URI of data encryption algorithm, TRIPLE DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

For the AES256-cbc and the AES192-CBC algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The data encryption algorithm configured for encryption for the generator side must match the data encryption algorithm that is configured for decryption for the consumer side.

Key encryption algorithms

This algorithm is used to encrypt and decrypt keys. This key information is used to specify the configuration that is needed to generate the key for digital signature and encryption. The signing information and encryption information configurations can share the key information. The key information on the consumer side is used for specifying the information about the key that is used for validating the digital signature in the received message or for decrypting the encrypted parts of the message. The request generator is configured for the client.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints using the policy sets.

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. The following pre-configured key encryption algorithms are supported:

Table 55. Supported pre-configured key encryption algorithms. The algorithms are used to encrypt and decrypt keys.

WSS API	URI
WSSEncryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128
WSSEncryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSEncryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256
WSSEncryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSEncryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSEncryption.KW_TRIPLE_DES	A URI of key encryption algorithm, key wrap TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

For Secure Conversation, additional key-related information must be specified, such as:
- algorithmName
- keyLength

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:
- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The key encryption algorithm for the generator must match the key decryption algorithm that is configured for the consumer.

This example provides sample code for encryption to use the Triple DES for the data encryption method and to use RSA1.5 for the key encryption method:

```
// get the message context
Object msgcontext = getMessageContext();

// generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

// generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "enc-sender.jceks",
  "jceks",
  "storepass".toCharArray(),
  "bob",
  null,
  "CN=Bob, O=IBM, C=US",
  null);

// generate the security token used to the encryption
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

// generate WSSEncryption instance to encrypt the SOAP body content
WSSEncryption enc = factory.newWSSEncryption(token);
enc.addEncryptPart(WSSEncryption.BODY_CONTENT);

// set the data encryption method
// DEFAULT: WSSEncryption.AES128
enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// add the WSSEncryption to the WSSGenerationContext
gencont.add(enc);

// generate the WS-Security header
gencont.process(msgcontext);
```

Adding encrypted parts using the WSSEncryptPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure encrypted parts for the request generator (client side) bindings, use the WSSEncryptPart API to define and add to the listing of elements in the encrypted part. WSSEncryptPart is an interface that is part of the com.ibm.websphere.wssecurity.wssapi.encryption package.

Before you begin

You can use the WSS APIs or configure policy sets using the administrative console to enable the encrypted parts. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:

- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts using the WSSEncryptpart API, as needed.

About this task

Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and which message parts to attach encrypted elements to. The encryption information on the generator side is used for encrypting an outgoing SOAP message. The request generator is configured for the client.

The WSSEncryptPart API specifies information related to encrypted parts and sets the encrypted parts that have been added for message confidentiality protection. Use the WSSEncryptPart to set the transform method and to specify the part to which the transform method is to be applied. Sets the transform method only if using SOAP with Attachments. The WSSEncryptPart is usually not needed except, in some case for tasks such as setting the transform method.

The encrypted parts and related information displayed in the following table are used to protect the confidentiality of messages.

Table 56. Encrypted parts. Use encrypted parts to secure messages.

Encrypted parts	Description
part	Adds the WSSEncryptPart object as a target of the encryption part.
keyword	Adds the encrypted parts using keywords. The default encryption parts that you can add using keywords are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • BODY_CONTENT • SIGNATURE
xpath	Adds the encrypted part by using an XPath expression.
signature	Adds the WSSSignature component as a target of the encrypted part. WSSSignature is applicable only if the SOAP message contains a signature element.
header	Adds the SOAP header, specified by QName, as a target of the encrypted part.
securityToken	Adds the SecurityToken object as a target of the encrypted part.

For encrypted parts, certain default behaviors occur. The simplest way to use the WSSEncryptPart API is to use the default behavior. The WSSEncryptPart API provides defaults for specifying the transform algorithm, setting objects as targets, specifying the encrypted parts, such as: the SOAP body content and the signature.

The encryption default behaviors include:

Table 57. Encrypted part decisions. Several encrypted message parts are set by default.

Encrypted part decisions	Default behavior
Which SOAP message parts to encrypt using keywords	Specifies which keywords to use for the encrypted parts. WebSphere Application Server sets the following SOAP message parts by default for encryption: • WSSEncryption.BODY_CONTENT • WSSEncryption.SIGNATURE
Which transform method to add	WebSphere Application Server does not specify any transform method by default. Specify a transform method only if using SOAP with Attachments.

Procedure

1. To encrypt the SOAP message parts using the WSSEncryptPart API, first ensure that the application server is installed.
2. The WSS API process using WSSEncryptPart follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the encryption.
   d. Creates WSSEncryption from the WSSFactory instance using SecurityToken.
   e. Creates WSSEncryptPart from WSSFactory.

f. Adds the parts to be encrypted and to be applied with the transform in WSSEncryptPart. WebSphere Application Server sets these encrypted parts by default for WSSEncryptPart: the BODY_CONTENT and SIGNATURE. After you add other encrypted parts, the default values are no longer valid. For example, if you call addEncryptPart(securityToken, false), only the security token is encrypted, and not the signature and body content. So if you want to encrypt the security token, the signature, and the body content, you must call addEncryptPart(securityToken, false), addEncryptPart(WSSEncryption.SIGNATURE), and addEncryptPart(WSSEncrypyion.BODY_CONTENT).
g. Sets the transform method.
h. Adds WSSEncryptPart to WSSEncryption.
i. Adds WSSEncryption to WSSGenerationContext.
j. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

If there is an error condition during encryption of the message parts, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

What to do next

After enabling encrypted parts for the request generator (client side) binding, you must specify the same parts to be decrypted for the response consumer (client side) bindings. Next, to configure decryption and decrypted parts, use the WSS APIs or configure policy sets using the administrative console.

Configuring generator signing information to protect message integrity using the WSS APIs:

You can configure the signing information to protect message integrity for the request (client side) generator binding. Signing information includes the signature and the signed parts. To keep the integrity of the message, digital signatures are typically applied.

Before you begin

In addition to using a digital signature and configuring the signing information, the following tasks should also be performed:
- Verify the signing information.
- Incorporate encryption.
- Attach security tokens.

About this task

Integrity refers to digital signature while confidentiality refers to encryption. Integrity is provided by applying a digital signature to a SOAP message. To configure the signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity decreases the risk of data modification when you transmit data across a network.

Also, message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using the signature algorithm methods. The WSS APIs specify which algorithm is to be used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports several pre-configured request signing algorithm methods.

You can use the following interfaces to configure Web Services Security and to protect SOAP message integrity:

- Use the administrative console to configure policy sets for the signing information.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client).

Perform the following signing tasks, using the WSS APIs, to configure the signing information and to protect message integrity for the generator binding.

Procedure

- Configure the signing information using the WSSSignature API. Configure the signing information for the generator binding using the WSSSignature API. Signing information is used to sign parts of a message including the SOAP body, the time stamp, and the WS-Addressing headers. Both signing and encryption can be applied to the same message parts, such as the SOAP body.
- Add or change signed parts using the WSSSignPart API.
- Configure the client for request signing methods using the WSSSignature or WSSSignPart APIs. To configure the client for request signing, choose the signing methods. The request signing methods include the signature, the canonicalization, the digest, and the transform methods. Use the WSSSignature API to configure the signature and canonicalization methods. Use the WSSSignPart API to configure the digest and transform methods.

Results

The WSS APIs also specify the security token for the generator (client) binding and set the type of token reference to protect message authenticity. By completing the steps in these tasks, you have configured generator signing to protect the integrity of the SOAP message.

What to do next

Next, verify the consumer signing information by using the WSS APIs or by configuring policy sets using the administrative console.

Configuring signing information using the WSS APIs:

You can configure the signing information for the client-side request generator (sender) bindings. Signing information is used to sign and validate parts of a message including the SOAP body, the timestamp information, and the Username token. To configure the client for request signing, specify which message parts to digitally sign when configuring the client.

Before you begin

WebSphere Application Server uses XML digital signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method. Prior to completing these steps, familiarize yourself with XML digital signature for signing and verifying digital signatures for digital content.

About this task

By including XML signature in SOAP messages, the following issues are realized: message integrity and authentication. *Integrity* refers to digital signature whereas confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. WebSphere Application Server uses the signing information for the default generator to sign parts of the message, such as the body, time stamp, and Username token.

For the signing information, you must specify the following:

- Which parts of the message are to be signed.
- The key information that is referenced by the key information for the signing keys.

- The signing algorithms.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

The WSSSignature API configures the following parts as signature parts:

Table 58. Pre-configured signature parts. Use the signing information to validate parts of a message.

Part	Description
Security token object	This object authenticates the client. If this option is specified, then the message is signed. You can digitally sign the message using a security token if a login configuration authentication method is selected.
WSSTimestamp object	This object adds a time stamp to a message. The time stamp determines if the message is valid based on the time that the message is sent and then received.
WSSSignature Part object	This object adds the signature parts to a message.
SOAP header and the QName as a target	This signature part adds the header, specified by QName, as a verification part.

The WSS APIs allow the use of keywords or an XPath expression to specify which parts of the message are to be signed. WebSphere Application Server supports the use of the following keywords:

Table 59. Supported signature keywords. Key information is used to specify which parts of a message are signed.

Keyword	References
ADDRESSING_HEADERS	The Web Services Addressing (WS-Addressing) headers.
BODY	The SOAP message body. The body is the user data portion of the message.
TIMESTAMP	The creation and expiration timestamp information.

The Web Services Security API (WSS API) are used to configure the signing information for the request generator (client side) section of the bindings file. To configure the signing information on the client side, use the WSS APIs or configure policy sets for signing using the administrative console.

If configuring using the WSS APIs, the WSSSignature and WSSSignPart APIs complete the following steps to specify which message parts to digitally sign when configuring the client for request generator signing:

Procedure

1. The WSSSignature API adds the required parts of the SOAP message to digitally sign. Either a keyword or an XPath expression can be used to specify the required encryption parts.
2. The WSSSignature API sets the signature method algorithm. The default signature method is RSA_SHA1. WebSphere Application Server supports the following pre-configured algorithms:
   - RSA SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - HMAC SHA1http://www.w3.org/2000/09/xmldsig#hmac-sha1

   WebSphere Application Server does not support the following algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1. You cannot use the DSA-SHA1 algorithm if you want to be compliant with the Basic Security Profile (BSP).

   Any ds:SignatureMethod/@Algorithm element in a signature is based on a symmetric key and must have a value of RSA-SHA1 or HMAC-SHA1.

   The algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.
3. The WSSSignature API sets the canonicalization method. The default signature method is EXC_C14N. WebSphere Application Server supports the following pre-configured algorithms:
   - The URI of the exclusive canonicalization algorithm, EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#.
   - The URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#.

The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer.

4. The WSSSignature API adds a security token. The API adds information about the security token that is to be used for the signature, such as:
   - The class for security token.
   - The callback handler
   - The name of the JAAS login configuration.
5. The WSSSignature API sets the type of security token and sets the type of token reference. WebSphere Application Server supports the following pre-configured token references:
   - SecurityToken.REF_STR

     Represents the security token reference as a token reference type.
   - SecurityToken.REF_KEYID

     Represents the key identifier reference as a token reference type.
   - SecurityToken.REF_EMBEDDED

     Represents the embedded reference as a token reference type.
   - SecurityToken.REF_THUMBPRINT

     Represents the thumbprint reference as a token reference type.
6. If SecurityToken.REF_KEYID is set as the type of token reference, the WSSSignature API sets the key information signature type and configures the key information that is referenced by the key information references. WebSphere Application Server supports the following:
   - Specifying that the KeyInfo element is not signed.
   - Specifying that the entire <KeyInfo> element is signed.
   - Specifying that the child elements <Keyinfochildelements> of the <KeyInfo> element are signed.

   If you do not specify one of the previous signature types, WebSphere Application Server specifies that the entire <KeyInfo> element is signed, by default.

   If you select Keyinfo or Keyinfochildelements and you select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm in a subsequent step, WebSphere Application Server also signs the referenced token.

   The key information signature type for the generator must match the signature type for the consumer.
7. The WSSSignature API specifies whether to require signature confirmation. The OASIS Web Services Security (WS-Security) Version 1.1 specification defines the use of signature confirmation. If you are using WS-Security Version 1.0, this function is not available.

   The signature confirmation value is stored in order to validate the signature confirmation with it after the receiving message is returned. This method is called if the response message is expected to attach the signature confirmation into the SOAP message.
8. The WSSSignPart API specifies the part reference. The part reference specifies which parts of the message to digitally sign.

   The part reference refers to the message part that is digitally signed. The part attribute refers to the name of the <Integrity> element when the <PartReference> element is specified for the signature. You can specify multiple <PartReference> elements within the <SigningInfo> element. The <PartReference> element has two child elements when it is specified for the signature verification: <DigestTransform> and <Transform>.
9. The WSSSignPart API specifies the digest method algorithm. The digest method algorithm specified within the <DigestMethod> element is used in the <SigningInfo> element.

   WebSphere Application Server supports the following pre-configured digest algorithms:
   - http://www.w3.org/2000/09/xmldsig#sha1
   - http://www.w3.org/2001/04/xmlenc#sha256
   - http://www.w3.org/2001/04/xmlenc#sha512

10. The WSSSignPart API specifies the transform algorithm. The transform algorithm is that is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following pre-configured transform algorithms:
   - http://www.w3.org/2001/10/xml-exc-c14n#
   - http://www.w3.org/TR/1999/REC-xpath-19991116

     Do not use this transform algorithm if you want to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.
   - http://www.w3.org/2002/06/xmldsig-filter2
   - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - http://www.w3.org/2002/07/decrypt#XML
   - http://www.w3.org/2000/09/xmldsig#enveloped-signature

   The transform algorithm that you select for the generator must match the transform algorithm that you select for the consumer.

   Important: If both of the following conditions are true, WebSphere Application Server signs the referenced token:
   - You previously selected the Keyinfo or the Keyinfochildelements option
   - You select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm.
11. If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. Configure policy sets using the administrative console to configure the same actor strings for the web service on the server, which processes the request and sends the response back.

    The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

Results

After the WSSSignature and WSSSignPart APIs complete these steps, the signing information is configured for the generator sections of the bindings files.

Example

The following example shows WSS API sample code to configure the signature, to generate the callback handler, and to specify the X.509 token type as the security token:

```
WSSFactory factory = WSSFactory.getInstance();
// Instantiate a generation context
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler and specify the X.509 token
   X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
   SecurityToken token = factory.newSecurityToken(X509Token.class,
                                                  callbackHandler);

// Set the signature information
WSSSignature sig = factory.newWSSSignature(token);
// Add the header using QName
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "To"));
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "MessageID"));
```

```
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "Action"));
// Apply the signature
   gencont.add(sig);

// Secure the message
gencont.process(msgctx);
```

What to do next

You must configure similar signature information for the client-side request consumer (receiver) bindings by completing the following verification tasks:

- Verify the signature
- Choose the signature algorithm methods.
- Change or add signed parts, as needed.

If signature verification is already configured, configure the encryption and decryption information, or configure the consumer and generator tokens.

Configuring signing information using the WSSSignature API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the signing information for the generator binding sections for the client-side request, use the WSSSignature API. The WSSSignature API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to enable the signing information. To secure SOAP messages, you must complete the following signing tasks:

- Configure the signing information.
- Choose the signing methods.
- Add or change signed parts, as needed.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing algorithms such as RSA-SHA1 and HMAC-SHA1.

XML signature defines many methods for describing key information and enables the definition of a new method. XML canonicalization (C14N) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. The C14N process is used to canonicalize XML information. Select an appropriate C14N algorithm because the information that is canonicalized depends on this algorithm.

The signing information specifies the integrity constraints that are applied to generated messages. The constraints include specifying which message parts within the generated message must be digitally signed, and the message parts to attach digitally signed Nonce and timestamp elements to. The following signature and related signature part information are configured:

Table 60. Signature parts information. Use the signature parts to secure messages.

signature parts	Description
keyword	Adds a signature part using keywords. Use the following keywords for the signature parts: • ADDRESSING_HEADERS • BODY • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds a signature part by using an XPath expression.
part	Adds a WSSSignPart object as a target of the signature part.
timestamp	Adds a WSSTimestamp object as a target of the signature part. When specified, the timestamp information also specifies when the message is generated and when it expires.
header	Adds the header, specified by QName, as a target of the signature part.
securityToken	Adds a SecurityToken object as a target of the signature part.

For signing information, certain default behaviors occur. The simplest way to use the WSSSignature API is to use the default behavior (see the example code). The default values are defined by the WSS API for the signing method, the canonicalization method, the security token references, and the signature parts.

Table 61. Signature default behaviors. Several signature behaviors are configured by default.

Signature decisions	Default behavior
Which keywords to use	Sets the keywords. WebSphere Application Server supports the following keywords by default: • ADDRESSING_HEADERS • BODY • TIMESTAMP
Which signature method to use	Sets the signature algorithm. The default signature method is RSA SHA1. WebSphere Application Server supports the following pre-configured signature methods: • WSSSignature.RSA_SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1 • WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1 The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported.
Which canonicalization method to use	Sets the canonicalization algorithm. The default canonicalization method is EXC C14N. WebSphere Application Server supports the following pre-configured canonicalization methods: • WSSSignature.EXC_C14N; http://www.w3.org/2001/10/xml-exc-c14n# • WSSSignature.C14N: http://www.w3.org/2001/10/xml-c14n#
Whether signature confirmation is required	Sets whether to require signature confirmation. The default value is **false**. Signature confirmation is defined in the OASIS Web Services Security Version 1.1 specification. If required, the value of your signature confirmation is stored in order to use it to validate the signature confirmation after receiving back the message that generated the signature confirmation in the response message. This method is for the requestor side.
Which security token to use	Sets the SecurityToken. The token type specifies which type of token to use for signing and validating messages. The X.509 token is the default token type. WebSphere Application Server provides the following pre-configured consumer token types: • Derived Key Token • X509 tokens You can also create custom token types, as needed.
Which token reference to set	Sets the refType. SecurityToken.REF_STR is the default value for the type of token reference. WebSphere Application Server supports these pre-configured token references types: • SecurityToken.REF_STR • SecurityToken.REF_KEYID • SecurityToken.REF_EMBEDDED • SecurityToken.REF_THUMBPRINT

If WSSSignature.requireSignatureConfirmation() is called, then the WSSSignature API expects that the response message will include the signature confirmation.

Procedure

1. To configure the signing information in a SOAP message by using the WSS API, first ensure that the application server is installed.
2. Use the WSSSignature API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signature follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance. WSSGenerationContext must be called in a JAX-WS client application.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.
   d. Creates WSSSignature from the WSSFactory instance using the SecurityToken. The default behavior of WSSSignature is to sign these signature parts: BODY, ADDRESSING_HEADERS, and TIMESTAMP.
   e. Adds the part to be signed, if the default part is not appropriate. If the digest method or transform method is changed, creates WSSSignPart and add it to WSSSignature.
   f. Creates WSSSignaturePart to WSSSignature. Calls the requiredSignatureConfirmation() method, if the signature confirmation is to be applied.
   g. Sets the canonicalization method, if the default is not appropriate.
   h. Sets the signature method, if the default is not appropriate.
   i. Sets the token reference, if the default is not appropriate.
   j. Adds WSSSignature to WSSGenerationContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

You have completed the steps to configure the signature for the generator section of the bindings. If there is an error condition when signing the message parts, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses methods that are defined in the WSSignature API.

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the com.ibm.websphere.wssecurity.wssapi.WSSFactory instance  (step: a)
   WSSFactory factory = com.ibm.websphere.wssecurity.wssapi.WSSFactory.getInstance();

// Generate the WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
       "",
       "dsig-sender.ks",
       "jks",
       "client".toCharArray(),
       "soaprequester",
       "client".toCharArray(),
       "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token to be used for the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class,
        callbackHandler);

// Generate the WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part to be signed  (step: e)
// DEFAULT: WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS,
//          and WSSSignature.TIMESTAMP.

// Set the part in the SOAP Header specified by QName  (step: e)
      sig.addSignHeader(new
                        QName("http://www.w3.org/2005/08/addressing",
```

```
                      "MessageID"));

// Set the part specified by the keyword  (step: e)
      sig.addSignPart(WSSSignature.BODY);

// Set the part specified by SecurityToken  (step: e)
   UNTGenerateCallbackHandler untCallbackHandler = new
      UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
         untCallbackHandler);
      sig.addSignPart(unt);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();
      sigPart.setSignPart(WSSSignature.TIMESTAMP);
      sigPart.setDigestMethod(WSSSignPart.SHA256);
      sig.addSignPart(sigPart);

// Set the part specified by WSSTimestamp  (step: e)
   WSSTimestamp timestamp = factory.newWSSTimestamp();
      sig.addSignPart(timestamp);

// Set the part specified by XPath expression  (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
      sig.addSignPartByXPath(sb.toString());

// Set to apply the signature confirmation  (step: f)
      sig.requireSignatureConfirmation();

// Set the canonicalization method  (step: g)
// DEFAULT: WSSSignature.EXC_C14N
      sig.setCanonicalizationMethod(WSSSignature.C14N);

// Set the signature method  (step: h)
// DEFAULT: WSSSignature.RSA_SHA1
      sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

// Set the token reference  (step: i)
// DEFAULT: SecurityToken.REF_STR
      sig.setTokenReference(SecurityToken.REF_KEYID);

// Add the WSSSignature to WSSGenerationContext  (step: j)
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgctx);
```

Note: The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

What to do next

Next, chose the algorithm methods if you want a method that is different from the default values. If the algorithm methods do not need to be changed, next use the WSSVerification API to verify the signature and specify the algorithm methods in the consumer section of the binding. Note that the WSSVerification API is only supported on the response consumer (client side).

Adding signed parts using the WSSSignPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure parts to be signed for the request generator (client side) bindings, use the WSSSignPart API to protect the integrity of messages and to configure the digest and transform algorithm methods. The WSSSignPart API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to configure the signing information. To secure SOAP messages using the signing information, you must complete one of the following tasks:

- Configure the signature information
- Configure signed parts, as needed.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing digest and transform algorithms (for example, SHA1 or TRANSFORM_EXC_C14N).

The signing information specifies the integrity constraints that are applied to generated messages. The signed parts are used to protect the integrity of messages. You can specify the signed parts to add for message integrity protection.

The following table shows the required signed parts when the digital signature security constraint (integrity) is defined:

Table 62. Signed parts information. Use the signed parts to secure messages.

Signed parts	Description
keyword	Adds signed parts using keywords. WebSphere Application Server supports the following keywords for signed parts: • BODY • ADDRESSING_HEADERS • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds the required signed parts by using an XPath expression.
header	Adds the header, specified by QName, as a signed part.
timestamp	Adds a WSSTimestamp object as a signed part. If specified, the timestamp information specifies when the message is generated and when it expires.

Different message parts can be specified in the message protection for request on the generator side. WSSSignPart allows for adding a transform algorithm, setting a digest method, setting objects as targets, specifying whether an element, and the signed parts, such as: the SOAP body, the WS-Addressing header, and timestamp information.

For signing information, certain default behaviors occur. The simplest way to use the WSSSignPart API is to use the default behavior (see the example code). The signed parts default behaviors include:

Table 63. Default behavior of signed parts. Several signed part characteristics are configured by default.

Signature decisions	Default behavior
Which SOAP message parts to sign	WebSphere Application Server supports the following SOAP message parts to be signed and used for message protection: • WSSSignature.BODY • WSSSignature.ADDRESSING_HEADERS • WSSSignature.TIMESTAMP
Which digest method to use	Sets the digest algorithm method. The digest method algorithm that is specified within the <DigestMethod> element is used in the <SigningInfo> element. WebSphere Application Server supports the following pre-configured digest methods: • WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1 • WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256 • WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512

Table 63. Default behavior of signed parts (continued). Several signed part characteristics are configured by default.

Signature decisions	Default behavior
Which transform algorithms to use	Adds the transform method. The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following pre-configured transform algorithms: • WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n# • WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2 Use this transform method to ensure compliance with the Basic Security Profile (BSP). • WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform • WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

Procedure

1. To enable Web Services Security by using the WSS API (WSSSignPart), first ensure that the application server is installed.
2. Use the WSSSignPart API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signed parts follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.
   d. Creates WSSSignature from the WSSFactory instance using the SecurityToken.
   e. Creates WSSSignPart from the WSSFactory instance.
   f. Sets the part to be signed and the digest method or transform method specified by step g or step h if the default is not appropriate.
   g. Sets the digest method if the default is not appropriate.
   h. Sets the transform method if the default is not appropriate.
   i. Adds WSSSignPart to WSSSignature. After any WSSSignPart is set to WSSSignature, the default parts to be signed, which are specified in WSSSignature, are ignored.
   j. Adds WSSSignature to WSSGenerationContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

You have completed the steps to configure the signed parts for the generator section of the bindings files. If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses all of methods that are defined in the WSSSignPart API:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance  (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler
      "",
      "dsig-sender.ks",
      "jks",
```

```
      "client".toCharArray(),
      "soaprequester",
      "client".toCharArray(),
      "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token used to the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

// Generate WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();

// Set the part specified by WSSSignPart  (step: f)
   sigPart.setSignPart(WSSSignature.BODY);

// Set the digest method specified by WSSSignPart  (step: g)
   sigPart.setDigestMethod(WSSSignPart.SHA256);

// Set the transform method specified by WSSSignPart  (step: h)
   sigPart.setTransformMethod(WSSSignPart.TRANSFORM_STRT10);

// Add the part specified by WSSSignPart  (step: i)
   sig.addSignPart(sigPart);

// Add the WSSSignature to the WSSGenerationContext  (step: j)
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgcontext);
```

Note: The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

What to do next

Use the WSSVerifyPart API or configure policy sets using the administrative console to verify the signed parts on the consumer side.

Configuring request signing methods for the client:

Use the WSSSignature and WSSSignPart APIs to choose the signing methods. The request signing methods include the signature, canonicalization, digest, and transform methods.

Before you begin

First, you must have specified which parts of the message sent by the client must be digitally signed using the WSS APIs or configuring policy sets using the administrative console.

About this task

The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following website http://www.w3.org/TR/xmldsig-core.

Table 64. Signing methods. Use the signing methods to secure messages.

Name of method	Description
Canonicalization algorithm	Canonicalizes the `<SignedInfo>` element before the information is digested as part of the signature operation.
Signature algorithm	Calculates the signature value of the canonicalized `<SignedInfo>` element. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Transform method	Transforms the parts to be signed before the information is digested as part of the signature operation.
Digest method	Calculates the digest value of the transformed parts. The algorithm selected for the client request sender configuration must match the algorithms selected in the server request receiver configuration.

You can use the WSS APIs or configure policy sets using the administrative console to configure the signing algorithm methods. If using the WSS APIs, use the WSSSignature and WSSSignPart APIs to specify which message parts to digitally sign when configuring the client for request signing.

The WSSSignature and WSSSignPart APIs complete the following steps to configure the signature and signed part algorithm methods:

Procedure

1. For the generator binding, the WSSSignature API specifies the signature method. WebSphere Application Server supports the following pre-configured signature methods:
   - WSSSignature.RSA_SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1

   For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 digital signature method, http://www.w3.org/2000/09/xmldsig#dsa-sha1.
2. For the generator binding, the WSSSignature API specifies the canonicalization method. WebSphere Application Server supports the following pre-configured canonicalization algorithms:
   - WSSSignature.EXC_C14N (the default value): The exclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSSignature.C14N: The inclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-c14n#
3. For the generator binding, the WSSSignPart API specifies the digest method. WebSphere Application Server supports the following pre-configured digest methods:
   - WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1
   - WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
4. For the generator binding, the WSSSignPart API specifies the transform method. WebSphere Application Server supports the following pre-configured transform algorithms:
   - WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:
   - http://www.w3.org/TR/1999/REC-xpath-19991116
   - http://www.w3.org/2002/07/decrypt#XML

Results

Using the WSS APIs, you have specified which algorithm methods are used to digitally sign a message when the client sends a message to a server.

Example

The following example is sample code for specifying the signature information, HMAC_SHA1 as signature method, C14N as a canonicalizaion method, SHA256 as a digest method, and EXC_C14N and TRANSFORM_STRT10 as the transform methods:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();
```

```
//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

//set the part specified by WSSSignPart
WSSSignPart sigPart = factory.newWSSSignPart();

//set the digest method
 // DEFAULT: WSSSignPart.SHA1
sigPart.setDigestMethod(WSSSignPart.SHA256);

//add the transform method
// DEFAULT: WSSSignPart.TRANSFORM_EXC_C14N
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_EXC_C14N);
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_STRT10);

 // add the WSSSignPart to the WSSSignature
 sig.addSignPart(sigPart);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

What to do next

After you configure the client to digitally sign the message and to choose the algorithm methods, you must configure the server to verify the digital signature for request signing and to choose the algorithm methods.

Configure policy sets using the administrative console to configure the signature verification information and methods on the server.

Digital signing methods using the WSSSignature API:

You can configure the signing information for the generator binding using the WSS API. To configure the client for request signing, choose the digital signing methods. The algorithm methods include the signing and canonicalization methods.

You must configure generator signing information to protect message integrity by digitally signing SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

After you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message.

Methods

Methods that are used for the signing information include the:

Signature method
Sets the signature algorithm method.

Canonicalization method
Sets the canonicalization algorithm method.

Signature algorithms

The signature algorithms specify the algorithm that is used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports the following pre-configured algorithms:

Table 65. Signature algorithms. The algorithms include the signing methods.

Algorithm	Description
WSSSignature.HMAC_SHA1	A URI of the signature algorithm, HMAC: http://www.w3.org/2000/09/xmldsig#hmac-sha1
WSSSignature.RSA_SHA1 (the default value)	A URI of the signature algorithm, RSA: http://www.w3.org/2000/09/xmldsig#rsa-sha1

For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 algorithm, http://www.w3.org/2000/09/xmldsig#dsa-sha1

The signing algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

Canonicalization algorithms

The canonicalization algorithms specify the Uniform Resource Identifiers (URI) of the canonicalization method. WebSphere Application Server supports the following pre-configured algorithms:

Table 66. Signature canonicalization algorithms. The algorithms include the canonicalization methods.

Algorithm	Description
WSSSignature.EXC_C14N (the default value)	A URI of the exclusive canonicalization algorithm EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#
WSSSignature.C14N	A URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#

The canonicalization algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

The following example provides sample WSS API code that specifies the HMAC_SHA1 as a signature method and C14n as a canonicalization method:

```
//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
     X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

//generate WSSSignature instance
```

```
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

Signed parts methods using the WSSSignPart API:

You can configure the signed parts information for the generator binding using the WSS API. The algorithms include the digest and transform methods.

You can protect message integrity by configuring signed parts and key information. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

Methods

Methods that are used for the signed parts include the:

Digest method
Sets the digest algorithm method.

Transform algorithm
Sets the transform algorithm method.

Digest algorithms

The digest method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

Table 67. Signed parts digest methods. The methods are used for the signed parts.

Digest method	Description
WSSSignPart.SHA1 (the default value)	A URI of the digest algorithm, SHA1: http://www.w3.org/2000/09/xmldsig#sha1
WSSSignPart.SHA256	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha256
WSSSignPart.SHA512	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha512

Transform algorithms

The transform method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

Table 68. Signed parts transform methods. The methods are used for the signed parts.

Digest method	Description
WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE	A URI of the transform algorithm, enveloped signature: http://www.w3.org/2000/09/xmldsig#enveloped-signature
WSSSignPart.TRANSFORM_STRT10	A URI of the transform algorithm, STR-Transform: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
WSSSignPart.TRANSFORM_EXC_C14N (the default value)	A URI of the transform algorithm, Exc-C14N: http://www.w3.org/2001/10/xml-exc-c14n#

Table 68. Signed parts transform methods (continued). The methods are used for the signed parts.

Digest method	Description
WSSSignPart.TRANSFORM_XPATH2_FILTER	A URI of the transform algorithm, XPath2 filter: http://www.w3.org/2002/06/xmldsig-filter2

The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signed part.

For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:
- http://www.w3.org/TR/1999/REC-xpath-19991116
- http://www.w3.org/2002/07/decrypt#XML

The following example provides sample WSS API code for specifying the signature and signed parts, setting the signing key and adding the STR-Transform transform algorithm as signed parts:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the part specified by WSSSignPart
WSSSignPart sigPart = factory.newWSSSignPart();

//set the part specified by WSSSignPart
sigPart.setSignPart(WSSSignature.BODY);

//set the digest method specified by WSSSignPart
sigPart.setDigestMethod(WSSSignPart.SHA256);

//set the transform method specified by WSSSignPart
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);

//set the part specified by WSSSignPart
sig.addSignPart(sigPart);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

Attaching the generator token using WSS APIs to protect message authenticity:

When you specify the token generator, the information is used on the generator side to generate the security token.

Before you begin

The token processing and pluggable token architecture in the Web Services Security run time reuses the same security token interface and Java Authentication and Authorization Service (JAAS) Login Module

from the Web Services Security APIs (WSS API). The same implementation of token creation and validation can be used in both the WSS API and the WSS SPI in the Web Services Security run time.

Restriction: The `com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent` interface is not used with JAX-WS web services. If you are using JAX-RPC web services, this interface is still valid.

Note that the key name (KeyName) element is not supported in the application server because there is no KeyName policy assertion defined in the current OASIS Web Services Security draft specification.

About this task

The JAAS callback handler (CallbackHandler) and the JAAS login module (LoginModule) are responsible for creating the security token on the generator side and validating (authenticating) the security token on the consumer side.

For example, on the generator side, the Username token is created by the JAAS LoginModule and using the JAAS CallbackHandler to pass the authentication data. The JAAS LoginModule creates the Username SecurityToken object and passes it to the Web Services Security run time.

Then, on the consumer side, the Username Token XML format is passed to the JAAS LoginModule for validation or authentication and the JAAS CallbackHandler is used to pass authentication data from the Web Services Security run time to the LoginModule. After the token is authenticated, a Username SecurityToken object is created and passed it to the Web Services Security run time.

Note: WebSphere Application Server does not support a stackable login module with the WebSphere Application Server default login module implementation, meaning adding the login module before or after the WebSphere Application Server login module implementation. If you want to stack the login module implementations, you must develop the required login modules because there is no default implementation.

The com.ibm.websphere.wssecurity.wssapi.token package provided by WebSphere Application Server includes support for these classes:

- Security token (SecurityTokenImpl)
- Binary security token (BinarySecurityTokenImpl)

In addition, WebSphere Application Server provides the following pre-configured sub-interfaces for security tokens:

- Derived key token
- Security context token (SCT)
- Username token
- LTPA token propagation
- LTPA token
- X509PKCS7 token
- X509PKIPath token
- X509v3 token
- Kerberos v5 token

The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity. The derived key token and the X.509 tokens are used by default for signing and encryption.

The WSS API and WSS SPI are only supported on the client. To specify the security token type on the generator side, you can also configure policy sets using the administrative console. You can also use the WSS APIs or policy sets for matching consumer security tokens.

The default Login Module and Callback implementations are designed to be used as a pair, meaning both a generator and a consumer part. To use the default implementations, select the appropriate generator and consumer security token in a pair. For example, select `system.wss.generate.x509` in the token generator and `system.wss.consume.x509` in the token consumer when the X.509 token is required.

To configure the generator-side security token, use the appropriate pre-configured token generator interface from the WSS APIs to complete the following token configuration process steps:

Procedure

1. Generate the wssFactory instance.
2. Generate the wssGenerationContext instance.

   The WSSGenerationContext interface stores the components for generating Web Services Security (WS-Security), such as the signing and encryption information, the security token, and the time stamp. When the generate() method is called, all of these components are generated.
3. Create the generator-side components, such as the WSSSignature and the WSSEncryption objects.
4. Specify a JAAS configuration by specifying the name of the JAAS login configuration. The Java Authentication and Authorization Service (JAAS) configuration specifies the name of the JAAS configuration. The JAAS configuration specifies how the token logs in on the consumer side. Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module.
5. Specifiy a token generator class name. The token generator class name specifies the required information to generate the SecurityToken. The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity.
6. Specifiy the settings for the callback handler by specifying a callback handler class name and also specifies the callback handler keys. This class name is the name of the callback handler implementation class that is used for the plug-in to the security token framework.

   The callback handler implementation obtains the required security token and passes it to the token generator. The token generator inserts the security token in the Web Services Security header within the SOAP message. Also, the token generator is a plug-in point for the pluggable security token framework. Service providers can provide their own implementation, but the implementation must use the WSSGenerationContext interface.

   WebSphere Application Server provides the following default callback handler implementations for the generator side:

   com.ibm.websphere.wssecurity.callbackhandler.PropertyCallback
   : This class is a callback for handling the name-value pair in elements in the Web Services Security (WS-Security) configuration XMI files.

   com.ibm.websphere.wssecurity.callbackhandler.UNTGUIPromptCallbackHandler
   : This class is a callback handler for the Username token with the GUI prompt on the generator side. This instance is used to set the WSSGenerationContext object to generate a Username token.

   com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler
   : This class is a callback handler for the Username token on the generator side. This instance is used to set into WSSGenerationContext object to attach a Username token. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

   com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler
   : This class is a callback handler that is used to generate the X.509 certificate that is inserted in the Web Services Security header within the SOAP message as a binary security token on the generator side. This instance is used to generate the WSSSignature and WSSEncryption objects, set the objects into the WSSGenerationContext object to generate the X.509 binary

security tokens. A keystore and a key definition are required for this callback handler. If you use this implementation, a key store password, path, and type must be provided on the generator side.

com.ibm.websphere.wssecurity.callbackhandler.LTPAGenerateCallbackHandler
This class is a callback handler for the Lightweight Third Party Authentication (LTPA) tokens on the generator side. This instance is used to generate WSSSignature object and WSSEncryption object to generate a LTPA token.

This callback handler is used to validate the LTPA security token inserted in the Web Services Security header within the SOAP message as a binary security token. However, if the user name and password are specified, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the Run As Subject. Use this callback handler only when the web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client. If you use this implementation, a basic authentication user ID and password must have been provided on the generator side.

com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler
This class is a callback handler for the Kerberos v5 token on the generator side. This instance is used to set the WSSGenerationContext object to generate the Kerberos v5 AP-REQ as a binary security token. The instance is also used to generate the WSSSignature and WSSEncryption objects to use the Kerberos session key or derived key in the SOAP message signature and encryption.

7. If a X.509 token is specified, additional token information is also specified.

Table 69. Information for X.509 token. Use the X.509 token for signing and encryption.

Token Information	Description
storeRef	The reference name of the keystore.
storePath	The keystore file path from which the keystore is loaded, if needed. It is recommended that you use the `${USER_INSTALL_ROOT}` in the path name as this variable expands to the WebSphere Application Server path on your machine. This path is required when you use the X.509 tokens callback handler implementations.
storePassword	The password that is used to check the integrity of the keystore, or the keystore password that is used to unlock the keystore and to access the keystore file. The keystore and its configuration are used for some of the default callback handler implementations that are provided by WebSphere Application Server.
storeType	The keystore type of keystore that is used for the key locator. This selection indicates the format that is used by the keystore file. The following values are available for selection: **JKS** Use this option if the keystore uses the Java Keystore (JKS) format. **JCEKS** Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in WebSphere Application Server. This option provides stronger protection for stored private keys by using Triple DES encryption. **JCERACFKS** Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only). **PKCS11KS (PKCS11)** Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection. **PKCS12KS (PKCS12)** Use this option if your keystore uses the PKCS#12 file format.
alias	The key alias name. The key alias is used by the key locator to find the key within the keystore file.
keyPassword	The key password that is used for recovering the key. This password is needed to access the key object within the keystore file.
keyName	The name of the key. For digital signatures, the key name is used by the request generator or response consumer signing information to determine which key is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption. The key name must be a fully qualified, distinguished name (DN). For example, `CN=Bob,O=IBM,C=US`.
certStores	A list of certificate stores. A collection certificate store includes a list of untrusted, intermediary certificates and certificate revocation lists (CRLs). This step configures a collection certificate store and certificate revocation lists for the generator bindings.

Table 69. Information for X.509 token (continued). Use the X.509 token for signing and encryption.

Token Information	Description
identityAssertion	Specifies whether identity assertion is used. Selects this item if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For an X.509 token generator, the application server sends the original signer certification only.
requestorCertificate	Specifies whether the certificate of the requestor is used.

The following can be specified for a X.509 token:

a. Without any keystore.

b. With a trust anchor. A trust anchor specifies a list of keystore configurations that contain trusted root certificates. These configurations are used to validate the certificate path of incoming X.509-formatted security tokens. For example, when you select the trust anchor or the certificate store of a trusted certificate, you must configure the trust anchor and the certificate store before setting the certificate path.

c. With a keystore that is used for the key locator.

First, you must have created the keystore file, by using a key tool utility, for example. The keystore is used to retrieve the X.509 certificate. This entry specifies the password that is used to access the keystore file. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trustworthiness of a certificate chain.

d. With keystore that is used for the key locator and the trust anchor.

e. With a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3
Specifies an X.509 certificate token.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1
Specifies X.509 certificates in a public key infrastructure (PKI) path. This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used. If you use this implementation, you must provide a key store password, path, and type on this panel.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7
Specifies a list of X.509 certificates and certificate revocation lists in a PKCS#7 format. This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web Services Security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You can specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. If you use this implementation, you must provide a key store password, path, and type.

For some tokens, WebSphere Application Server provides a predefined local name for the value type. When you specify the following local name, you do not need to specify a value type URI:

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
For an LTPA token, you can use `LTPA` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` to be specified for the value type Uniform Resource Identifier (URI).

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
For LTPA token propagation, you can use `LTPA_PROPAGATION` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype` to be specified for the value type Uniform Resource Identifier (URI).

8. If the Username token is specified as the token generator class name, the following token information can be specified:
   a. Whether to use IdentityAssertion option. This option is selected if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For example, WebSphere Application Server sends only the user name of the original caller for a Username token generator.
   b. Whether to use RunAsSubject identity option. This option is used if an identity assertion is defined and you want to use the Run As identity instead of the initial caller identity for identity assertion in a downstream call. This option is valid only if you have configured the Username token as the token generator.
   c. Whether to use sendRealm.
   d. Whether to specify the nonce.

      This option indicates whether a Nonce is included for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of Username tokens. Nonce is valid only when the generated token type is a Username token, and it is available only for the request generator binding.
   e. Specifies the keyword of the time stamp. This option indicates whether to verify a time stamp in the Username token. The time stamp is valid only when the incorporated token type is a Username token.
   f. Specifies a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

      URI value type: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
      Specifies a Username token.
9. If the Kerberos v5 token is specified as the token generator class name, the following token information can be specified:

Token Information	Description	Default Value
name	Kerberos client principal name	
password	Kerberos client password	
realm	Kerberos realm associated with the Kerberos client	Default realm name in Kerberos configuration file. Specify null to use the default value.
targetService	Kerberos service name associated with the target web services.	
targetHost	Kerberos realm name associated with the Kerberos service name.	
tokenValueType	Kerberos token value type in QName defined by Oasis Kerberos Token Profile v1.1 specification.	http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ
targetRealm	Kerberos realm name associated with the Kerberos service name.	Default realm name in the Kerberos configuration file
prompt	A boolean value to enable the login prompt.	false

Token Information	Description	Default Value
supportTokenRequireSHA1	A boolean value to require a SHA1 key that is used in subsequent request messages when the Kerberos token is used as a supporting token.	false SHA1 key is consumed only if the supporting Kerberos token is protected. If set to true, the SHA1 key is always consumed.
alwaysAPREQ	A boolean value to indicate that the client should always send the Kerberos AP_REQ token in the request messages.	false The SHA1 key is used instead in the subsequent messages. If set to true, the Kerberos AP_REQ token is always used.
requireDKT	A boolean value to require a derived key for message protection.	false
clabel	The client label for the derived key.	WS-SecureConversation Specify null to use the default value.
slabel	The service label for the derived key.	WS-SecureConversation Specify null to use the default value.
keylen	The length of the derived key.	16 Specify zero to use the default value
noncelen	The length of the nonce.	16 Specify zero to use the default value
encComponent	An instance of WSSEncryption.	Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component.
sigComponent	An instance of WSSSignature.	Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component.

Additional token value types are defined in the OASIS Kerberos Token Profile v1.1 specification. Specify the token value type as the local name. It is not necessary to specify the value type URI for the Kerberos v5 token.

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

10. If Secure Conversation is used for message protection, the following information must be specified:

Information	Description
bootstrapWSSGenerationContext	The bootstrap configuration used to secure the RequestSecurityToken (RST) token.
bootstrapWSSConmingContext	The bootstrap configuration used for consuming a secured RequestSecurityTokenResponse (RSTR).
ENDPOINT_URL	The service end point URL.
EncryptionAlgorithm	This determines the key size.
cLabel	The client label used when creating the derived key.
sLabel	The server label used when creating the derived key.

11. Set the components into the wssGenerationContext object.
12. Invoke the wssGenerationContext.process() method.

Results

Using the Web Services Security API (WSS API) process, you can configured the token generator.

What to do next

Next, you must specify a similar token consumer configuration.

Configuring generator security tokens using the WSS API:

You can secure the SOAP messages, without using policy sets, by using the Web Services Security APIs. To configure the token on the generator side, use the Web Services Security APIs (WSS API). The generator security tokens are part of the `com.ibm.websphere.wssecurity.wssapi.token` interface package.

Before you begin

The pluggable token framework in WebSphere Application Server has been redesigned so that the same framework from the WSS API can be reused. The same implementation of creating and validating security token can be used both for the Web Services Security runtime and for the WSS API application code. The redesigned framework also simplifies the SPI programming model and will make it easier to add security token types.

You can use the WSS API or you can configure the tokens by using the administrative console. To configure tokens, you must complete the following token tasks:

- Configure the generator tokens.
- Configure the consumer tokens.

About this task

The JAAS CallbackHandler and JAAS LoginModule are responsible for creating the security token on the generator side.

On the generator side, the token is created by using the JAAS LoginModule and by using JAAS CallbackHandler to pass authentication data. Then, the JAAS LoginModule creates the securityToken object, such as the UsernameToken, and passes it to the Web Services Security run time.

On the consumer side, the XML format is passed to the JAAS LoginModule for validation or authentication. then the JAAS CallbackHandler is used to pass authentication data from the Web Services Security runtime to the LoginModule. After the token is authenticated, a security token object is created, and the token is passed it to the Web Services Security runtime.

When using the WSS API for generator token creation, certain default behaviors occur. The simplest way to use the WSS API is to use the default behavior (see the example code). The WSS API provide default values for the token type, the token value, and the JAAS confirmation name. The default token behaviors include:

Table 70. Token decisions and default behaviors. Several token characteristics are configured by default.

Generator token decisions	Default behavior
Which token type to use	The token type specifies which type of token to use for message integrity, message confidentiality, or message authenticity. WebSphere Application Server provides the following pre-configured generator token types for message integrity and message confidentiality: • Derived key token • X509 tokens You can also create custom token types, as needed. WebSphere Application Server also provides the following pre-configured generator token types for the message authenticity: • Username token • LTPA tokens • X509 tokens You can also create custom token types, as needed.
What JAAS login configuration name to specify	The JAAS login configuration name specifies which JAAS login configuration name to use.
Which configuration type to use	The JAAS login module specifies the configuration type. Only the pre-configured generator configuration types can be used for generator token types.

The SecurityToken class (com.ibm.websphere.wssecurity.wssapi.token.SecurityToken) is the generic token class and represents the security token that has methods to get the identity, the XML format, and the cryptographic keys. Using the SecurityToken class, you can apply both the signature and encryption to the SOAP message. However, to apply both, you must have two SecurityToken objects, one for the signature and one for encryption, respectively.

The following tokens types are subclasses of the generic security token class:

Table 71. Subclasses of the SecurityToken. Use the subclasses to represent the security token.

Token type	JAAS login configuration name
Username token	system.wss.generate.unt
Security context token	system.wss.generate.sct
Derived key token	system.wss.generate.dkt

The following tokens types are subclasses of the binary security token class:

Table 72. Subclasses of the BinarySecurityToken. Use the subclasses to represent the binary security token.

Token type	JAAS login configuration name
LTPA token	system.wss.generate.ltpa
LTPA propagation token	system.wss.generate.ltpaProp
X.509 token	system.wss.generate.x509
X.509 PKI Path token	system.wss.generate.pkiPath
X.509 PKCS7 token	system.wss.generate.pkcs7

Note:

- For each JAAS login token generator configuration name, there is a respective token consumer configuration name. For example, for the Username token, the respective token consumer configuration name is system.wss.consume.unt.

- The LTPA and LTPA propagation tokens are only available to a requester that is running as a server-based client. The LTPA and LTPA propagation tokens are not supported for the Java SE 6 or Java EE application client.

Procedure

1. To configure the securityToken package, `com.ibm.websphere.wssecurity.wssapi.token`, first ensure that the application server is installed.
2. Use the Web Services Security token generator process to configure the tokens. For each token type, the process is similar to the following process that demonstrates the UsernameToken token generator process:
   a. Use WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Create the WSSGenerationContext instance from the WSSFactory instance.
   c. Create a JAAS CallbackHandler. The authentication data, such as the user name and password are specified as part of the CallbackHandler. For example, the following code specifies Chris as the user name and sirhC as the password: UNTGenerationCallbackHandler("Chris", "sirhC");
   d. Call any JAAS CallbackHandler parameters and review the token class information for which parameters are required or optional. For example, for the UsernameToken, the following parameters can be configured also:

      Nonce
      Indicates whether a nonce is included in the user name token for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens. The nonce value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.

      Created timestamp
      Indicates whether to insert a time stamp into the UsernameToken. The timestamp value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.
   e. Create the SecurityToken from WSSFactory.

      By default, the UsernameToken API specifies the ValueType as: "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken"

      By default, the UsernameToken API provides the QName of this class and specifies the NamespaceURI as http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd and also specifies the LocalPart as UsernameToken.
   f. Optional: Specify the JAAS login module configuration name. On the generator side, the configuration type is always `generate` (for example, `system.wss.generate.unt`).
   g. Add the SecurityToken to the WSSGenerationContext.
   h. Call WSSGenerationContext.process() and generate the WS-Security header.

Results

If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext process() is called, and the security token for the generator binding is attached.

Example

The following example code shows how to use WSS APIs to create a Username security token, attach the Username token to the SOAP message, and configure the Username token in the generator binding.

```
// import the packages
import javax.xml.ws.BindingProvider;
import com.ibm.websphere.wssecurity.wssapi.*;
import com.ibm.websphere.wssecurity.callbackhandler.*;
...
   // obtain the binding provider
```

```
   BindingProvider bp = ... ;

   // get the request context
   Map<String, Object> reqContext = bp.getRequestContext();

   // generate WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

   // generate WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

   // generate callback handler
   UNTGenerateCallbackHandler untCallbackHandler =
   new UNTGenerateCallbackHandler("Chris", "sirhC");

   // generate the username token
   SecurityToken unt = factory.newSecurityToken(UsernameToken.class, untCallbackHandler);

   // add the SecurityToken to the WSSGenerationContext
   gencont.add(unt);

   // generate the WS-Security header
   gencont.process(reqContext);
```

The following example code shows how to modify the preceding Username token sample to create an LTPAv2 token from the runAs identity on the current thread. The two lines of code that instantiate the callback handler and create the security token are replaced with the following two lines of code:

```
// generate callback handler
LTPAGenerateCallbackHandler ltpaCallbackHandler = new LTPAGenerateCallbackHandler(null, null);

// generate the LTPAv2 token
SecurityToken ltpa = wssfactory.newSecurityToken(LTPAv2Token.class, ltpaCallbackHandler);
```

The instantiation of the LTPAGenerateCallbackHandler object with (null, null) indicates that the LTPA token should be generated from the current runAs identity. If the callback handler is instantiated with basicAuth information, ("userName", "password"), a new LTPA token is created using the specified basicAuth information.

The following example shows how to use secure conversation with the WSS APIs to configure the generator tokens, as well as the consumer tokens. In this example, the SecurityContextToken token is created using the WS-SecureConversation draft namespace: `http://schemas.xmlsoap.org/ws/2005/02/sc/sct`. To use the WS-SecureConversation version 1.3 namespace, `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct`, specify SecurityContextToken13.class instead of SecurityContextToken.class.

```
// import the packages
import javax.xml.ws.BindingProvider;
import com.ibm.websphere.wssecurity.wssapi.*;
import com.ibm.websphere.wssecurity.callbackhandler.*;
...
   // obtain the binding provider
   BindingProvider bp = ... ;

   // get the request context
   Map<String, Object> reqContext = bp.getRequestContext();

   // generate WSSFactory instance
   WSSFactory wssFactory = WSSFactory.getInstance();

   WSSGenerationContext bootstrapGenCon = wssFactory.newWSSGenerationContext();

   // Create a Timestamp
   ...
   // Add Timestamp
   ...

   // Sign the SOAP Body, WS-Addressing headers, and Timestamp
   X509GenerateCallbackHandler btspReqSigCbHandler = new X509GenerateCallbackHandler(...);
   SecurityToken btspReqSigToken = wssFactory.newSecurityToken(X509Token.class,
                                                           btspReqSigCbHandler);
   WSSSignature bootstrapReqSig = wssFactory.newWSSSignature(btspReqSigToken);
   bootstrapReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);

   // Add Sign Parts
   ...
   bootstrapGenCon.add(bootstrapReqSig);

   // Encrypt the SOAP Body and the Signature
   X509GenerateCallbackHandler btspReqEncCbHandler = new X509GenerateCallbackHandler(...);
   SecurityToken btspReqEncToken = wssFactory.newSecurityToken(X509Token.class,
```

```
                                                            btspReqEncCbHandler);
WSSEncryption bootstrapReqEnc = wssFactory.newWSSEncryption(btspReqEncToken);
bootstrapReqEnc.setEncryptionMethod(WSSEncryption.AES128);
bootstrapReqEnc.setKeyEncryptionMethod(WSSEncryption.KW_RSA15);

// Add Encryption parts
...
bootstrapGenCon.add(bootstrapReqEnc);
WSSConsumingContext bootstrapConCon = wssFactory.newWSSConsumingContext();
X509ConsumeCallbackHandler btspRspVfyCbHandler = new X509ConsumeCallbackHandler(....);
WSSVerification bootstrapRspVfy = wssFactory.newWSSVerification(X509Token.class,
                                                                btspRspVfyCbHandler);
bootstrapRspVfy.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

// Add Verify parts
...
bootstrapConCon.add(bootstrapRspVfy);
X509ConsumeCallbackHandler btspRspDecCbHandler = new X509ConsumeCallbackHandler(...);
WSSDecryption bootstrapRspDec = wssFactory.newWSSDecryption(X509Token.class,
                                                            btspRspDecCbHandler);
bootstrapRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
bootstrapRspDec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_RSA15);

// Add Decryption parts
...
bootstrapConCon.add(bootstrapRspDec);
SCTGenerateCallbackHandler sctgch = new SCTGenerateCallbackHandler(bootstrapGenCon,
                                                                   bootstrapConCon,
                                                                   ENDPOINT_URL,
                                                                   WSSEncryption.AES128);
SecurityToken[] scts = wssFactory.newSecurityTokens(new Class[]{SecurityContextToken.class},
                                                    sctgch);
SecurityContextToken sct = (SecurityContextToken)scts[0];

// Use the SCT to generate DKTs for Secure Conversation
// Signature algorithm and client and service labels
DerivedKeyToken dktSig = sct.getDerivedKeyToken(WSSSignature.HMAC_SHA1,
                                                "WS-SecureConversation",
                                                "WS-SecureConversation");

// Encryption algorithm and client and service labels
DerivedKeyToken dktEnc = sct.getDerivedKeyToken(WSSEncryption.AES128,
                                                "WS-SecureConversation",
                                                "WS-SecureConversation");

// Create the application generation context for the request message
WSSGenerationContext applicationGenCon = wssFactory.newWSSGenerationContext();

// Create and add Timestamp
...

// Add the derived key token and Sign the SOAP Body and WS-Addressing headers
WSSSignature appReqSig = wssFactory.newWSSSignature(dktSig);
appReqSig.setSignatureMethod(WSSSignature.HMAC_SHA1);
appReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
...
applicationGenCon.add(appReqSig);

// Add the derived key token and Encrypt the SOAP Body and the Signature
WSSEncryption appReqEnc = wssFactory.newWSSEncryption(dktEnc);
appReqEnc.setEncryptionMethod(WSSEncryption.AES128);
appReqEnc.setTokenReference(SecurityToken.REF_STR);
appReqEnc.encryptKey(false);
...
applicationGenCon.add(appReqEnc);

// Create the application consuming context for the response message
WSSConsumingContext applicationConCon = wssFactory.newWSSConsumingContext();

//client and service labels and decryption algorithm
SCTConsumeCallbackHandler sctCbHandler = new SCTConsumeCallbackHandler("WS-SecureConversation",
                                                                       "WS-SecureConversation",
                                                                       WSSDecryption.AES128);

// Derive the token from SCT and use it to Decrypt the SOAP Body and the Signature
WSSDecryption appRspDec = wssFactory.newWSSDecryption(SecurityContextToken.class,
                                                      sctCbHandler);
appRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
appRspDec.encryptKey(false);
...
applicationConCon.add(appRspDec);

// Derive the token from SCT and use it to Verify the
// signature on the SOAP Body, WS-Addressing headers, and Timestamp
WSSVerification appRspVfy = wssFactory.newWSSVerification(SecurityContextToken.class,
                                                          sctCbHandler);
...
```

```
applicationConCon.add(appRspVfy);
...
applicationGenCon.process(reqContext);
applicationConCon.process(reqContext);
```

What to do next

For each token type, configure the token using the WSS APIs or using the administrative console. Next, specify the similar consumer tokens if you have not done so.

If both the generator and consumer tokens are configured, continue securing SOAP messages either by signing the SOAP message or by encrypting the message, as needed. You can use either the WSS APIs or the administrative console to secure the SOAP messages.

Securing messages at the request generator using WSS APIs:

You can secure SOAP messages by configuring signing information, encryption, and generator tokens to protect message integrity, confidentiality, and authenticity, respectively. This request (client-side) generator configuration defines the Web Services Security requirements for the outgoing SOAP message request.

Before you begin

To secure web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. Therefore, in addition to securing messages at the request generator level, you must also secure messages at the response consumer level.

About this task

The request (client-side) generator configuration requirements involve generating a SOAP message request that uses a digital signature, incorporates encryption, and attaches security tokens.

To secure web service applications, you must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

You can use the following interfaces to configure Web Services Security and to define policy types to secure the SOAP messages:
- Use the administrative console to configure policy sets.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

The following high-level steps use the WSS APIs:

Procedure
- Configure generator signing to protect message integrity.
- Configure encryption to protect message confidentiality.
- Attach generator tokens to protect message authenticity.
- Propagate self-issued SAML bearer tokens using WSS APIs.
- Propagate self-issued SAML sender-vouches tokens with message protection using WSS APIs.
- Propagate self-issued SAML sender-vouches tokens with transport protection using WSS APIs.
- “Sending self-issued SAML holder-of-key tokens with symmetric key using WSS APIs” on page 514.
- “Sending self-issued SAML holder-of-key tokens with asymmetric key using WSS APIs” on page 516.

Results

After completing these procedures, you have secured messages at the request generator level.

What to do next

Next, if not already configured, secure messages with signature verification, decryption, and consumer tokens at the response consumer (client-side) level.

Configuring encryption to protect message confidentiality using the WSS APIs:

You can configure encryption information for the client-side request generator (sender) bindings. Encryption information is used to specify how the generators (senders) encrypt outgoing SOAP messages. To configure encryption, specify which message parts to encrypt and specify which algorithm methods and security tokens are to be used for encryption.

Before you begin

Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. Prior to configuring encryption, familiarize yourself with XML encryption.

About this task

For encryption, you must specify the following:
- Which parts of the message are to be encrypted.
- Which encryption algorithms to specify.

To configure encryption and encrypted parts on the client side, use the WSSEncryption and WSSEncryptPart APIs, or configure policy sets using the administrative console.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

WebSphere Application Server uses encryption information for the default generator to encrypt parts of the SOAP message. The WSSEncryption API configures the following required parts as encrypted parts.

Table 73. Required encrypted parts. Use encrypted parts to increase the confidentiality of SOAP messages.

Encryption parts	Description
Keywords	Keywords are used to add the encrypted parts to the SOAP message.
XPath expression	An XPath expression is used to add the encrypted parts to the SOAP message.
WSSEncryptPart object	This object adds the encrypted parts to the SOAP message.
WSSSignature object	This object adds the signature component as an encrypted part.
Header	This part adds the header in the SOAP header, specified by QName, as an encryption part.
Security token object	This object adds the security token as an encryption part.

Web Services Security API (WSS API) supports symmetric encryption, by using a shared key, only when Web Services Secure Conversation (WS-SecureConversation) is used.

The WSS APIs allow the use of either keywords or an XPath expression to specify the parts of the message that are to be encrypted. WebSphere Application Server supports the use of the following keywords:

Table 74. Supported encryption keywords. Use keywords to specify encrypted parts.

Keyword	References
BODY_CONTENT	The keyword for the contents of the SOAP message body as an encryption target.
SIGNATURE	The keyword for the signature component as an encryption target.

If configuring using the WSS APIs, the WSSEncryption and WSSEncryptPart APIs complete these high-level steps:

Procedure

1. Use the WSSEncryption API to configure encryption. The WSSEncryption API performs these tasks by default:
   a. Generates the callback handler.
   b. Generates the generator security token object.
   c. Adds the security token reference type.
   d. Adds the signature component.
   e. Adds the WSSEncryptPart object.
   f. Adds the parts to be encrypted. Adds the default parts as targets of encryption by using keywords and XPath expressions.
   g. Adds the header in the SOAP message, specified by QName.
   h. Sets the default data encryption method.
   i. Specifies whether the key is to be encrypted using a Boolean value.
   j. Sets the default key encryption method.
   k. Selects a part reference.
   l. Sets the MTOM optimization Boolean value.
2. Use the WSSEncryptPart API to configure encrypted parts or add a transform method. The WSSEncryptPart API performs these tasks by default:
   a. Sets the encrypted parts specified by using keywords or an XPath expression.
   b. Sets the encrypted parts specified by an XPath expression.
   c. Sets the signature component object, WSSSignature.
   d. Sets the header in the SOAP message, specified by QName.
   e. Sets the generator security token.
   f. Adds the transform method, if needed.
3. Change from the default values for algorithm or message parts, as needed. For example: you could change one or more of the following items:
   - Change the data encryption algorithm from the default value of AES 128.
   - Change the key encryption algorithm from the default value of KW_RSA_OAEP.
   - Specify to not encrypt the key (false).
   - Change the security token type from default of X.509 token.
   - Change the security token reference type from the default value of SecurityToken.REF_STR.
   - Only use BODY_CONTENT as an encryption part and not use SIGNATURE also.
   - Turn MTOM optimization on (true).

Results

The encryption information is configured for the generator binding.

Example

The following is an example of the WSSEncryption API:

```
WSSFactory factory = WSSFactory.getInstance();
WSSGenerationContext gencont = factory.newWSSGenerationContext();

X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);
WSSEncryption enc = factory.newWSSEncryption(token);

gencont.add(enc);
```

What to do next

You must configure similar decryption information for the client-side response consumer (receiver) bindings, if you have not already configured the information.

Next, review the WSSEncryption API process.

Encrypting the SOAP message using the WSSEncryption API:

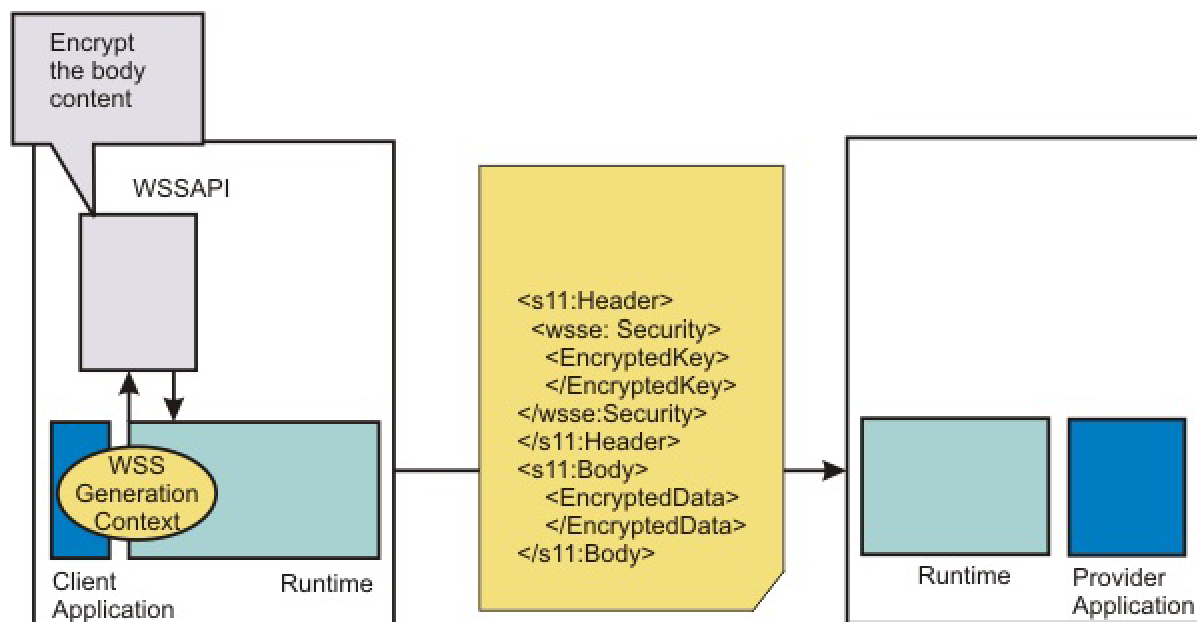
You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the client for request encryption on the generator side, use the WSSEncryption API to encrypt the SOAP message. The WSSEncryption API specifies which request SOAP message parts to encrypt when configuring the client.

Before you begin

You can use the WSS API or use policy sets on the administrative console to enable encryption and add generator security tokens in the SOAP message. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:
- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts, as needed, using the WSSEncryptPart API.

About this task

The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The client generator configuration must match the configuration for the provider consumer.



Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and which message parts to attach encrypted Nonce and timestamp elements to.

The following encryption parts can be configured:

Table 75. Encryption parts. Use the encryption parts to enable encryption in messages.

Encryption parts	Description
part	Adds the WSSEncryptPart object as a target of the encryption part.
keyword	Adds the encryption parts using keywords. WebSphere Application Server supports the following keywords: • BODY_CONTENT • SIGNATURE
xpath	Adds the encryption part using an XPath expression.
signature	Adds the WSSignature component as a target of the encrypted part.
header	Adds the SOAP header, specified by QName, as a target of the encrypted part.
securityToken	Adds the SecurityToken object as a target of the encrypted part.

For encryption, certain default behaviors occur. The simplest way to use the WSSEncryption API is to use the default behavior (see the example code).

WSSEncryption provides defaults for the key encryption algorithm, the data encryption algorithm, the security token reference method, and the encryption parts such as the SOAP body content and the signature. The encryption default behaviors include:

Table 76. Encryption decisions. Use encryption default behavior to secure the message body content and signature.

Encryption decisions	Default behavior
Which SOAP message parts to encrypt using keywords	Sets the encryption parts that you can add using keywords. The default encryption parts are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • WSSEncryption.BODY_CONTENT • WSSEncryption.SIGNATURE

Table 76. Encryption decisions (continued). Use encryption default behavior to secure the message body content and signature.

Encryption decisions	Default behavior
Which data encryption method to choose (algorithm)	Sets the data encryption method. Both data and key encryption methods can be specified. The default data encryption algorithm method is AES 128. WebSphere Application Server supports these data encryption methods: • WSSEncryption.AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc • WSSEncryption.AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc • WSSEncryption.AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc • WSSEncryption.TRIPLE_DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc
Whether to encrypt the key (isEncrypt)	Specifies whether to encrypt the key. The values are `true` or `false`. The default value is to encrypt the key (`true`).
Which key encryption method to choose (algorithm)	Sets the key encryption method. Both data and key encryption methods can be specified. The default key encryption algorithm method is key wrap RSA OAEP. WebSphere Application Server supports these key encryption methods: • WSSEncryption.KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128 • WSSEncryption.KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192 • WSSEncryption.KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256 • WSSEncryption.KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p • WSSEncryption.KW_RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5 • WSSEncryption.KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes
Which security token to specify (securityToken)	Sets the SecurityToken. The default security token type is the X509Token. WebSphere Application Server provides the following pre-configured consumer token types: • Derived key token • X.509 tokens
Which token reference to use (refType)	Sets the type of the security token reference. The default token reference is SecurityToken.REF_KEYID. WebSphere Application Server supports the following token reference types: • SecurityToken.REF_KEYID • SecurityToken.REF_STR • SecurityToken.REF_EMBEDDED • SecurityToken.REF_THUMBPRINT
Whether to use MTOM (mtomOptimize)	Sets Message Transmission Optimization Mechanism (MTOM) optimization for the encrypted part.

Procedure

1. To encrypt the SOAP message using the WSSEncryption API, first ensure that the application server is installed.
2. The WSS API process for encryption performs these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory used for encryption.
   d. Creates WSSEncryption from the WSSFactory instance using the SecurityToken. The default behavior of WSSEncryption is to encrypt the body content and the signature.
   e. Adds a new part to be encrypted in WSSEncryption if the existing part is not appropriate. After addEncryptPart(), addEncryptHeader(), or addEncryptPartByXPath() is called, the default part is cleared.
   f. Calls the encryptKey(false) if the key is not to be encrypted.
   g. Sets the data encryption method if the default method is not appropriate.
   h. Sets the key encryption method if the default method is not appropriate.
   i. Sets the token reference if the default token reference is not appropriate.
   j. Adds WSSEncryption to WSSConsumingContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

If there is an error condition during encryption, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample code using methods that are defined in WSSEncryption:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance (step: b)
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
X509GenerateCallbackHandler callbackHandler = new
        X509GenerateCallbackHandler(
     "",
     "enc-sender.jceks",
     "jceks",
     "storepass".toCharArray(),
     "bob",
     null,
     "CN=Bob, O=IBM, C=US",
     null);

// Generate the security token used for encryption (step: c)
SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);

// Generate WSSEncryption instance (step: d)
WSSEncryption enc = factory.newWSSEncryption(token);

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part specified by the keyword (step: e)
   enc.addEncryptPart(WSSEncryption.BODY_CONTENT);

// Set the part in the SOAP Header specified by QName (step: e)
   enc.addEncryptHeader(new QName("http://www.w3.org/2005/08/addressing",
                                  "MessageID"));

// Set the part specified by WSSSignature (step: e)
   SecurityToken sigToken = getSecurityToken();
      WSSSignature sig = factory.newWSSSignature(sigToken);
   enc.addEncryptPart(sig);

// Set the part specified by SecurityToken (step: e)
   UNTGenerateCallbackHandler untCallbackHandler =
          new UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
                                                   untCallbackHandler);
   enc.addEncryptPart(unt, false);

// sSt the part specified by XPath expression (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
   enc.addEncryptPartByXPath(sb.toString());

// Set whether the key is encrypted (step: f)
// DEFAULT: true
   enc.encryptKey(true);

// Set the data encryption method (step: g)
// DEFAULT: WSSEncryption.AES128
   enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method (step: h)
// DEFAULT: WSSEncryption.KW_RSA_OAEP
   enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Set the token reference (step: i)
// DEFAULT: SecurityToken.REF_KEYID
```

```
  enc.setTokenReference(SecurityToken.REF_STR);

// Add the WSSEncryption to the WSSGenerationContext (step: j)
  gencont.add(enc);

// Process the WS-Security header (step: k)
gencont.process(msgcontext);
```

Note: The X509GenerationCallbackHandler does not need the key password because the public key is used for encryption. You do not need a password to obtain the public key from the Java keystore.

What to do next

If you have not previously specified which encryption methods to choose, use the WSS API or configure the policy sets using the administrative console to choose the data and key encryption algorithm methods.

Choosing encryption methods for generator bindings:

To configure the client for request encryption for the generator binding, you must specify which encryption methods to use when the client encrypts the SOAP messages.

Before you begin

Prior to completing these steps, read the XML encryption information to become familiar with encrypting and decrypting SOAP messages.

To specify which algorithm methods are to be used when the client encrypts the SOAP messages, complete the following tasks:

- Use the WSSEncryption API to configure the data encryption algorithm and the key encryption algorithm methods.
- Use the WSSEncryptPart API to configure a transform algorithm method, if needed. The default is no transform algorithm.

About this task

Some of the encryption-related definitions are based on the XML-Encryption specification. The following information defines some data encryption-related terms:

Data encryption method algorithm

Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. This algorithm encrypts and decrypts data in fixed size, multiple octet blocks.

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Key encryption method algorithm

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the method to encrypt the key that is used to encrypt data. The algorithm represents public key encryption algorithms that are specified for encrypting and decrypting keys.

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property.

The property name is: com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

Table 77. Encryption usage types. The encryption usage types describe encryptions methods.

Usage types	Description
Data encryption	Specifies the algorithm URI that is used for both encrypting and decrypting data. Encrypts and decrypts data in fixed size, multiple octet blocks.
Key encryption	Specifies the algorithm URI that is used for encrypting and decrypting the encryption key.

Data encryption

WebSphere Application Server supports the following pre-configured data encryption algorithms:

Table 78. Data encryption algorithms. These pre-configuring encryption algorithms are supported by WebSphere Application Server.

Data encryption name	Algorithm URI
WSSEncryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSEncryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSEncryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSEncryption.TRIPLE_DES	A URI of data encryption algorithm, 3DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc

Key encryption

WebSphere Application Server supports the following pre-configured key encryption algorithms:

Table 79. Key encryption algorithms. These pre-configured encryption algorithms are supported by WebSphere Application Server.

Key encryption name	Algorithm URI
WSSEncryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128
WSSEncryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSEncryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256

Table 79. Key encryption algorithms (continued). These pre-configured encryption algorithms are supported by WebSphere Application Server.

Key encryption name	Algorithm URI
WSSEncryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSEncryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSEncryption.KW_TRIPLE_DES	http://www.w3.org/2001/04/xmlenc#kw-tripledes

To configure the encryption and encrypted part algorithm methods, use the WSSEncryption API, or configure policy sets using the administrative console.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints that use policy sets.

The WSS API process completes the following high-level steps to specify which encryption methods to use when configuring the client for request encryption:

Procedure

1. Using the WSSEncryption API, adds the required data encryption algorithm. The data encryption algorithm is used for encrypting or decrypting parts of a SOAP message. Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method.

   The client generator configuration must match the configuration for the provider consumer.

   The default data encryption algorithm is AES 128. The data encryption name is AES128, and the URI of the data encryption algorithm, is http://www.w3.org/2001/04/xmlenc#aes128-cbc. WebSphere Application Server supports the following pre-configured data encryption algorithms:

   - AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc

     The AES 128 algorithm is the default data algorithm method.
   - AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc

     Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

     To use this AES 192-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - TRIPLEDES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc
2. As needed, changes the WSSEncryption API method to specify another data encryption algorithm. For example, you might add the following code to change from the default AES 128 algorithm to the Triple DES algorithm:

```
// Default data encryption algorithm: AES128
WSSEncryption enc = factory.newWSSEncryption(x509t);
   enc.setEncryptionMethod(EncryptionMethod.TRIPLEDES_CBC);
gencont.add(enc);
```

3. Using the WSSEncryption API, adds the required key encryption algorithm. The key encryption algorithm is used for encrypting the key that is used for encrypting the message parts within the SOAP message. If the encryption key, which is the key that is used for encrypting the message parts, is not encrypted, then the decryption API selects `false` to match the encryption key.

   The client generator configuration must match the configuration for the provider consumer.

The default key encryption algorithm value is key wrap RSA OAP. The key encryption name is KW_RSA_OAEP, and the URI of the key encryption algorithm is http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p. WebSphere Application Server supports the following pre-configured key encryption algorithms:

- KW AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128
- KW AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192

  To use this key wrap AES 192 algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

  Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).KW AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256

  To use this key wrap AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
- KW RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

  The KW RSA OAEP algorithm is the default key algorithm method.

  When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this algorithm. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5. See more information at http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
- KW RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5
- KW TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

Note: For Web Services Secure Conversation, the WSSEncryption API might specify addition key-related information, such as the:
- algorithmName
- keyLength

Results

If there is an error condition, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample WSS API code using WSSEncryption.setEncryptionMethod() and WSSEncryption.setKeyEncryptionMethod().

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
                                "",
                                "enc-sender.jceks",
                                "jceks",
                                "storepass".toCharArray(),
                                "bob",
                                null,
                                "CN=Bob, O=IBM, C=US",
                                null);

// Generate the security token used for encryption
   SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);
```

```
// Generate WSSEncryption instance
   WSSEncryption enc = factory.newWSSEncryption(token);

// Set the data encryption method
// DEFAULT: WSSEncryption.AES128
      enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
      enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Add the WSSEncryption to the WSSGenerationContext
   gencont.add(enc);

// Generate the WS-Security header
gencont.process(msgcontext);
```

What to do next

Next, if you want to add a transform algorithm, review the WSSEncryptPart API process task.

Encryption methods:

For request generator binding settings, the encryption methods include specifying the data and key encryption algorithms to use to encrypt the SOAP message. The WSS API for encryption (WSSEncryption) specifies the algorithm name and the matching algorithm uniform resource identifier (URI) for the data and key encryption methods. If the data and key encryption algorithms are specified, only elements that are encrypted with those algorithms are accepted.

Data encryption algorithms

The data encryption algorithm is used to encrypt parts of the SOAP message, including the body and the signature. Data encryption algorithms specify the algorithm uniform resource identifier (URI) for each type of data encryption algorithms.

The following pre-configured data encryption algorithms are supported:

Table 80. Data encryption algorithms. The algorithms are used to encrypt SOAP messages.

Data encryption algorithm name	Algorithm URI
WSSEncryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSEncryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSEncryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSEncryption.TRIPLE_DES	A URI of data encryption algorithm, TRIPLE DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

For the AES256-cbc and the AES192-CBC algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The data encryption algorithm configured for encryption for the generator side must match the data encryption algorithm that is configured for decryption for the consumer side.

Key encryption algorithms

This algorithm is used to encrypt and decrypt keys. This key information is used to specify the configuration that is needed to generate the key for digital signature and encryption. The signing information and encryption information configurations can share the key information. The key information on the consumer side is used for specifying the information about the key that is used for validating the digital signature in the received message or for decrypting the encrypted parts of the message. The request generator is configured for the client.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints using the policy sets.

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. The following pre-configured key encryption algorithms are supported:

Table 81. Supported pre-configured key encryption algorithms. The algorithms are used to encrypt and decrypt keys.

WSS API	URI
WSSEncryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128
WSSEncryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSEncryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256
WSSEncryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSEncryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSEncryption.KW_TRIPLE_DES	A URI of key encryption algorithm, key wrap TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

For Secure Conversation, additional key-related information must be specified, such as:

- algorithmName
- keyLength

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The key encryption algorithm for the generator must match the key decryption algorithm that is configured for the consumer.

This example provides sample code for encryption to use the Triple DES for the data encryption method and to use RSA1.5 for the key encryption method:

```
// get the message context
Object msgcontext = getMessageContext();

// generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

// generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "enc-sender.jceks",
  "jceks",
  "storepass".toCharArray(),
  "bob",
  null,
  "CN=Bob, O=IBM, C=US",
  null);

// generate the security token used to the encryption
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

// generate WSSEncryption instance to encrypt the SOAP body content
WSSEncryption enc = factory.newWSSEncryption(token);
enc.addEncryptPart(WSSEncryption.BODY_CONTENT);

// set the data encryption method
// DEFAULT: WSSEncryption.AES128
enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// add the WSSEncryption to the WSSGenerationContext
gencont.add(enc);

// generate the WS-Security header
gencont.process(msgcontext);
```

Adding encrypted parts using the WSSEncryptPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure encrypted parts for the request generator (client side) bindings, use the WSSEncryptPart API to define and add to the listing of elements in the encrypted part. WSSEncryptPart is an interface that is part of the com.ibm.websphere.wssecurity.wssapi.encryption package.

Before you begin

You can use the WSS APIs or configure policy sets using the administrative console to enable the encrypted parts. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:

- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts using the WSSEncryptpart API, as needed.

About this task

Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and

which message parts to attach encrypted elements to. The encryption information on the generator side is used for encrypting an outgoing SOAP message. The request generator is configured for the client.

The WSSEncryptPart API specifies information related to encrypted parts and sets the encrypted parts that have been added for message confidentiality protection. Use the WSSEncryptPart to set the transform method and to specify the part to which the transform method is to be applied. Sets the transform method only if using SOAP with Attachments. The WSSEncryptPart is usually not needed except, in some case for tasks such as setting the transform method.

The encrypted parts and related information displayed in the following table are used to protect the confidentiality of messages.

Table 82. Encrypted parts. Use encrypted parts to secure messages.

Encrypted parts	Description
part	Adds the WSSEncryptPart object as a target of the encryption part.
keyword	Adds the encrypted parts using keywords. The default encryption parts that you can add using keywords are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • BODY_CONTENT • SIGNATURE
xpath	Adds the encrypted part by using an XPath expression.
signature	Adds the WSSSignature component as a target of the encrypted part. WSSSignature is applicable only if the SOAP message contains a signature element.
header	Adds the SOAP header, specified by QName, as a target of the encrypted part.
securityToken	Adds the SecurityToken object as a target of the encrypted part.

For encrypted parts, certain default behaviors occur. The simplest way to use the WSSEncryptPart API is to use the default behavior. The WSSEncryptPart API provides defaults for specifying the transform algorithm, setting objects as targets, specifying the encrypted parts, such as: the SOAP body content and the signature.

The encryption default behaviors include:

Table 83. Encrypted part decisions. Several encrypted message parts are set by default.

Encrypted part decisions	Default behavior
Which SOAP message parts to encrypt using keywords	Specifies which keywords to use for the encrypted parts. WebSphere Application Server sets the following SOAP message parts by default for encryption: • WSSEncryption.BODY_CONTENT • WSSEncryption.SIGNATURE
Which transform method to add	WebSphere Application Server does not specify any transform method by default. Specify a transform method only if using SOAP with Attachments.

Procedure

1. To encrypt the SOAP message parts using the WSSEncryptPart API, first ensure that the application server is installed.
2. The WSS API process using WSSEncryptPart follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the encryption.
   d. Creates WSSEncryption from the WSSFactory instance using SecurityToken.
   e. Creates WSSEncryptPart from WSSFactory.
   f. Adds the parts to be encrypted and to be applied with the transform in WSSEncryptPart. WebSphere Application Server sets these encrypted parts by default for WSSEncryptPart: the BODY_CONTENT and SIGNATURE. After you add other encrypted parts, the default values are no

longer valid. For example, if you call addEncryptPart(securityToken, false), only the security token is encrypted, and not the signature and body content. So if you want to encrypt the security token, the signature, and the body content, you must call addEncryptPart(securityToken, false), addEncryptPart(WSSEncryption.SIGNATURE), and addEncryptPart(WSSEncrypyion.BODY_CONTENT).

g. Sets the transform method.
h. Adds WSSEncryptPart to WSSEncryption.
i. Adds WSSEncryption to WSSGenerationContext.
j. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

If there is an error condition during encryption of the message parts, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

What to do next

After enabling encrypted parts for the request generator (client side) binding, you must specify the same parts to be decrypted for the response consumer (client side) bindings. Next, to configure decryption and decrypted parts, use the WSS APIs or configure policy sets using the administrative console.

Configuring generator signing information to protect message integrity using the WSS APIs:

You can configure the signing information to protect message integrity for the request (client side) generator binding. Signing information includes the signature and the signed parts. To keep the integrity of the message, digital signatures are typically applied.

Before you begin

In addition to using a digital signature and configuring the signing information, the following tasks should also be performed:

- Verify the signing information.
- Incorporate encryption.
- Attach security tokens.

About this task

Integrity refers to digital signature while confidentiality refers to encryption. Integrity is provided by applying a digital signature to a SOAP message. To configure the signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity decreases the risk of data modification when you transmit data across a network.

Also, message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using the signature algorithm methods. The WSS APIs specify which algorithm is to be used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports several pre-configured request signing algorithm methods.

You can use the following interfaces to configure Web Services Security and to protect SOAP message integrity:

- Use the administrative console to configure policy sets for the signing information.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client).

Perform the following signing tasks, using the WSS APIs, to configure the signing information and to protect message integrity for the generator binding.

Procedure

- Configure the signing information using the WSSSignature API. Configure the signing information for the generator binding using the WSSSignature API. Signing information is used to sign parts of a message including the SOAP body, the time stamp, and the WS-Addressing headers. Both signing and encryption can be applied to the same message parts, such as the SOAP body.
- Add or change signed parts using the WSSSignPart API.
- Configure the client for request signing methods using the WSSSignature or WSSSignPart APIs. To configure the client for request signing, choose the signing methods. The request signing methods include the signature, the canonicalization, the digest, and the transform methods. Use the WSSSignature API to configure the signature and canonicalization methods. Use the WSSSignPart API to configure the digest and transform methods.

Results

The WSS APIs also specify the security token for the generator (client) binding and set the type of token reference to protect message authenticity. By completing the steps in these tasks, you have configured generator signing to protect the integrity of the SOAP message.

What to do next

Next, verify the consumer signing information by using the WSS APIs or by configuring policy sets using the administrative console.

Configuring signing information using the WSS APIs:

You can configure the signing information for the client-side request generator (sender) bindings. Signing information is used to sign and validate parts of a message including the SOAP body, the timestamp information, and the Username token. To configure the client for request signing, specify which message parts to digitally sign when configuring the client.

Before you begin

WebSphere Application Server uses XML digital signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method. Prior to completing these steps, familiarize yourself with XML digital signature for signing and verifying digital signatures for digital content.

About this task

By including XML signature in SOAP messages, the following issues are realized: message integrity and authentication. *Integrity* refers to digital signature whereas confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. WebSphere Application Server uses the signing information for the default generator to sign parts of the message, such as the body, time stamp, and Username token.

For the signing information, you must specify the following:

- Which parts of the message are to be signed.
- The key information that is referenced by the key information for the signing keys.
- The signing algorithms.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

The WSSSignature API configures the following parts as signature parts:

Table 84. Pre-configured signature parts. Use the signing information to validate parts of a message.

Part	Description
Security token object	This object authenticates the client. If this option is specified, then the message is signed. You can digitally sign the message using a security token if a login configuration authentication method is selected.
WSSTimestamp object	This object adds a time stamp to a message. The time stamp determines if the message is valid based on the time that the message is sent and then received.
WSSSignature Part object	This object adds the signature parts to a message.
SOAP header and the QName as a target	This signature part adds the header, specified by QName, as a verification part.

The WSS APIs allow the use of keywords or an XPath expression to specify which parts of the message are to be signed. WebSphere Application Server supports the use of the following keywords:

Table 85. Supported signature keywords. Key information is used to specify which parts of a message are signed.

Keyword	References
ADDRESSING_HEADERS	The Web Services Addressing (WS-Addressing) headers.
BODY	The SOAP message body. The body is the user data portion of the message.
TIMESTAMP	The creation and expiration timestamp information.

The Web Services Security API (WSS API) are used to configure the signing information for the request generator (client side) section of the bindings file. To configure the signing information on the client side, use the WSS APIs or configure policy sets for signing using the administrative console.

If configuring using the WSS APIs, the WSSSignature and WSSSignPart APIs complete the following steps to specify which message parts to digitally sign when configuring the client for request generator signing:

Procedure

1. The WSSSignature API adds the required parts of the SOAP message to digitally sign. Either a keyword or an XPath expression can be used to specify the required encryption parts.
2. The WSSSignature API sets the signature method algorithm. The default signature method is RSA_SHA1. WebSphere Application Server supports the following pre-configured algorithms:
   - RSA SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - HMAC SHA1http://www.w3.org/2000/09/xmldsig#hmac-sha1

   WebSphere Application Server does not support the following algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1. You cannot use the DSA-SHA1 algorithm if you want to be compliant with the Basic Security Profile (BSP).

   Any ds:SignatureMethod/@Algorithm element in a signature is based on a symmetric key and must have a value of RSA-SHA1 or HMAC-SHA1.

   The algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.
3. The WSSSignature API sets the canonicalization method. The default signature method is EXC_C14N. WebSphere Application Server supports the following pre-configured algorithms:
   - The URI of the exclusive canonicalization algorithm, EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#.
   - The URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#.

   The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer.
4. The WSSSignature API adds a security token. The API adds information about the security token that is to be used for the signature, such as:
   - The class for security token.

- The callback handler
- The name of the JAAS login configuration.

5. The WSSSignature API sets the type of security token and sets the type of token reference. WebSphere Application Server supports the following pre-configured token references:
   - SecurityToken.REF_STR

     Represents the security token reference as a token reference type.
   - SecurityToken.REF_KEYID

     Represents the key identifier reference as a token reference type.
   - SecurityToken.REF_EMBEDDED

     Represents the embedded reference as a token reference type.
   - SecurityToken.REF_THUMBPRINT

     Represents the thumbprint reference as a token reference type.
6. If SecurityToken.REF_KEYID is set as the type of token reference, the WSSSignature API sets the key information signature type and configures the key information that is referenced by the key information references. WebSphere Application Server supports the following:
   - Specifying that the KeyInfo element is not signed.
   - Specifying that the entire <KeyInfo> element is signed.
   - Specifying that the child elements <Keyinfochildelements> of the <KeyInfo> element are signed.

   If you do not specify one of the previous signature types, WebSphere Application Server specifies that the entire <KeyInfo> element is signed, by default.

   If you select Keyinfo or Keyinfochildelements and you select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm in a subsequent step, WebSphere Application Server also signs the referenced token.

   The key information signature type for the generator must match the signature type for the consumer.
7. The WSSSignature API specifies whether to require signature confirmation. The OASIS Web Services Security (WS-Security) Version 1.1 specification defines the use of signature confirmation. If you are using WS-Security Version 1.0, this function is not available.

   The signature confirmation value is stored in order to validate the signature confirmation with it after the receiving message is returned. This method is called if the response message is expected to attach the signature confirmation into the SOAP message.
8. The WSSSignPart API specifies the part reference. The part reference specifies which parts of the message to digitally sign.

   The part reference refers to the message part that is digitally signed. The part attribute refers to the name of the <Integrity> element when the <PartReference> element is specified for the signature. You can specify multiple <PartReference> elements within the <SigningInfo> element. The <PartReference> element has two child elements when it is specified for the signature verification: <DigestTransform> and <Transform>.
9. The WSSSignPart API specifies the digest method algorithm. The digest method algorithm specified within the <DigestMethod> element is used in the <SigningInfo> element.

   WebSphere Application Server supports the following pre-configured digest algorithms:
   - http://www.w3.org/2000/09/xmldsig#sha1
   - http://www.w3.org/2001/04/xmlenc#sha256
   - http://www.w3.org/2001/04/xmlenc#sha512
10. The WSSSignPart API specifies the transform algorithm. The transform algorithm is that is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following pre-configured transform algorithms:
    - http://www.w3.org/2001/10/xml-exc-c14n#
    - http://www.w3.org/TR/1999/REC-xpath-19991116

Do not use this transform algorithm if you want to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.

- http://www.w3.org/2002/06/xmldsig-filter2
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
- http://www.w3.org/2002/07/decrypt#XML
- http://www.w3.org/2000/09/xmldsig#enveloped-signature

The transform algorithm that you select for the generator must match the transform algorithm that you select for the consumer.

Important: If both of the following conditions are true, WebSphere Application Server signs the referenced token:

- You previously selected the Keyinfo or the Keyinfochildelements option
- You select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm.

11. If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. Configure policy sets using the administrative console to configure the same actor strings for the web service on the server, which processes the request and sends the response back.

    The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

Results

After the WSSSignature and WSSSignPart APIs complete these steps, the signing information is configured for the generator sections of the bindings files.

Example

The following example shows WSS API sample code to configure the signature, to generate the callback handler, and to specify the X.509 token type as the security token:

```
WSSFactory factory = WSSFactory.getInstance();
// Instantiate a generation context
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler and specify the X.509 token
   X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
   SecurityToken token = factory.newSecurityToken(X509Token.class,
                                                  callbackHandler);

// Set the signature information
WSSSignature sig = factory.newWSSSignature(token);
// Add the header using QName
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "To"));
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "MessageID"));
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "Action"));
// Apply the signature
   gencont.add(sig);

// Secure the message
gencont.process(msgctx);
```

What to do next

You must configure similar signature information for the client-side request consumer (receiver) bindings by completing the following verification tasks:

- Verify the signature
- Choose the signature algorithm methods.
- Change or add signed parts, as needed.

If signature verification is already configured, configure the encryption and decryption information, or configure the consumer and generator tokens.

Configuring signing information using the WSSSignature API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the signing information for the generator binding sections for the client-side request, use the WSSSignature API. The WSSSignature API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to enable the signing information. To secure SOAP messages, you must complete the following signing tasks:

- Configure the signing information.
- Choose the signing methods.
- Add or change signed parts, as needed.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing algorithms such as RSA-SHA1 and HMAC-SHA1.

XML signature defines many methods for describing key information and enables the definition of a new method. XML canonicalization (C14N) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. The C14N process is used to canonicalize XML information. Select an appropriate C14N algorithm because the information that is canonicalized depends on this algorithm.

The signing information specifies the integrity constraints that are applied to generated messages. The constraints include specifying which message parts within the generated message must be digitally signed, and the message parts to attach digitally signed Nonce and timestamp elements to. The following signature and related signature part information are configured:

Table 86. Signature parts information. Use the signature parts to secure messages.

signature parts	Description
keyword	Adds a signature part using keywords. Use the following keywords for the signature parts: • ADDRESSING_HEADERS • BODY • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds a signature part by using an XPath expression.
part	Adds a WSSSignPart object as a target of the signature part.
timestamp	Adds a WSSTimestamp object as a target of the signature part. When specified, the timestamp information also specifies when the message is generated and when it expires.

Table 86. Signature parts information (continued). Use the signature parts to secure messages.

signature parts	Description
header	Adds the header, specified by QName, as a target of the signature part.
securityToken	Adds a SecurityToken object as a target of the signature part.

For signing information, certain default behaviors occur. The simplest way to use the WSSSignature API is to use the default behavior (see the example code). The default values are defined by the WSS API for the signing method, the canonicalization method, the security token references, and the signature parts.

Table 87. Signature default behaviors. Several signature behaviors are configured by default.

Signature decisions	Default behavior
Which keywords to use	Sets the keywords. WebSphere Application Server supports the following keywords by default: • ADDRESSING_HEADERS • BODY • TIMESTAMP
Which signature method to use	Sets the signature algorithm. The default signature method is RSA SHA1. WebSphere Application Server supports the following pre-configured signature methods: • WSSSignature.RSA_SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1 • WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1 The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported.
Which canonicalization method to use	Sets the canonicalization algorithm. The default canonicalization method is EXC C14N. WebSphere Application Server supports the following pre-configured canonicalization methods: • WSSSignature.EXC_C14N; http://www.w3.org/2001/10/xml-exc-c14n# • WSSSignature.C14N: http://www.w3.org/2001/10/xml-c14n#
Whether signature confirmation is required	Sets whether to require signature confirmation. The default value is **false**. Signature confirmation is defined in the OASIS Web Services Security Version 1.1 specification. If required, the value of your signature confirmation is stored in order to use it to validate the signature confirmation after receiving back the message that generated the signature confirmation in the response message. This method is for the requestor side.
Which security token to use	Sets the SecurityToken. The token type specifies which type of token to use for signing and validating messages. The X.509 token is the default token type. WebSphere Application Server provides the following pre-configured consumer token types: • Derived Key Token • X509 tokens You can also create custom token types, as needed.
Which token reference to set	Sets the refType. SecurityToken.REF_STR is the default value for the type of token reference. WebSphere Application Server supports these pre-configured token references types: • SecurityToken.REF_STR • SecurityToken.REF_KEYID • SecurityToken.REF_EMBEDDED • SecurityToken.REF_THUMBPRINT

If WSSSignature.requireSignatureConfirmation() is called, then the WSSSignature API expects that the response message will include the signature confirmation.

Procedure

1. To configure the signing information in a SOAP message by using the WSS API, first ensure that the application server is installed.
2. Use the WSSSignature API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signature follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance. WSSGenerationContext must be called in a JAX-WS client application.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.

d. Creates WSSSignature from the WSSFactory instance using the SecurityToken. The default behavior of WSSSignature is to sign these signature parts: BODY, ADDRESSING_HEADERS, and TIMESTAMP.
e. Adds the part to be signed, if the default part is not appropriate. If the digest method or transform method is changed, creates WSSSignPart and add it to WSSSignature.
f. Creates WSSSignaturePart to WSSSignature. Calls the requiredSignatureConfirmation() method, if the signature confirmation is to be applied.
g. Sets the canonicalization method, if the default is not appropriate.
h. Sets the signature method, if the default is not appropriate.
i. Sets the token reference, if the default is not appropriate.
j. Adds WSSSignature to WSSGenerationContext.
k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

You have completed the steps to configure the signature for the generator section of the bindings. If there is an error condition when signing the message parts, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses methods that are defined in the WSSignature API.

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the com.ibm.websphere.wssecurity.wssapi.WSSFactory instance  (step: a)
   WSSFactory factory = com.ibm.websphere.wssecurity.wssapi.WSSFactory.getInstance();

// Generate the WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
       "",
       "dsig-sender.ks",
       "jks",
       "client".toCharArray(),
       "soaprequester",
       "client".toCharArray(),
       "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token to be used for the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class,
        callbackHandler);

// Generate the WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part to be signed  (step: e)
// DEFAULT: WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS,
//          and WSSSignature.TIMESTAMP.

// Set the part in the SOAP Header specified by QName  (step: e)
      sig.addSignHeader(new
                        QName("http://www.w3.org/2005/08/addressing",
                        "MessageID"));

// Set the part specified by the keyword  (step: e)
      sig.addSignPart(WSSSignature.BODY);

// Set the part specified by SecurityToken  (step: e)
   UNTGenerateCallbackHandler untCallbackHandler = new
      UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
         untCallbackHandler);
      sig.addSignPart(unt);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();
      sigPart.setSignPart(WSSSignature.TIMESTAMP);
      sigPart.setDigestMethod(WSSSignPart.SHA256);
      sig.addSignPart(sigPart);
```

```
// Set the part specified by WSSTimestamp  (step: e)
   WSSTimestamp timestamp = factory.newWSSTimestamp();
      sig.addSignPart(timestamp);

// Set the part specified by XPath expression  (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
      sig.addSignPartByXPath(sb.toString());

// Set to apply the signature confirmation  (step: f)
      sig.requireSignatureConfirmation();

// Set the canonicalization method  (step: g)
// DEFAULT: WSSSignature.EXC_C14N
      sig.setCanonicalizationMethod(WSSSignature.C14N);

// Set the signature method  (step: h)
// DEFAULT: WSSSignature.RSA_SHA1
      sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

// Set the token reference  (step: i)
// DEFAULT: SecurityToken.REF_STR
      sig.setTokenReference(SecurityToken.REF_KEYID);

// Add the WSSSignature to WSSGenerationContext  (step: j)
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgctx);
```

Note: The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

What to do next

Next, chose the algorithm methods if you want a method that is different from the default values. If the algorithm methods do not need to be changed, next use the WSSVerification API to verify the signature and specify the algorithm methods in the consumer section of the binding. Note that the WSSVerification API is only supported on the response consumer (client side).

Adding signed parts using the WSSSignPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure parts to be signed for the request generator (client side) bindings, use the WSSSignPart API to protect the integrity of messages and to configure the digest and transform algorithm methods. The WSSSignPart API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to configure the signing information. To secure SOAP messages using the signing information, you must complete one of the following tasks:
- Configure the signature information
- Configure signed parts, as needed.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing digest and transform algorithms (for example, SHA1 or TRANSFORM_EXC_C14N).

The signing information specifies the integrity constraints that are applied to generated messages. The signed parts are used to protect the integrity of messages. You can specify the signed parts to add for message integrity protection.

The following table shows the required signed parts when the digital signature security constraint (integrity) is defined:

Table 88. Signed parts information. Use the signed parts to secure messages.

Signed parts	Description
keyword	Adds signed parts using keywords. WebSphere Application Server supports the following keywords for signed parts: • BODY • ADDRESSING_HEADERS • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds the required signed parts by using an XPath expression.
header	Adds the header, specified by QName, as a signed part.
timestamp	Adds a WSSTimestamp object as a signed part. If specified, the timestamp information specifies when the message is generated and when it expires.

Different message parts can be specified in the message protection for request on the generator side. WSSSignPart allows for adding a transform algorithm, setting a digest method, setting objects as targets, specifying whether an element, and the signed parts, such as: the SOAP body, the WS-Addressing header, and timestamp information.

For signing information, certain default behaviors occur. The simplest way to use the WSSSignPart API is to use the default behavior (see the example code). The signed parts default behaviors include:

Table 89. Default behavior of signed parts. Several signed part characteristics are configured by default.

Signature decisions	Default behavior
Which SOAP message parts to sign	WebSphere Application Server supports the following SOAP message parts to be signed and used for message protection: • WSSSignature.BODY • WSSSignature.ADDRESSING_HEADERS • WSSSignature.TIMESTAMP
Which digest method to use	Sets the digest algorithm method. The digest method algorithm that is specified within the <DigestMethod> element is used in the <SigningInfo> element. WebSphere Application Server supports the following pre-configured digest methods: • WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1 • WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256 • WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
Which transform algorithms to use	Adds the transform method. The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following pre-configured transform algorithms: • WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n# • WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2 Use this transform method to ensure compliance with the Basic Security Profile (BSP). • WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform • WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

Procedure

1. To enable Web Services Security by using the WSS API (WSSSignPart), first ensure that the application server is installed.

2. Use the WSSSignPart API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signed parts follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.
   d. Creates WSSSignature from the WSSFactory instance using the SecurityToken.
   e. Creates WSSSignPart from the WSSFactory instance.
   f. Sets the part to be signed and the digest method or transform method specified by step g or step h if the default is not appropriate.
   g. Sets the digest method if the default is not appropriate.
   h. Sets the transform method if the default is not appropriate.
   i. Adds WSSSignPart to WSSSignature. After any WSSSignPart is set to WSSSignature, the default parts to be signed, which are specified in WSSSignature, are ignored.
   j. Adds WSSSignature to WSSGenerationContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

You have completed the steps to configure the signed parts for the generator section of the bindings files. If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses all of methods that are defined in the WSSSignPart API:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance  (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler
      "",
      "dsig-sender.ks",
      "jks",
      "client".toCharArray(),
      "soaprequester",
      "client".toCharArray(),
      "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token used to the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

// Generate WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();

// Set the part specified by WSSSignPart  (step: f)
   sigPart.setSignPart(WSSSignature.BODY);

// Set the digest method specified by WSSSignPart  (step: g)
   sigPart.setDigestMethod(WSSSignPart.SHA256);

// Set the transform method specified by WSSSignPart  (step: h)
   sigPart.setTransformMethod(WSSSignPart.TRANSFORM_STRT10);

// Add the part specified by WSSSignPart  (step: i)
   sig.addSignPart(sigPart);

// Add the WSSSignature to the WSSGenerationContext  (step: j)
```

```
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgcontext);
```

Note: The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

What to do next

Use the WSSVerifyPart API or configure policy sets using the administrative console to verify the signed parts on the consumer side.

Configuring request signing methods for the client:

Use the WSSSignature and WSSSignPart APIs to choose the signing methods. The request signing methods include the signature, canonicalization, digest, and transform methods.

Before you begin

First, you must have specified which parts of the message sent by the client must be digitally signed using the WSS APIs or configuring policy sets using the administrative console.

About this task

The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following website http://www.w3.org/TR/xmldsig-core.

Table 90. Signing methods. Use the signing methods to secure messages.

Name of method	Description
Canonicalization algorithm	Canonicalizes the `<SignedInfo>` element before the information is digested as part of the signature operation.
Signature algorithm	Calculates the signature value of the canonicalized `<SignedInfo>` element. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Transform method	Transforms the parts to be signed before the information is digested as part of the signature operation.
Digest method	Calculates the digest value of the transformed parts. The algorithm selected for the client request sender configuration must match the algorithms selected in the server request receiver configuration.

You can use the WSS APIs or configure policy sets using the administrative console to configure the signing algorithm methods. If using the WSS APIs, use the WSSSignature and WSSSignPart APIs to specify which message parts to digitally sign when configuring the client for request signing.

The WSSSignature and WSSSignPart APIs complete the following steps to configure the signature and signed part algorithm methods:

Procedure

1. For the generator binding, the WSSSignature API specifies the signature method. WebSphere Application Server supports the following pre-configured signature methods:
   - WSSSignature.RSA_SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1

   For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 digital signature method, http://www.w3.org/2000/09/xmldsig#dsa-sha1.
2. For the generator binding, the WSSSignature API specifies the canonicalization method. WebSphere Application Server supports the following pre-configured canonicalization algorithms:

- WSSSignature.EXC_C14N (the default value): The exclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-exc-c14n#
- WSSSignature.C14N: The inclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-c14n#

3. For the generator binding, the WSSSignPart API specifies the digest method. WebSphere Application Server supports the following pre-configured digest methods:
   - WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1
   - WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
4. For the generator binding, the WSSSignPart API specifies the transform method. WebSphere Application Server supports the following pre-configured transform algorithms:
   - WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:
   - http://www.w3.org/TR/1999/REC-xpath-19991116
   - http://www.w3.org/2002/07/decrypt#XML

Results

Using the WSS APIs, you have specified which algorithm methods are used to digitally sign a message when the client sends a message to a server.

Example

The following example is sample code for specifying the signature information, HMAC_SHA1 as signature method, C14N as a canonicalizaion method, SHA256 as a digest method, and EXC_C14N and TRANSFORM_STRT10 as the transform methods:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);
```

```
//set the part specified by WSSSignPart
WSSSignPart sigPart = factory.newWSSSignPart();

//set the digest method
 // DEFAULT: WSSSignPart.SHA1
sigPart.setDigestMethod(WSSSignPart.SHA256);

//add the transform method
// DEFAULT: WSSSignPart.TRANSFORM_EXC_C14N
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_EXC_C14N);
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_STRT10);

 // add the WSSSignPart to the WSSSignature
 sig.addSignPart(sigPart);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

What to do next

After you configure the client to digitally sign the message and to choose the algorithm methods, you must configure the server to verify the digital signature for request signing and to choose the algorithm methods.

Configure policy sets using the administrative console to configure the signature verification information and methods on the server.

Digital signing methods using the WSSSignature API:

You can configure the signing information for the generator binding using the WSS API. To configure the client for request signing, choose the digital signing methods. The algorithm methods include the signing and canonicalization methods.

You must configure generator signing information to protect message integrity by digitally signing SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

After you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message.

Methods

Methods that are used for the signing information include the:

Signature method
: Sets the signature algorithm method.

Canonicalization method
: Sets the canonicalization algorithm method.

Signature algorithms

The signature algorithms specify the algorithm that is used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports the following pre-configured algorithms:

Table 91. Signature algorithms. The algorithms include the signing methods.

Algorithm	Description
WSSSignature.HMAC_SHA1	A URI of the signature algorithm, HMAC: http://www.w3.org/2000/09/xmldsig#hmac-sha1
WSSSignature.RSA_SHA1 (the default value)	A URI of the signature algorithm, RSA: http://www.w3.org/2000/09/xmldsig#rsa-sha1

For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 algorithm, http://www.w3.org/2000/09/xmldsig#dsa-sha1

The signing algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

Canonicalization algorithms

The canonicalization algorithms specify the Uniform Resource Identifiers (URI) of the canonicalization method. WebSphere Application Server supports the following pre-configured algorithms:

Table 92. Signature canonicalization algorithms. The algorithms include the canonicalization methods.

Algorithm	Description
WSSSignature.EXC_C14N (the default value)	A URI of the exclusive canonicalization algorithm EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#
WSSSignature.C14N	A URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#

The canonicalization algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

The following example provides sample WSS API code that specifies the HMAC_SHA1 as a signature method and C14n as a canonicalization method:

```
//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
     X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

Signed parts methods using the WSSSignPart API:

You can configure the signed parts information for the generator binding using the WSS API. The algorithms include the digest and transform methods.

You can protect message integrity by configuring signed parts and key information. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

Methods

Methods that are used for the signed parts include the:

Digest method
Sets the digest algorithm method.

Transform algorithm
Sets the transform algorithm method.

Digest algorithms

The digest method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

Table 93. Signed parts digest methods. The methods are used for the signed parts.

Digest method	Description
WSSSignPart.SHA1 (the default value)	A URI of the digest algorithm, SHA1: http://www.w3.org/2000/09/xmldsig#sha1
WSSSignPart.SHA256	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha256
WSSSignPart.SHA512	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha512

Transform algorithms

The transform method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

Table 94. Signed parts transform methods. The methods are used for the signed parts.

Digest method	Description
WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE	A URI of the transform algorithm, enveloped signature: http://www.w3.org/2000/09/xmldsig#enveloped-signature
WSSSignPart.TRANSFORM_STRT10	A URI of the transform algorithm, STR-Transform: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
WSSSignPart.TRANSFORM_EXC_C14N (the default value)	A URI of the transform algorithm, Exc-C14N: http://www.w3.org/2001/10/xml-exc-c14n#
WSSSignPart.TRANSFORM_XPATH2_FILTER	A URI of the transform algorithm, XPath2 filter: http://www.w3.org/2002/06/xmldsig-filter2

The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signed part.

For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:

- http://www.w3.org/TR/1999/REC-xpath-19991116
- http://www.w3.org/2002/07/decrypt#XML

The following example provides sample WSS API code for specifying the signature and signed parts, setting the signing key and adding the STR-Transform transform algorithm as signed parts:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler(
```

```
    "",
    "dsig-sender.ks",
    "jks",
    "client".toCharArray(),
    "soaprequester",
    "client".toCharArray(),
    "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
    null);

  //generate the security token used to the signature
  SecurityToken token = factory.newSecurityToken(X509Token.class,
       callbackHandler);

  //generate WSSSignature instance
  WSSSignature sig = factory.newWSSSignature(token);

  //set the part specified by WSSSignPart
  WSSSignPart sigPart = factory.newWSSSignPart();

  //set the part specified by WSSSignPart
  sigPart.setSignPart(WSSSignature.BODY);

  //set the digest method specified by WSSSignPart
  sigPart.setDigestMethod(WSSSignPart.SHA256);

  //set the transform method specified by WSSSignPart
  sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);

  //set the part specified by WSSSignPart
  sig.addSignPart(sigPart);

  //add the WSSSignature to the WSSGenerationContext
  gencont.add(sig);

  //generate the WS-Security header
  gencont.process(msgcontext);
```

Attaching the generator token using WSS APIs to protect message authenticity:

When you specify the token generator, the information is used on the generator side to generate the security token.

Before you begin

The token processing and pluggable token architecture in the Web Services Security run time reuses the same security token interface and Java Authentication and Authorization Service (JAAS) Login Module from the Web Services Security APIs (WSS API). The same implementation of token creation and validation can be used in both the WSS API and the WSS SPI in the Web Services Security run time.

Restriction: The `com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent` interface is not used with JAX-WS web services. If you are using JAX-RPC web services, this interface is still valid.

Note that the key name (KeyName) element is not supported in the application server because there is no KeyName policy assertion defined in the current OASIS Web Services Security draft specification.

About this task

The JAAS callback handler (CallbackHandler) and the JAAS login module (LoginModule) are responsible for creating the security token on the generator side and validating (authenticating) the security token on the consumer side.

For example, on the generator side, the Username token is created by the JAAS LoginModule and using the JAAS CallbackHandler to pass the authentication data. The JAAS LoginModule creates the Username SecurityToken object and passes it to the Web Services Security run time.

Then, on the consumer side, the Username Token XML format is passed to the JAAS LoginModule for validation or authentication and the JAAS CallbackHandler is used to pass authentication data from the Web Services Security run time to the LoginModule. After the token is authenticated, a Username SecurityToken object is created and passed it to the Web Services Security run time.

Note: WebSphere Application Server does not support a stackable login module with the WebSphere Application Server default login module implementation, meaning adding the login module before or after the WebSphere Application Server login module implementation. If you want to stack the login module implementations, you must develop the required login modules because there is no default implementation.

The com.ibm.websphere.wssecurity.wssapi.token package provided by WebSphere Application Server includes support for these classes:

- Security token (SecurityTokenImpl)
- Binary security token (BinarySecurityTokenImpl)

In addition, WebSphere Application Server provides the following pre-configured sub-interfaces for security tokens:

- Derived key token
- Security context token (SCT)
- Username token
- LTPA token propagation
- LTPA token
- X509PKCS7 token
- X509PKIPath token
- X509v3 token
- Kerberos v5 token

The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity. The derived key token and the X.509 tokens are used by default for signing and encryption.

The WSS API and WSS SPI are only supported on the client. To specify the security token type on the generator side, you can also configure policy sets using the administrative console. You can also use the WSS APIs or policy sets for matching consumer security tokens.

The default Login Module and Callback implementations are designed to be used as a pair, meaning both a generator and a consumer part. To use the default implementations, select the appropriate generator and consumer security token in a pair. For example, select `system.wss.generate.x509` in the token generator and `system.wss.consume.x509` in the token consumer when the X.509 token is required.

To configure the generator-side security token, use the appropriate pre-configured token generator interface from the WSS APIs to complete the following token configuration process steps:

Procedure

1. Generate the wssFactory instance.
2. Generate the wssGenerationContext instance.

   The WSSGenerationContext interface stores the components for generating Web Services Security (WS-Security), such as the signing and encryption information, the security token, and the time stamp. When the generate() method is called, all of these components are generated.
3. Create the generator-side components, such as the WSSSignature and the WSSEncryption objects.
4. Specify a JAAS configuration by specifying the name of the JAAS login configuration. The Java Authentication and Authorization Service (JAAS) configuration specifies the name of the JAAS configuration. The JAAS configuration specifies how the token logs in on the consumer side. Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module.

5. Specifiy a token generator class name. The token generator class name specifies the required information to generate the SecurityToken. The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity.
6. Specifiy the settings for the callback handler by specifying a callback handler class name and also specifies the callback handler keys. This class name is the name of the callback handler implementation class that is used for the plug-in to the security token framework.

   The callback handler implementation obtains the required security token and passes it to the token generator. The token generator inserts the security token in the Web Services Security header within the SOAP message. Also, the token generator is a plug-in point for the pluggable security token framework. Service providers can provide their own implementation, but the implementation must use the WSSGenerationContext interface.

   WebSphere Application Server provides the following default callback handler implementations for the generator side:

   com.ibm.websphere.wssecurity.callbackhandler.PropertyCallback
   This class is a callback for handling the name-value pair in elements in the Web Services Security (WS-Security) configuration XMI files.

   com.ibm.websphere.wssecurity.callbackhandler.UNTGUIPromptCallbackHandler
   This class is a callback handler for the Username token with the GUI prompt on the generator side. This instance is used to set the WSSGenerationContext object to generate a Username token.

   com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler
   This class is a callback handler for the Username token on the generator side. This instance is used to set into WSSGenerationContext object to attach a Username token. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

   com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler
   This class is a callback handler that is used to generate the X.509 certificate that is inserted in the Web Services Security header within the SOAP message as a binary security token on the generator side. This instance is used to generate the WSSSignature and WSSEncryption objects, set the objects into the WSSGenerationContext object to generate the X.509 binary security tokens. A keystore and a key definition are required for this callback handler. If you use this implementation, a key store password, path, and type must be provided on the generator side.

   com.ibm.websphere.wssecurity.callbackhandler.LTPAGenerateCallbackHandler
   This class is a callback handler for the Lightweight Third Party Authentication (LTPA) tokens on the generator side. This instance is used to generate WSSSignature object and WSSEncryption object to generate a LTPA token.

   This callback handler is used to validate the LTPA security token inserted in the Web Services Security header within the SOAP message as a binary security token. However, if the user name and password are specified, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the Run As Subject. Use this callback handler only when the web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client. If you use this implementation, a basic authentication user ID and password must have been provided on the generator side.

   com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler
   This class is a callback handler for the Kerberos v5 token on the generator side. This instance is used to set the WSSGenerationContext object to generate the Kerberos v5 AP-REQ as a binary security token. The instance is also used to generate the WSSSignature and WSSEncryption objects to use the Kerberos session key or derived key in the SOAP message signature and encryption.
7. If a X.509 token is specified, additional token information is also specified.

Table 95. Information for X.509 token. Use the X.509 token for signing and encryption.

Token Information	Description
storeRef	The reference name of the keystore.
storePath	The keystore file path from which the keystore is loaded, if needed. It is recommended that you use the `${USER_INSTALL_ROOT}` in the path name as this variable expands to the WebSphere Application Server path on your machine. This path is required when you use the X.509 tokens callback handler implementations.
storePassword	The password that is used to check the integrity of the keystore, or the keystore password that is used to unlock the keystore and to access the keystore file. The keystore and its configuration are used for some of the default callback handler implementations that are provided by WebSphere Application Server.
storeType	The keystore type of keystore that is used for the key locator. This selection indicates the format that is used by the keystore file. The following values are available for selection: **JKS** Use this option if the keystore uses the Java Keystore (JKS) format. **JCEKS** Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in WebSphere Application Server. This option provides stronger protection for stored private keys by using Triple DES encryption. **JCERACFKS** Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only). **PKCS11KS (PKCS11)** Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection. **PKCS12KS (PKCS12)** Use this option if your keystore uses the PKCS#12 file format.
alias	The key alias name. The key alias is used by the key locator to find the key within the keystore file.
keyPassword	The key password that is used for recovering the key. This password is needed to access the key object within the keystore file.
keyName	The name of the key. For digital signatures, the key name is used by the request generator or response consumer signing information to determine which key is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption. The key name must be a fully qualified, distinguished name (DN). For example, `CN=Bob,O=IBM,C=US`.
certStores	A list of certificate stores. A collection certificate store includes a list of untrusted, intermediary certificates and certificate revocation lists (CRLs). This step configures a collection certificate store and certificate revocation lists for the generator bindings.
identityAssertion	Specifies whether identity assertion is used. Selects this item if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For an X.509 token generator, the application server sends the original signer certification only.
requestorCertificate	Specifies whether the certificate of the requestor is used.

The following can be specified for a X.509 token:

a. Without any keystore.

b. With a trust anchor. A trust anchor specifies a list of keystore configurations that contain trusted root certificates. These configurations are used to validate the certificate path of incoming X.509-formatted security tokens. For example, when you select the trust anchor or the certificate store of a trusted certificate, you must configure the trust anchor and the certificate store before setting the certificate path.

c. With a keystore that is used for the key locator.

   First, you must have created the keystore file, by using a key tool utility, for example. The keystore is used to retrieve the X.509 certificate. This entry specifies the password that is used to access the keystore file. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trustworthiness of a certificate chain.

d. With keystore that is used for the key locator and the trust anchor.

e. With a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

   ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3
   Specifies an X.509 certificate token.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1
: Specifies X.509 certificates in a public key infrastructure (PKI) path. This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used. If you use this implementation, you must provide a key store password, path, and type on this panel.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7
: Specifies a list of X.509 certificates and certificate revocation lists in a PKCS#7 format. This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web Services Security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You can specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. If you use this implementation, you must provide a key store password, path, and type.

For some tokens, WebSphere Application Server provides a predefined local name for the value type. When you specify the following local name, you do not need to specify a value type URI:

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
: For an LTPA token, you can use `LTPA` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` to be specified for the value type Uniform Resource Identifier (URI).

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
: For LTPA token propagation, you can use `LTPA_PROPAGATION` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype` to be specified for the value type Uniform Resource Identifier (URI).

8. If the Username token is specified as the token generator class name, the following token information can be specified:
   a. Whether to use IdentityAssertion option. This option is selected if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For example, WebSphere Application Server sends only the user name of the original caller for a Username token generator.
   b. Whether to use RunAsSubject identity option. This option is used if an identity assertion is defined and you want to use the Run As identity instead of the initial caller identity for identity assertion in a downstream call. This option is valid only if you have configured the Username token as the token generator.
   c. Whether to use sendRealm.
   d. Whether to specify the nonce.

      This option indicates whether a Nonce is included for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of Username tokens. Nonce is valid only when the generated token type is a Username token, and it is available only for the request generator binding.
   e. Specifies the keyword of the time stamp. This option indicates whether to verify a time stamp in the Username token. The time stamp is valid only when the incorporated token type is a Username token.
   f. Specifies a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

URI value type: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
Specifies a Username token.

9. If the Kerberos v5 token is specified as the token generator class name, the following token information can be specified:

Token Information	Description	Default Value
name	Kerberos client principal name	
password	Kerberos client password	
realm	Kerberos realm associated with the Kerberos client	Default realm name in Kerberos configuration file. Specify null to use the default value.
targetService	Kerberos service name associated with the target web services.	
targetHost	Kerberos realm name associated with the Kerberos service name.	
tokenValueType	Kerberos token value type in QName defined by Oasis Kerberos Token Profile v1.1 specification.	http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ
targetRealm	Kerberos realm name associated with the Kerberos service name.	Default realm name in the Kerberos configuration file
prompt	A boolean value to enable the login prompt.	false
supportTokenRequireSHA1	A boolean value to require a SHA1 key that is used in subsequent request messages when the Kerberos token is used as a supporting token.	false SHA1 key is consumed only if the supporting Kerberos token is protected. If set to true, the SHA1 key is always consumed.
alwaysAPREQ	A boolean value to indicate that the client should always send the Kerberos AP_REQ token in the request messages.	false The SHA1 key is used instead in the subsequent messages. If set to true, the Kerberos AP_REQ token is always used.
requireDKT	A boolean value to require a derived key for message protection.	false
clabel	The client label for the derived key.	WS-SecureConversation Specify null to use the default value.
slabel	The service label for the derived key.	WS-SecureConversation Specify null to use the default value.
keylen	The length of the derived key.	16 Specify zero to use the default value
noncelen	The length of the nonce.	16 Specify zero to use the default value

Token Information	Description	Default Value
encComponent	An instance of WSSEncryption.	Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component.
sigComponent	An instance of WSSSignature.	Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component.

Additional token value types are defined in the OASIS Kerberos Token Profile v1.1 specification. Specify the token value type as the local name. It is not necessary to specify the value type URI for the Kerberos v5 token.

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

10. If Secure Conversation is used for message protection, the following information must be specified:

Information	Description
bootstrapWSSGenerationContext	The bootstrap configuration used to secure the RequestSecurityToken (RST) token.
bootstrapWSSConmingContext	The bootstrap configuration used for consuming a secured RequestSecurityTokenResponse (RSTR).
ENDPOINT_URL	The service end point URL.
EncryptionAlgorithm	This determines the key size.
cLabel	The client label used when creating the derived key.
sLabel	The server label used when creating the derived key.

11. Set the components into the wssGenerationContext object.
12. Invoke the wssGenerationContext.process() method.

Results

Using the Web Services Security API (WSS API) process, you can configured the token generator.

What to do next

Next, you must specify a similar token consumer configuration.

Configuring generator security tokens using the WSS API:

You can secure the SOAP messages, without using policy sets, by using the Web Services Security APIs. To configure the token on the generator side, use the Web Services Security APIs (WSS API). The generator security tokens are part of the `com.ibm.websphere.wssecurity.wssapi.token` interface package.

Before you begin

The pluggable token framework in WebSphere Application Server has been redesigned so that the same framework from the WSS API can be reused. The same implementation of creating and validating security token can be used both for the Web Services Security runtime and for the WSS API application code. The redesigned framework also simplifies the SPI programming model and will make it easier to add security token types.

You can use the WSS API or you can configure the tokens by using the administrative console. To configure tokens, you must complete the following token tasks:

- Configure the generator tokens.
- Configure the consumer tokens.

About this task

The JAAS CallbackHandler and JAAS LoginModule are responsible for creating the security token on the generator side.

On the generator side, the token is created by using the JAAS LoginModule and by using JAAS CallbackHandler to pass authentication data. Then, the JAAS LoginModule creates the securityToken object, such as the UsernameToken, and passes it to the Web Services Security run time.

On the consumer side, the XML format is passed to the JAAS LoginModule for validation or authentication. then the JAAS CallbackHandler is used to pass authentication data from the Web Services Security runtime to the LoginModule. After the token is authenticated, a security token object is created, and the token is passed it to the Web Services Security runtime.

When using the WSS API for generator token creation, certain default behaviors occur. The simplest way to use the WSS API is to use the default behavior (see the example code). The WSS API provide default values for the token type, the token value, and the JAAS confirmation name. The default token behaviors include:

Table 96. Token decisions and default behaviors. Several token characteristics are configured by default.

Generator token decisions	Default behavior
Which token type to use	The token type specifies which type of token to use for message integrity, message confidentiality, or message authenticity. WebSphere Application Server provides the following pre-configured generator token types for message integrity and message confidentiality: • Derived key token • X509 tokens You can also create custom token types, as needed. WebSphere Application Server also provides the following pre-configured generator token types for the message authenticity: • Username token • LTPA tokens • X509 tokens You can also create custom token types, as needed.
What JAAS login configuration name to specify	The JAAS login configuration name specifies which JAAS login configuration name to use.
Which configuration type to use	The JAAS login module specifies the configuration type. Only the pre-configured generator configuration types can be used for generator token types.

The SecurityToken class (com.ibm.websphere.wssecurity.wssapi.token.SecurityToken) is the generic token class and represents the security token that has methods to get the identity, the XML format, and the cryptographic keys. Using the SecurityToken class, you can apply both the signature and encryption to the SOAP message. However, to apply both, you must have two SecurityToken objects, one for the signature and one for encryption, respectively.

The following tokens types are subclasses of the generic security token class:

Table 97. Subclasses of the SecurityToken. Use the subclasses to represent the security token.

Token type	JAAS login configuration name
Username token	system.wss.generate.unt
Security context token	system.wss.generate.sct
Derived key token	system.wss.generate.dkt

The following tokens types are subclasses of the binary security token class:

Table 98. Subclasses of the BinarySecurityToken. Use the subclasses to represent the binary security token.

Token type	JAAS login configuration name
LTPA token	system.wss.generate.ltpa
LTPA propagation token	system.wss.generate.ltpaProp
X.509 token	system.wss.generate.x509
X.509 PKI Path token	system.wss.generate.pkiPath
X.509 PKCS7 token	system.wss.generate.pkcs7

Note:

- For each JAAS login token generator configuration name, there is a respective token consumer configuration name. For example, for the Username token, the respective token consumer configuration name is system.wss.consume.unt.
- The LTPA and LTPA propagation tokens are only available to a requester that is running as a server-based client. The LTPA and LTPA propagation tokens are not supported for the Java SE 6 or Java EE application client.

Procedure

1. To configure the securityToken package, `com.ibm.websphere.wssecurity.wssapi.token`, first ensure that the application server is installed.
2. Use the Web Services Security token generator process to configure the tokens. For each token type, the process is similar to the following process that demonstrates the UsernameToken token generator process:
   a. Use WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Create the WSSGenerationContext instance from the WSSFactory instance.
   c. Create a JAAS CallbackHandler. The authentication data, such as the user name and password are specified as part of the CallbackHandler. For example, the following code specifies Chris as the user name and sirhC as the password: UNTGenerationCallbackHandler("Chris", "sirhC");
   d. Call any JAAS CallbackHandler parameters and review the token class information for which parameters are required or optional. For example, for the UsernameToken, the following parameters can be configured also:

      Nonce

      Indicates whether a nonce is included in the user name token for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens. The nonce value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.

Created timestamp
Indicates whether to insert a time stamp into the UsernameToken. The timestamp value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.

e. Create the SecurityToken from WSSFactory.

   By default, the UsernameToken API specifies the ValueType as: "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken"

   By default, the UsernameToken API provides the QName of this class and specifies the NamespaceURI as http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd and also specifies the LocalPart as UsernameToken.

f. Optional: Specify the JAAS login module configuration name. On the generator side, the configuration type is always `generate` (for example, `system.wss.generate.unt`).

g. Add the SecurityToken to the WSSGenerationContext.

h. Call WSSGenerationContext.process() and generate the WS-Security header.

Results

If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext process() is called, and the security token for the generator binding is attached.

Example

The following example code shows how to use WSS APIs to create a Username security token, attach the Username token to the SOAP message, and configure the Username token in the generator binding.

```
// import the packages
import javax.xml.ws.BindingProvider;
import com.ibm.websphere.wssecurity.wssapi.*;
import com.ibm.websphere.wssecurity.callbackhandler.*;
...
   // obtain the binding provider
   BindingProvider bp = ... ;

   // get the request context
   Map<String, Object> reqContext = bp.getRequestContext();

   // generate WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

   // generate WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

   // generate callback handler
   UNTGenerateCallbackHandler untCallbackHandler =
   new UNTGenerateCallbackHandler("Chris", "sirhC");

   // generate the username token
   SecurityToken unt = factory.newSecurityToken(UsernameToken.class, untCallbackHandler);

   // add the SecurityToken to the WSSGenerationContext
   gencont.add(unt);

   // generate the WS-Security header
   gencont.process(reqContext);
```

The following example code shows how to modify the preceding Username token sample to create an LTPAv2 token from the runAs identity on the current thread. The two lines of code that instantiate the callback handler and create the security token are replaced with the following two lines of code:

```
// generate callback handler
LTPAGenerateCallbackHandler ltpaCallbackHandler = new LTPAGenerateCallbackHandler(null, null);

// generate the LTPAv2 token
SecurityToken ltpa = wssfactory.newSecurityToken(LTPAv2Token.class, ltpaCallbackHandler);
```

The instantiation of the LTPAGenerateCallbackHandler object with (null, null) indicates that the LTPA token should be generated from the current runAs identity. If the callback handler is instantiated with basicAuth information, ("userName", "password"), a new LTPA token is created using the specified basicAuth information.

The following example shows how to use secure conversation with the WSS APIs to configure the generator tokens, as well as the consumer tokens. In this example, the SecurityContextToken token is created using the WS-SecureConversation draft namespace: `http://schemas.xmlsoap.org/ws/2005/02/sc/sct`. To use the WS-SecureConversation version 1.3 namespace, `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct`, specify SecurityContextToken13.class instead of SecurityContextToken.class.

```
// import the packages
import javax.xml.ws.BindingProvider;
import com.ibm.websphere.wssecurity.wssapi.*;
import com.ibm.websphere.wssecurity.callbackhandler.*;
...
   // obtain the binding provider
   BindingProvider bp = ... ;

   // get the request context
   Map<String, Object> reqContext = bp.getRequestContext();

   // generate WSSFactory instance
   WSSFactory wssFactory = WSSFactory.getInstance();

   WSSGenerationContext bootstrapGenCon = wssFactory.newWSSGenerationContext();

   // Create a Timestamp
   ...
   // Add Timestamp
   ...

   // Sign the SOAP Body, WS-Addressing headers, and Timestamp
   X509GenerateCallbackHandler btspReqSigCbHandler = new X509GenerateCallbackHandler(...);
   SecurityToken btspReqSigToken = wssFactory.newSecurityToken(X509Token.class,
                                                               btspReqSigCbHandler);
   WSSSignature bootstrapReqSig = wssFactory.newWSSSignature(btspReqSigToken);
   bootstrapReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);

   // Add Sign Parts
   ...
   bootstrapGenCon.add(bootstrapReqSig);

   // Encrypt the SOAP Body and the Signature
   X509GenerateCallbackHandler btspReqEncCbHandler = new X509GenerateCallbackHandler(...);
   SecurityToken btspReqEncToken = wssFactory.newSecurityToken(X509Token.class,
                                                               btspReqEncCbHandler);
   WSSEncryption bootstrapReqEnc = wssFactory.newWSSEncryption(btspReqEncToken);
   bootstrapReqEnc.setEncryptionMethod(WSSEncryption.AES128);
   bootstrapReqEnc.setKeyEncryptionMethod(WSSEncryption.KW_RSA15);

   // Add Encryption parts
   ...
   bootstrapGenCon.add(bootstrapReqEnc);
   WSSConsumingContext bootstrapConCon = wssFactory.newWSSConsumingContext();
   X509ConsumeCallbackHandler btspRspVfyCbHandler = new X509ConsumeCallbackHandler(....);
   WSSVerification bootstrapRspVfy = wssFactory.newWSSVerification(X509Token.class,
                                                                   btspRspVfyCbHandler);
   bootstrapRspVfy.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

   // Add Verify parts
   ...
   bootstrapConCon.add(bootstrapRspVfy);
   X509ConsumeCallbackHandler btspRspDecCbHandler = new X509ConsumeCallbackHandler(...);
   WSSDecryption bootstrapRspDec = wssFactory.newWSSDecryption(X509Token.class,
                                                               btspRspDecCbHandler);
   bootstrapRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
   bootstrapRspDec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_RSA15);

   // Add Decryption parts
   ...
   bootstrapConCon.add(bootstrapRspDec);
   SCTGenerateCallbackHandler sctgch = new SCTGenerateCallbackHandler(bootstrapGenCon,
                                                                      bootstrapConCon,
                                                                      ENDPOINT_URL,
                                                                      WSSEncryption.AES128);
   SecurityToken[] scts = wssFactory.newSecurityTokens(new Class[]{SecurityContextToken.class},
                                                       sctgch);
   SecurityContextToken sct = (SecurityContextToken)scts[0];

   // Use the SCT to generate DKTs for Secure Conversation
   // Signature algorithm and client and service labels
   DerivedKeyToken dktSig = sct.getDerivedKeyToken(WSSSignature.HMAC_SHA1,
                                                   "WS-SecureConversation",
                                                   "WS-SecureConversation");

   // Encryption algorithm and client and service labels
   DerivedKeyToken dktEnc = sct.getDerivedKeyToken(WSSEncryption.AES128,
                                                   "WS-SecureConversation",
                                                   "WS-SecureConversation");
```

```
// Create the application generation context for the request message
WSSGenerationContext applicationGenCon = wssFactory.newWSSGenerationContext();

// Create and add Timestamp
...

// Add the derived key token and Sign the SOAP Body and WS-Addressing headers
WSSSignature appReqSig = wssFactory.newWSSSignature(dktSig);
appReqSig.setSignatureMethod(WSSSignature.HMAC_SHA1);
appReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
...
applicationGenCon.add(appReqSig);

// Add the derived key token and Encrypt the SOAP Body and the Signature
WSSEncryption appReqEnc = wssFactory.newWSSEncryption(dktEnc);
appReqEnc.setEncryptionMethod(WSSEncryption.AES128);
appReqEnc.setTokenReference(SecurityToken.REF_STR);
appReqEnc.encryptKey(false);
...
applicationGenCon.add(appReqEnc);

// Create the application consuming context for the response message
WSSConsumingContext applicationConCon = wssFactory.newWSSConsumingContext();

//client and service labels and decryption algorithm
SCTConsumeCallbackHandler sctCbHandler = new SCTConsumeCallbackHandler("WS-SecureConversation",
                                                                       "WS-SecureConversation",
                                                                       WSSDecryption.AES128);

// Derive the token from SCT and use it to Decrypt the SOAP Body and the Signature
WSSDecryption appRspDec = wssFactory.newWSSDecryption(SecurityContextToken.class,
                                                      sctCbHandler);
appRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
appRspDec.encryptKey(false);
...
applicationConCon.add(appRspDec);

// Derive the token from SCT and use it to Verify the
// signature on the SOAP Body, WS-Addressing headers, and Timestamp
WSSVerification appRspVfy = wssFactory.newWSSVerification(SecurityContextToken.class,
                                                          sctCbHandler);
...
applicationConCon.add(appRspVfy);
...
applicationGenCon.process(reqContext);
applicationConCon.process(reqContext);
```

What to do next

For each token type, configure the token using the WSS APIs or using the administrative console. Next, specify the similar consumer tokens if you have not done so.

If both the generator and consumer tokens are configured, continue securing SOAP messages either by signing the SOAP message or by encrypting the message, as needed. You can use either the WSS APIs or the administrative console to secure the SOAP messages.

Sending self-issued SAML bearer tokens using WSS APIs:

You can create self-issued SAML tokens with the bearer subject confirmation method and then send these tokens with Web services request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API).

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings.

About this task

You can build your web services client to use SAML tokens with the bearer subject confirmation method in SOAP request messages using the Web Services Security programming interfaces. Using the programming interfaces in a web services client to specify the use of SAML tokens with bearer subject confirmation is an alternative approach to using policy sets and binding configurations.

You can create a self-issued SAML token and then send the SAML token in web services request messages from a web services client. The web services application client used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code snippets from the sample are described in the procedure section, and a complete, ready-to-use web services client sample is provided in the Example section.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to add the Web Services Security API to pass SAML tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

   b. Obtain the JaxWSServicesSamples client code.

      For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

      The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

   c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

   Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS Web services thin client; however, you can also use a managed client.

2. Attach the SAML20 Bearer WSHTTPS default policy set to the web services provider. This policy set is used to protect messages using HTTPS transport. Read about configuring client and provider bindings for the SAML Bearer token for details on how to attach the SAML20 Bearer WSHTTPS default policy set to the Web services provider. The example in this procedure uses self-issued SAML tokens. When you configure the provider bindings, the truststore configuration and certificate must match the signing key of the self-issued token.

3. Assign the SAML Bearer Provider sample default general bindings to the sample web services provider. Read about configuring client and provider bindings for the SAML bearer token for details on assigning the SAML Bearer Provider sample default general bindings to your web services application.

4. Create the self-issued SAML token. The following code snippet illustrates creating the SAML token:

```
// Create the SAML token.
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Use the CallService() method to specify the Web services security configuration parameters that are required to invoke a target Web services provider using a self-issued SAML token.

      The CallService() method sets the configuration parameters that are required by the Web Services Security runtime environment via the `com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext` custom property to generate a self-issued SAMLToken.

Read about configuring a SAML token during token creation for more information about how you can specify configuration properties to control how the token is configured.

b. Add the Thin Client for JAX-WS JAR file to the class path. Add the *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.

c. Use the WSSFactory newSecurityToken method to specify how to create the SAML token.

Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

Creating a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the PolicyTool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken
```

The `SAMLToken.class` parameter specifies the type of security token to create.

The `callbackHandler` object contains parameters that define the characteristics of the SAMLToken that you are creating. This object points to a SAMLGenerateCallbackHandler object that specifies the configuration parameters described in the following table:

Table 99. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the bearer subject confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `Bearer` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Uses the constant value, `WSSConstants.SAML.SAML20_VALUE_TYPE`, to specify a SAML 2.0 token type. When a web services client has policy set attachments, this property is not used by Web Services Security runtime environment. In this scenario, specify the token value type by the valueType attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 2.0 token; however, you can also use the `WSSConstants.SAML.SAML11_VALUE_TYPE` value.	Yes
SamlConstants.SAML_NAME_IDENTIFIER	Specifies a user identity such as `myname` as the NameID value in the SAML token. If you do not define this parameter when using the Thin Client for JAX-WS, the NameID value does not contain useful information. If you are using a web services managed client, such a Java Platform, Enterprise Edition (Java EE) application making a web services request invocation, the Web Services Security runtime environment tries to extract user security information from the security context. Similarly, if you do not define this parameter for a managed web services client, the NameID value contains an UNAUTHENTICATED name identifier. This property is not used if your web services client has policy set attachments. Read about sending SAML tokens to learn more about sending the SAML token identity and attributes.	No
SamlConstants.SIGNATURE_REQUIRED	Specifies whether the issuer is required to digitally sign the SAML token. A `true` value specifies that issuer is required to digitally sign the SAML token. This value is the default.	No

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");
```

d. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

5. Add the SAML token to the SOAP security header of a Web services request messages.
   a. Initialize the web services client and configure the SOAPAction properties. The following code snippet illustrates these actions:

```
// Initialize web services client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);

// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");
```

   b. Initialize the WSSGenerationContext. The following code illustrates the use of the WSSGenerationContext interface to initialize a generation context and enable you to insert the `SAMLToken` into the web services request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

   Specifically, the `gencont.add(samlToken)` method call specifies to put the SAML token into a request message. Use the PolicyTool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
 "permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

   c. Add the timestamp element in the SOAP messages security header. The SAML20 Bearer WSHTTPS default policy set requires web services requests and response messages to carry a timestamp element in SOAP messages Security header. In the following code snippet, the `factory.newWSSTimestamp()` method call generates the timestamp, and the `gencont.add(timestamp)` method call specifies the timestamp to put into a request message:

```
// Add a timestamp to the request message.
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencont.add(timestamp);

gencont.process(requestContext);
```

   d. Attach WSSGenerationContext object to the web services RequestContext object. The WSSGenerationContext object now contains all the security information that is required to format a request message. The `gencont.process(requestContext)` method call attaches the WSSGenerationContext object to the web services RequestContext object to enable the Web Services Security runtime environment to format the required SOAP security header; for example:

```
// Attaches WSSGenerationContext object to the web services RequestContext object.
gencont.process(requestContext);
```

   e. Specify SSL transport level message protection using JVM properties.

   The SAML20 Bearer WSHTTPS default policy set requires transport-level message protection using SSL. Specify SSL transport-level message protection using the following JVM property:

```
-Dcom.ibm.SSL.ConfigURL=file:profile_root\properties\ssl.client.props
```

   Alternatively, you can define the SSL configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

Results

You have created a self-issued SAML token with the bearer subject confirmation method and then sent this token with web services request messages using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a web services client application that demonstrates how to create a self-issued SAML token and send that SAML token in web services request messages. If your usage

scenario requires SAML tokens, but does not require your application to pass the SAML tokens using web services messages, you only need to use the first part of the following sample code, up through the `// Initialize web services client` section.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */

package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;

import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.util.Map;
import java.util.HashMap;

import javax.xml.ws.BindingProvider;

/**
 * SampleClient
 * main entry point for thin client JAR sample
 * and worker class to communicate with the services
 */
public class SampleClient {

  private String urlHost = "localhost";
  private String urlPort = "9443";
  private static final String CONTEXT_BASE = "/WSSampleSei/";
  private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
  private String message = "HELLO";
  private String uriString = "https://" + urlHost + ":" + urlPort;
  private String endpointURL = uriString + ECHO_CONTEXT12;
  private String input = message;

  /**
   * main()
   *
   * see printusage() for command-line arguments
   *
   * @param args
   */
  public static void main(String[] args) {
    SampleClient sample = new SampleClient();
    sample.CallService();
  }

  /**
   * CallService Parms were already read. Now call the service proxy classes
   *
   */
  void CallService() {
    String response = "ERROR!:";
    try {
      System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");
      System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");

      // Initialize WSSFactory object
      WSSFactory factory = WSSFactory.getInstance();
      // Initialize WSSGenerationContext
      WSSGenerationContext gencont = factory.newWSSGenerationContext();
      // Initialize SAML issuer configuration via custom properties
      HashMap<Object, Object> customProps = new HashMap<Object,Object>();

      customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
      customProps.put(SamlConstants.TTL_PROP, "3600000");
      customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
      customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
      customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
      customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
      customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
      gencont.add(customProps); //Add custom properties
```

```
      // Create SAMLToken
      HashMap<Object, Object> map = new HashMap<Object, Object>();
      map.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");
      map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
      map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
      map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
      SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);

      SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");

      System.out.println("SAMLToken id = " + samlToken.getId());

      // Initialize web services client
      EchoService12PortProxy echo = new EchoService12PortProxy();
      echo._getDescriptor().setEndpoint(endpointURL);

      // Configure SOAPAction properties
      BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
      Map<String, Object> requestContext = bp.getRequestContext();
      requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
      requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
      requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

      gencont.add(samlToken);

      // Add timestamp
      WSSTimestamp timestamp = factory.newWSSTimestamp();
      gencont.add(timestamp);

      gencont.process(requestContext);

      // Build the input object
      EchoStringInput echoParm =
        new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
      echoParm.setEchoInput(input);
      System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

      // Prepare to consume timestamp in response message.
      WSSConsumingContext concont = factory.newWSSConsumingContext();
      concont.add(WSSConsumingContext.TIMESTAMP);
      concont.process(requestContext);

      // Call the service
      response = echo.echoOperation(echoParm).getEchoResponse();

      System.out.println(">> CLIENT: SEI Echo invocation complete.");
      System.out.println(">> CLIENT: SEI Echo response is: " + response);
    } catch (Exception e) {
      System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
      e.printStackTrace();
    }
  }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _191EBC44865015D9AB1270745072344
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to https://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Inserting SAML attributes using WSS APIs:

You can insert custom attributes into self-issued SAML tokens by using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs).

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about propagating self-issued SAML bearer tokens by using WSS APIs.
- Read about propagating self-issued SAML sender-vouches tokens by using WSS APIs with message level protection.

- Read about propagating self-issued SAML sender-vouches tokens by using WSS APIs with SSL transport protection.
- Read about propagating self-issued SAML holder-of-key tokens with symmetric key by using WSS APIs.
- Read about propagating self-issued SAML holder-of-key tokens with asymmetric key by using WSS APIs.

About this task

This task shows example code that inserts custom attributes into self-issued SAML security tokens. This particular example uses the bearer subject confirmation method. You can add attributes to any SAML security tokens, and the same code can be used with other subject confirmation methods.

Procedure

Insert custom attributes when creating SAML security tokens; for example:

```
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;
import com.ibm.wsspi.wssecurity.saml.data.SAMLAttribute;

WSSFactory factory = WSSFactory.getInstance();
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");
map.put(SamlConstants.Token_REQUEST, "issue");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
String groups[] = {"IBMer", "Texan"};
SAMLAttribute sattribute = new SAMLAttribute("Membership", groups, null,null, null, null);
al.add(sattribute);
String gender[] = {"Female"};
sattribute = new SAMLAttribute("Gender", gender, null,null, null, null);
al.add(sattribute);
map.put(SamlConstants.SAML_ATTRIBUTES, al);
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler,
                                                   "system.wss.generate.saml");
```

Results

You have inserted custom attributes to a SAML security token.

Example

The following example shows the custom attributes in the SAML Assertion:

```
<saml2:Assertion xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
                  Version="2.0"
                       ID="_E62A1CA3C2F21D9A9B1287772824570"
                       IssueInstant="2010-10-22T18:40:24.531Z">
    <saml2:Issuer>example.com</samls2:Issuer>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    ...
    </ds:Signature>
    <saml2:Subject>
        <saml2:NameID>Alice</saml2:NameID>
        <saml2:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer"></saml2:SubjectConfirmation>
    </saml2:Subject>
    <saml2:Conditions NotBefore="2010-10-22T18:40:24.531Z"
                    NotOnOrAfter="2010-10-22T19:40:24.531Z">
    </saml2:Conditions>
    <saml2:AttributeStatement>
        <saml2:Attribute Name="Membership">
            <saml2:AttributeValue>IBMer</saml2:AttributeValue>
            <saml2:AttributeValue>Texan</saml2:AttributeValue>
        </saml2:Attribute>
```

```
        <saml2:Attribute Name="Gender">
            <saml2:AttributeValue>Female</saml2:AttributeValue>
        </saml2:Attribute>
    </saml2:AttributeStatement>
</saml2:Assertion>
```

What to do next

Merge the code with the example code listed in the “Propagating self-issued SAML bearer tokens by using WSS APIs” topic to generate SAML security tokens. You can see SAML attributes in the SAML Assertions.

Sending self-issued SAML sender-vouches tokens using WSS APIs with message level protection:

You can create self-issued SAML tokens with the sender-vouches subject confirmation method and then use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to send these tokens with web services request messages with message level protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings.

About this task

You can protect SOAP request messages and SAML tokens by using the Web Services Security programming interface to satisfy the sender-vouches subject confirmation method validation requirements with message level protection. Using the programming interfaces in web services client is an alternative approach to using policy set and binding configuration.

You can create a self-issued SAML token and then send the SAML token in web services request messages from a web services client. The web services application client used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code snippets from the sample are described in the procedure section, and a complete, ready-to-use web services client sample is provided in the Example section.

This product does not provide a default policy set that requires SAML tokens with sender-vouches subject confirmation method. Read about configuring client and provider bindings for the SAML sender-vouches token to learn more about how to create a Web Services Security policy to require SAML tokens with sender-vouches subject confirmation and how to create a custom binding configuration. You must attach the policy and binding to the web services provider. The code sample described in this task assumes that the web services provider policy requires that both the SAML tokens and the message bodies are digitally signed by using an X.509 security token.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to add the Web Services Security API to pass SAML sender-vouches tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.
   b. Obtain the JaxWSServicesSamples client code.

For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS Web services thin client; however, you can also use a managed client.

2. Use the CallService() method to specify the Web services security configuration parameters that are required to invoke a target Web services provider using a self-issued SAML token.

   The CallService() method sets configuration parameters that are required by the Web Services Security runtime environment via the `com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext` custom property to generate a self-issued SAMLToken.

   The following code snippet illustrates using the CallService() method to set the `SamlConstants.SAML_SELF_ISSUER_CONFIG` system property:

```
public static void main(String[] args) {
  SampleSamlSVClient sample = new SampleSamlSVClient();
  sample.CallService();
}

/**
 * CallService Parms were already read. Now call the service proxy classes
 *
 */
void CallService() {
  String response = "ERROR!:";
  try {
    System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas.conf");
```

   Read about configuring a SAML token during token creation for more information about how you can specify configuration properties to control how the token is configured.

3. Add the Thin Client for JAX-WS JAR file to the class path. Add *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.

4. Create the self-issued SAML token. The following code snippet illustrates creating the SAML token:

```
// Create SAMLToken
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");
System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Use the WSSFactory newSecurityToken method to specify how to create the SAML token.

      Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

      Creating a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken
```

      The `SAMLToken.class` parameter specifies the type of security token to create.

      The `callbackHandler` object contains parameters that define the characteristics of the SAMLToken that you are creating. This object points to a SAMLGenerateCallbackHandler object that specifies the following configuration parameters described in the following table:

Table 100. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the sender-vouches confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `sender-vouches` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Uses the constant value, `WSSConstants.SAML.SAML20_VALUE_TYPE` to specify a SAML 2.0 token type. When a web services client has policy set attachments, this property is not used by Web Services Security runtime environment. In this scenario, specify the token value type by the valueType attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 2.0 token; however, you can also use the `WSSConstants.SAML.SAML11_VALUE_TYPE` value.	Yes
SamlConstants.SAML_NAME_IDENTIFIER	Specifies a user identity such as `myname` as the NameID value in the SAMLToken. If you do not define this parameter when using the Thin Client for JAX-WS, the NameID value does not contain useful information. If you are using a web services managed client, such a Java Platform, Enterprise Edition (Java EE) application making a web services request invocation, the Web Services Security runtime environment tries to extract user security information from the security context. Similarly, if you do not define this parameter for a managed web services client, the NameID value contains an UNAUTHENTICATED name identifier. This property is not used if your web services client has policy set attachments. Read about sending SAML tokens to learn more about sending the SAML token identity and attributes.	No
SamlConstants.SIGNATURE_REQUIRED	Specifies whether the issuer is required to digitally sign the SAML token. A `true` value specifies that issuer is required to digitally sign the SAML token. This value is the default.	No

The `system.wss.generate.saml` parameter specifies to use a Java Authentication and Authorization Service (JAAS) login configuration and specifies the login module that is invoked to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
Djava.security.auth.login.config=profile_root/properties/wsjaas.conf
```

Alternatively, you can specify a JAAS login configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas.conf");
```

b. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

5. Add the SAML token to the SOAP security header of web services request messages.

a. Initialize the web services client and configure the SOAPAction properties. The following code example illustrates these actions:

```
// Initialize web services client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);

// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

b. Initialize the WSSGenerationContext. The following code snippet illustrates the use of the gencont.object of the WSSGenerationContext type to initialize a generation context to enable you to insert the `SAMLToken` into a web services request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

Specifically, the `gencont.add(samlToken)` method call specifies to put the SAML token into a request message. This operation requires the client code to have the following Java 2 Security permission:

```
"permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

6. Add an X.509 token for message protection.

This sample code uses the dsig-sender.ks key file and the SOAPRequester sample key. You must not use the sample key in a production environment. The following code snippet illustrates adding an X.509 token for message protection:

```
// Add an X.509 Token for message protection
X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
        null,
        "profile_root/etc/ws-security/samples/dsig-sender.ks",
        "JKS",
        "client".toCharArray(),
        "soaprequester",
        "client".toCharArray(),
        "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

SecurityToken x509 = factory.newSecurityToken(X509Token.class,
        x509callbackHandler, "system.wss.generate.x509");

WSSSignature sig = factory.newWSSSignature(x509);
sig.setSignatureMethod(WSSSignature.RSA_SHA1);

WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(samlToken);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
sig.addSignPart(sigPart);
sig.addSignPart(WSSSignature.BODY);
```

a. Create a WSSSignature object with the X509 token. The following line of code creates a WSSSignature object with the X509 token:

```
WSSSignature sig = factory.newWSSSignature(x509);
```

b. Add the signed part to use for message protection. The following line of code specifies to add WSSSignature.BODY as the signed part:

```
sig.addSignPart(WSSSignature.BODY);
```

c. Add the timestamp element in the SOAP messages security header. The SAML20 SenderVouches WSHTTPS and SAML11 SenderVouches WSHTTPS policy sets require web services requests and response messages to carry a timestamp element in the SOAP messages Security header. In the following code snippet, the `factory.newWSSTimestamp()` method call generates the timestamp, and the `gencont.add(timestamp)` method call adds the timestamp into the request message:

```
// Add Timestamp
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencont.add(timestamp);
sig.addSignPart(WSSSignature.TIMESTAMP);

gencont.add(sig);

WSSConsumingContext concont = factory.newWSSConsumingContext();
```

d. Configure the verification of the digital signature in the response message.

A separate WSSSignPart is needed to specify the SecurityTokenReference transformation algorithm that is represented by the `WSSSignPart.TRANSFORM_STRT10` attribute. A SAML Token cannot be digitally signed directly. This attribute enables the Web Services Security runtime environment to generate a SecurityTokenReference element to reference the SAMLToken and to digitally sign the SAMLToken using the SecurityTokenReference transformation. The following line of code specifies to use the `WSSSignPart.TRANSFORM_STRT10` attribute:

```
WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(samlToken);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
```

e. Attach the WSSGenerationContext object to the web services RequestContext object. The WSSGenerationContext object now contains all the security information that is required to format a

request message. The `gencont.process(requestContext)` method call attaches the WSSGenerationContext object to the web services RequestContext object to enable the Web Services Security runtime environment to format the required SOAP security header; for example:

```
// Attaches the WSSGenerationContext object to the web services RequestContext object.
gencont.process(requestContext);
```

7. Use the X.509 token to validate the digital signature and the integrity of the response message. If the provider policy requires the response message to be digitally signed, you must initialize the X.509 token.
   a. A X509ConsumeCallbackHandler object is initialized with a truststore, dsig-receiver.ks, and a certificate path object to validate the provider digital signature. The following line of code is used to initialize the X509ConsumeCallbackHandler object:

```
X509ConsumeCallbackHandler callbackHandlerVer = new X509ConsumeCallbackHandler(
        "profile_root/etc/ws-security/samples/dsig-receiver.ks",
        "JKS",
        "server".toCharArray(),
        certList,
        java.security.Security.getProvider("IBMCertPath"));
```

   b. A WSSVerification object is created and the message body is added to the verification object so that the Web Services Security runtime environment validates the digital signature.

      The following line of code is used to initialize the WSSVerification object:

```
WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);
```

The WSSConsumingContext object now contains all the security information that is required to format a request message. The `concont.process(requestContext)` method call attaches the WSSConsumingContext object to the response method; for example:

```
// Attaches the WSSConsumingContext object to the web services RequestContext object.
concont.process(requestContext);
```

Results

You have created a self-issued SAML token with the sender-vouches confirmation method and then sent this token with web services request messages using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a complete, ready-to-use web services client application that demonstrates how to create a self-issued SAML sender-vouches token and send that SAML token in web services request messages. This sample code illustrates the procedure steps described previously.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */
package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.WSSException;
import com.ibm.websphere.wssecurity.wssapi.signature.WSSSignPart;
import com.ibm.websphere.wssecurity.wssapi.signature.WSSSignature;
import com.ibm.websphere.wssecurity.wssapi.verification.WSSVerification;
import com.ibm.websphere.wssecurity.wssapi.token.X509Token;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.security.InvalidAlgorithmParameterException;
```

```
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.cert.CertStore;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.CollectionCertStoreParameters;
import java.security.cert.X509Certificate;
import java.util.HashSet;
import java.util.Set;
import java.util.HashMap;
import java.util.Map;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
  private String urlHost = "localhost";
  private String urlPort = "9081";
  private static final String CONTEXT_BASE = "/WSSampleSei/";
  private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
  private String message = "HELLO";
  private String uriString = "http://" + urlHost + ":" + urlPort;
  private String endpointURL = uriString + ECHO_CONTEXT12;
  private String input = message;

  /**
   * main()
   *
   * see printusage() for command-line arguments
   *
   * @param args
   */
  public static void main(String[] args) {
    SampleSamlSVClient sample = new SampleSamlSVClient();
    sample.CallService();
  }

  /**
   * CallService Parms were already read. Now call the service proxy classes.
   *
   */
  void CallService() {
    String response = "ERROR!:";
    try {
      System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas.conf");

      // Initialize WSSFactory object
      WSSFactory factory = WSSFactory.getInstance();
      // Initialize WSSGenerationContext
      WSSGenerationContext gencont = factory.newWSSGenerationContext();
      // Initialize SAML issuer configuration via custom properties
      HashMap<Object, Object> customProps = new HashMap<Object,Object>();

      customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
      customProps.put(SamlConstants.TTL_PROP, "3600000");
      customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
      customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
      customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
      customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
      customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
      gencont.add(customProps); //Add custom properties

      // Create SAMLToken
      HashMap<Object, Object> map = new HashMap<Object, Object>();
      map.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");
      map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
      map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
      map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
      SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);

      SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");

      System.out.println("SAMLToken id = " + samlToken.getId());

      // Initialize web services client.
      EchoService12PortProxy echo = new EchoService12PortProxy();
      echo._getDescriptor().setEndpoint(endpointURL);

      // Configure SOAPAction properties
      BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
      Map<String, Object> requestContext = bp.getRequestContext();
      requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
      requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
      requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

      // Initialize WSSGenerationContext
      WSSGenerationContext gencont = factory.newWSSGenerationContext();
```

```
gencont.add(samlToken);

// Add X.509 Tokens for message protection
X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
    null,
    "profile_root/etc/ws-security/samples/dsig-sender.ks",
    "JKS",
    "client".toCharArray(),
    "soaprequester",
    "client".toCharArray(),
    "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

SecurityToken x509 = factory.newSecurityToken(X509Token.class,
    x509callbackHandler, "system.wss.generate.x509");

WSSSignature sig = factory.newWSSSignature(x509);
sig.setSignatureMethod(WSSSignature.RSA_SHA1);

WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(samlToken);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
sig.addSignPart(sigPart);
sig.addSignPart(WSSSignature.BODY);

// Add timestamp
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencont.add(timestamp);
sig.addSignPart(WSSSignature.TIMESTAMP);

gencont.add(sig);

WSSConsumingContext concont = factory.newWSSConsumingContext();

// Prepare to consume timestamp in response message
concont.add(WSSConsumingContext.TIMESTAMP);

// Prepare to verify digital signature in response message
X509Certificate x509cert = null;
try {
  InputStream is = new FileInputStream("profile_root/etc/ws-security/samples/intca2.cer");
  CertificateFactory cf = CertificateFactory.getInstance("X.509");
  x509cert = (X509Certificate) cf.generateCertificate(is);
} catch (FileNotFoundException e1) {
  throw new WSSException(e1);
} catch (CertificateException e2) {
  throw new WSSException(e2);
}
Set<Object> eeCerts = new HashSet<Object>();
eeCerts.add(x509cert);

java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
CollectionCertStoreParameters certparam = new CollectionCertStoreParameters(eeCerts);

CertStore cert = null;
try {
  cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
} catch (NoSuchProviderException e1) {
  throw new WSSException(e1);
} catch (InvalidAlgorithmParameterException e2) {
  throw new WSSException(e2);
} catch (NoSuchAlgorithmException e3) {
  throw new WSSException(e3);
}
if (certList != null) {
  certList.add(cert);
}

X509ConsumeCallbackHandler callbackHandlerVer = new X509ConsumeCallbackHandler(
    "profile_root/etc/ws-security/samples/dsig-receiver.ks",
    "JKS",
    "server".toCharArray(),
    certList,
    java.security.Security.getProvider("IBMCertPath"));

WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);

ver.addRequiredVerifyPart(WSSVerification.BODY);
concont.add(ver);

gencont.process(requestContext);
concont.process(requestContext);

// Build the input object
EchoStringInput echoParm =
    new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
echoParm.setEchoInput(input);
System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

// Call the service
response = echo.echoOperation(echoParm).getEchoResponse();
```

```
      System.out.println(">> CLIENT: SEI Echo invocation complete.");
      System.out.println(">> CLIENT: SEI Echo response is: " + response);
    } catch (Exception e) {
      System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
      e.printStackTrace();
    }
  }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _6CDDF0DBF91C044D211271166233407
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to http://localhost:9080/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Sending self-issued SAML sender-vouches tokens using WSS APIs with SSL transport protection:

You can create self-issued SAML tokens with the sender-vouches subject confirmation method and then use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to send these tokens with web services request messages with transport protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, SSL transport protection, and the use of policy sets to configure and administer web services settings.

About this task

You can build your web services client to use SAML tokens with the sender-vouches subject confirmation method in SOAP request messages using the Web Services Security programming interfaces. Using the programming interfaces in a web services client to specify the use of SAML tokens with sender-vouches subject confirmation using message protection at the transport level is an alternative approach to using policy sets and binding configurations.

You can create a self-issued SAML token and then send the SAML token in web services request messages from a web services client. The web services client application used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code examples from the sample are described in the procedure section, and a complete, ready-to-use web services client sample is provided in the Example section.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to use the Web Services Security API to pass SAML sender-vouches tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

   b. Obtain the JaxWSServicesSamples client code.

      For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

   c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

   Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS Web services thin client; however, you can also use a managed client.

2. Create a copy of either the SAML20 Bearer WSHTTPS default policy set or the SAML11 Bearer WSHTTPS default policy set.

   Provide a name for the copy of the policy set; for example SAML20 SenderVouches WSHTTPS or SAML11 SenderVouches WSHTTPS to help you identify that this new policy set uses the sender-vouches confirmation method.

   No additional change is required to the new policy file because the subject confirmation method is specified in the binding configuration and not in the policy.

   The new policy file contains either SAMLToken20Bearer or the SAMLToken11Bearer as the policy identifiers. Change the identifier of the SAMLToken20Bearer policy to SAMLToken20SV or change the identifier of the SAMLToken11Bearer policy to SAMLToken11SV to specify a more descriptive name. Changing the identifier of the policy does not change the policy enforcement in any way; however, adding a descriptive identifier helps you to identify that these policy identifiers use the sender-vouches confirmation method.

   If you want to view the settings of these policies, use the administrative console to complete the following actions:

   a. Click **Services** > **Policy sets** > **Application policy sets** > ***policy_set_name***.
   b. Click the **WS-Security** policy in the policies table.
   c. Click the **Main policy** link or the **Bootstrap policy** link.
   d. Click **Request token policies** from the Policy Details section.

3. Attach the new SAML20 SenderVouches WSHTTPS or SAML11 SenderVouches WSHTTPS policy set to the web services provider application. Read about configuring client and provider bindings for the SAML sender-vouches token for details on attaching this policy set to your web services provider application.

4. Create a copy of the SAML Bearer Provider sample default general bindings.

   a. For the new copy of the default policy set, provide a name that includes sender-vouches, such as `SAML Sender-vouches provider binding`.
   b. Change the value of the confirmationMethod property to `sender-vouches` in the token consumer configuration for the intended SAML token version. Read about configuring client and provider bindings for the SAML sender-vouches token for details on modifying the sender-vouches bindings to satisfy the vouching requirement.

5. Assign the new provider binding to the JaxWSServicesSamples provider sample. Read about configuring client and provider bindings for the SAML sender-vouches for details on assigning the SAML sender-vouches provider sample, default general bindings to your web services provider application.

6. Enable the web services provider SSL configuration attribute, clientAuthentication, to require X.509 client certificate authentication.

   The clientAuthentication attribute determines whether SSL client authentication is required. To specify the clientAuthentication attribute, use the administrative console to complete the following actions:

   a. Click **Security** > **SSL certificates and key management** > **Manage endpoint security configurations** > **{Inbound | Outbound}** > ***SSL_configuration***.
   b. Click the **WC_defaulthost_secure** link.
   c. Under Related Items, click the **SSL_configurations** link.

d. Select the **NodeDefaultSSLSettings** resource.

e. Click **Quality of protection (QoP) settings** link.

f. Select **Required** from the menu to specify client authentication.

Read about creating a secure sockets layer configuration to learn more about configuring the clientAuthentication attribute.

7. In the web services client code, use the CallService() method to specify the properties file that contains configuration parameters required to generate a self-issued SAML token.

   The CallService() method specifies configuration parameters that are required by the Web Services Security runtime environment to generate a self-issued SAMLToken.

   The following code snippet illustrates using the CallService() method to specify Web services security configuration parameters:

```
public static void main(String[] args) {
  SampleSamlSVClient sample = new SampleSamlSVClient();
  sample.CallService();
 }

/**
 * CallService Parms were already read. Now call the service proxy classes
 *
 */
void CallService() {
  String response = "ERROR!:";
  try {
    System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");
    // Initialize WSSFactory object
    WSSFactory factory = WSSFactory.getInstance();

    // Initialize WSSGenerationContext
    WSSGenerationContext gencont = factory.newWSSGenerationContext();
    // Initialize SAML issuer configuration via custom properties
    HashMap <Object, Object> customProps = new HashMap<Object,Object>();

    customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
    customProps.put(SamlConstants.TTL_PROP, "3600000");
    customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
    customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
    customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
    customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
    customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
    customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
    customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
    customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
    customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
    gencont.add(customProps); //Add custom properties
```

   Read about configuring a SAML token during token creation for more information about how you can specify configuration properties to control how the token is configured.

8. Add the Thin Client for JAX-WS JAR file to the classpath. Add *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the classpath. See the testing web services-enabled clients information for more information about adding this JAR file to the classpath.

9. Create the self-issued SAML token. The following code snippet illustrates creating the SAML sender-vouches token:

```
// Create SAMLToken
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");
System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Use the WSSFactory newSecurityToken method to specify how to create the SAML token.

      Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

      Creating a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the PolicyTool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken
```

      The `SAMLToken.class` parameter specifies the type of security token to create.

The `callbackHandler` object contains parameters that define the characteristics of the SAMLToken that you are creating. This object points to a SAMLGenerateCallbackHandler object that specifies the configuration parameters described in the following table:

Table 101. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the sender-vouches confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `sender-vouches` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Uses the constant value, `WSSConstants.SAML.SAML20_VALUE_TYPE` to specify a SAML 2.0 token type. When a web services client has policy set attachments, this property is not used by Web Services Security runtime environment. In this scenario, specify the token value type by the valueType attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 2.0 token; however, you can also use the `WSSConstants.SAML.SAML11_VALUE_TYPE` value.	Yes
SamlConstants.SAML_NAME_IDENTIFIER	Specifies a user identity such as `myname` as the NameID value in the SAMLToken. If you do not define this parameter when using the Thin Client for JAX-WS, the NameID value does not contain useful information. If you are using a web services managed client, such a Java Platform, Enterprise Edition (Java EE) application making a web services request invocation, the Web Services Security runtime environment tries to extract user security information from the security context. Similarly, if you do not define this parameter for a managed web services client, the NameID value contains an UNAUTHENTICATED name identifier. This property is not used if your web services client has policy set attachments. Read about sending SAML tokens to learn more about sending the SAML token identity and attributes.	No
SamlConstants.SIGNATURE_REQUIRED	Specifies whether the issuer is required to digitally sign the SAML token. A `true` value specifies that issuer is required to digitally sign the SAML token. This value is the default.	No

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");
```

b. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

Results

You have created a self-issued SAML token with the sender-vouches confirmation method with transport protection and then sent this token with web services request messages using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a complete, ready-to-use web services client application that demonstrates how to create a self-issued SAML sender-vouches token and send that SAML token in web services request messages. This sample code illustrates the procedure steps described previously.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */
package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;

import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.util.Map;
import java.util.HashMap;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
  private String urlHost = "localhost";
  private String urlPort = "9081";
  private static final String CONTEXT_BASE = "/WSSampleSei/";
  private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
  private String message = "HELLO";
  private String uriString = "http://" + urlHost + ":" + urlPort;
  private String endpointURL = uriString + ECHO_CONTEXT12;
  private String input = message;

  /**
   * main()
   *
   * see printusage() for command-line arguments
   *
   * @param args
   */
  public static void main(String[] args) {
    SampleSamlSVClient sample = new SampleSamlSVClient();
    sample.CallService();
  }

  /**
   * CallService Parms were already read. Now call the service proxy classes.
   *
   */
  void CallService() {
    String response = "ERROR!:";
    try {
      System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");

      // Initialize WSSFactory object
      WSSFactory factory = WSSFactory.getInstance();
      // Initialize WSSGenerationContext
      WSSGenerationContext gencont = factory.newWSSGenerationContext();
      // Initialize SAML issuer configuration via custom properties
     HashMap<Object, Object> customProps = new HashMap<Object,Object>();

      customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
      customProps.put(SamlConstants.TTL_PROP, "3600000");
      customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
      customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
      customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
      customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
      customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
      gencont.add(customProps); //Add custom properties

      // Create SAMLToken
      HashMap<Object, Object> map = new HashMap<Object, Object>();
      map.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");
      map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
      map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
      map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
      SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
      SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");

      System.out.println("SAMLToken id = " + samlToken.getId());

      // Initialize web services client
```

```
        EchoService12PortProxy echo = new EchoService12PortProxy();
        echo._getDescriptor().setEndpoint(endpointURL);

        // Configure SOAPAction properties
        BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
        Map<String, Object> requestContext = bp.getRequestContext();
        requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
        requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
        requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

        gencont.add(samlToken);

        // Add timestamp
        WSSTimestamp timestamp = factory.newWSSTimestamp();
        gencont.add(timestamp);

        gencont.process(requestContext);

        // Build the input object
        EchoStringInput echoParm =
            new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
        echoParm.setEchoInput(input);
        System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

        // Prepare to consume timestamp in response message
        WSSConsumingContext concont = factory.newWSSConsumingContext();
        concont.add(WSSConsumingContext.TIMESTAMP);
        concont.process(requestContext);

        // Call the service
        response = echo.echoOperation(echoParm).getEchoResponse();

        System.out.println(">> CLIENT: SEI Echo invocation complete.");
        System.out.println(">> CLIENT: SEI Echo response is: " + response);
      } catch (Exception e) {
        System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
        e.printStackTrace();
      }
    }
  }
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _6CDDF0DBF91C044D211271166233407
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to http://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Sending self-issued SAML holder-of-key tokens with symmetric key using WSS APIs:

You can create self-issued SAML tokens with the holder-of-key subject confirmation method and then use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to send these tokens with web services request messages.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about sending self-issued SAML bearer tokens by using WSS APIs.
- Read about sending self-issued SAML sender-vouches tokens by using WSS APIs with message level protection.

About this task

This task focuses on using the symmetric key that is embedded in SAML security tokens to generate a digital signature of selected SOAP message elements in order to satisfy holder-of-key subject confirmation method security requirements. The Web Services Security policy attached to the web services provider is that of the *SAML20 HoK Symmetric WSSecurity default* policy set that is shipped in WebSphere Application Server 7.0.0.7 and later releases.

Procedure

1. Create a SAML security token that contains holder-of-key subject confirmation method; for example:

```
WSSFactory factory = WSSFactory.getInstance();
// Initialize WSSGenerationContext
com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext gencont = factory.newWSSGenerationContext();
// Initialize SAML issuer configuration via custom properties
HashMap<Object, Object> customProps = new HashMap<Object,Object>();

customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
customProps.put(SamlConstants.TTL_PROP, "3600000");
customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
gencont.add(customProps); //Add custom properties
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "holder-of-key");
map.put(SamlConstants.Token_REQUEST, "issue");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
map.put(SamlConstants.SERVICE_ALIAS, "soaprecipient");
map.put(SamlConstants.KEY_TYPE,
        "http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey");
map.put(SamlConstants.SAML_APPLIES_TO, "http://localhost:9080/your_Web_service");
map.put(RequesterConfiguration.RSTT.ENCRYPTIONALGORITHM,
        "http://www.w3.org/2001/04/xmlenc#aes256-cbc");
map.put(SamlConstants.KEY_SIZE, "256");
SAMLGenerateCallbackHandler callbackHandler = new
        SAMLGenerateCallbackHandler(map);
SAMLToken samlToken = (SAMLToken) factory.newSecurityToken(SAMLToken.class,
        callbackHandler, "system.wss.generate.saml");
```

   The embedded proof key in the SAML security token is encrypted for the target Web service. The public key of the target service that encrypts the proof key is specified by the `SamlConstants.SERVICE_ALIAS` property which specifies a public certificate in the trust file. The trust file location is specified by a `com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext` custom property. In this example, you must import the Java Cryptography Extension (JCE) policy file because encryption uses 256 bit key size. For more information, read about using the unrestricted JCE policy files in the "Tuning Web Services Security applications" topic.

   If you prefer to use derived keys for digital signing and for encryption instead of using symmetric key directly, add the following name-value pair:

```
map.put(SamlConstants.REQUIRE_DKT, "true");
```

2. Use the `WSSGenerationContext` object to prepare for request message security header processing; for example:

```
gencon.add(samlToken); //this line of code can be omitted

WSSTimestamp timestamp = factory.newWSSTimestamp();
gencon.add(timestamp);

WSSSignature sig = factory.newWSSSignature(samlToken);

sig.setSignatureMethod(WSSSignature.HMAC_SHA1);
sig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
sig.addSignPart(WSSSignature.BODY);
sig.addSignPart(WSSSignature.TIMESTAMP);
sig.addSignPart(WSSSignature.ADDRESSING_HEADERS);
sig.setTokenReference(SecurityToken.REF_KEYID);
//If the gencon.add(samlToken); line of code is omitted, or DerivedKey is used
//the above line of code must be replaced with
//sig.setTokenReference(SecurityToken.REF_STR);

gencon.add(sig);

WSSEncryption enc = factory.newWSSEncryption(samlToken);

enc.setEncryptionMethod(WSSEncryption.AES256);
enc.setTokenReference(SecurityToken.REF_KEYID);
//If the gencon.add(samlToken); line of code is omitted, or DerivedKey is used
//the above line of code must be replaced with
//enc.setTokenReference(SecurityToken.REF_STR);
```

```
enc.encryptKey(false);
enc.addEncryptPart(WSSEncryption.BODY_CONTENT);
enc.addEncryptPart(WSSEncryption.SIGNATURE);
gencon.add(enc);
```

3. Create the `WSSConsumingContext` object to prepare for response message, security header processing; for example:

```
WSSConsumingContext concont = factory.newWSSConsumingContext();

HashMap<Object, Object> map = new HashMap<Object, Object>();

SAMLConsumerCallbackHandler callbackHandler = new
        SAMLConsumerCallbackHandler(map);

WSSDecryption dec = factory.newWSSDecryption(SAMLToken.class, callbackHandler,
        "system.wss.consume.saml");
dec.addAllowedEncryptionMethod(WSSDecryption.AES256);
dec.encryptKey(false);
dec.addRequiredDecryptPart(WSSDecryption.BODY_CONTENT);

concont.add(dec);

callbackHandler = new SAMLConsumerCallbackHandler(map);
WSSVerification ver = factory.newWSSVerification(SAMLToken.class, callbackHandler,
        "system.wss.consume.saml");
ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);
ver.addRequiredVerifyPart(WSSVerification.BODY);
ver.addRequiredVerifyPart(WSSVerification.TIMESTAMP);

concont.add(ver);
```

4. Use the JDK `keytool` utility to generate the `saml-provider.jceks` and `recipient.jceks`files that are used to test the example code; for example:

```
keytool -genkey -alias samlissuer -keystore saml-provider.jceks -dname "CN=SAMLIssuer, O=ACME" -storepass issuerstorepass
 -keypass issuerkeypass -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -genkey -alias soaprecipient -keystore recipient.jceks -dname "CN=SOAPRecipient, O=ACME" -storepass reciptstorepass
 -keypass reciptkeypass -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -export -alias soaprecipient -file reciptpub.cer -keystore recipient.jceks -storepass reciptstorepass -storetype jceks

keytool -import -alias soaprecipient -file reciptpub.cer -keystore saml-provider.jceks -storepass issuerstorepass -storetype jceks
 -keypass issuerkeypass -noprompt
```

Results

You have learned key building blocks to create a web services client application to send a SAML security token in a SOAP message and to use the symmetric key that is embedded in SAML security in message level protection.

Sending self-issued SAML holder-of-key tokens with asymmetric key using WSS APIs:

You can create self-issued SAML tokens with the holder-of-key subject confirmation method and then use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to send these tokens with web services request messages.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about sending self-issued SAML bearer tokens by using WSS APIs.
- Read about sending self-issued SAML sender-vouches tokens by using WSS APIs with message level protection.

About this task

This task focuses on using the asymmetric key that is identified by SAML security tokens to generate a digital signature of selected SOAP message elements in order to satisfy holder-of-key subject confirmation method security requirements. The X.509 certificate of the sender is embedded in the SAML security token. The sender signs selected parts of request message elements by using its corresponding private key and encrypts the request message by using the public key of the recipient. The recipient signs the selected elements of the response message by using the private key of the recipient, and encrypts selected elements of the response message by using the public key of the sender in SAML security tokens. The Web services security policy attached to the web services provider is provided for your reference.

Procedure

1. Create a SAML security token that contains the holder-of-key subject confirmation method; for example:

```
WSSFactory factory = WSSFactory.getInstance();
// Initialize WSSGenerationContext
com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext gencont = factory.newWSSGenerationContext();
// Initialize SAML issuer configuration via custom properties
HashMap<Object, Object> customProps = new HashMap<Object,Object>();
customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
customProps.put(SamlConstants.TTL_PROP, "3600000");
customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
gencont.add(customProps); //Add custom properties
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "holder-of-key");
map.put(SamlConstants.Token_REQUEST, "issue");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
map.put(SamlConstants.KEY_TYPE,
        "http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey");
map.put(SamlConstants.SAML_APPLIES_TO, "http://localhost:9080/your_Web_service");
map.put(SamlConstants.KEY_ALIAS, "soapinitiator" );
map.put(SamlConstants.KEY_NAME, "CN=SOAPInitator, O=ACME");
map.put(SamlConstants.KEY_PASSWORD, "keypass");
map.put(SamlConstants.KEY_STORE_PATH, "keystores/initiator.jceks");
map.put(SamlConstants.KEY_STORE_PASSWORD, "storepass");
map.put(SamlConstants.KEY_STORE_TYPE, "jceks");
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SAMLToken samlToken = (SAMLToken) factory.newSecurityToken(SAMLToken.class,
                      callbackHandler, "system.wss.generate.saml");
```

   The private key of the sender is specified by the `SamlConstants.KEY_ALIAS` property and is used to sign selected elements of the request message.

2. Use the `WSSGenerationContext` object to prepare for request message security header processing; for example:

```
gencon.add(samlToken); //this line of code can be omitted
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencon.add(timestamp);
WSSSignature sig = factory.newWSSSignature(samlToken);
sig.setTokenReference(SecurityToken.REF_KEYID);
//If the gencon.add(samlToken); line of code is omitted,
//the above line of code must be replaced with
//sig.setTokenReference(SecurityToken.REF_STR);

sig.setSignatureMethod(WSSSignature.RSA_SHA1);
sig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
sig.addSignPart(WSSSignature.BODY);
sig.addSignPart(WSSSignature.TIMESTAMP);
sig.addSignPart(WSSSignature.ADDRESSING_HEADERS);
gencon.add(sig);
```

```
X509GenerateCallbackHandler x509callbackHandler2 = new X509GenerateCallbackHandler(
        null,
        "keystores/initiator.jceks",
        "jceks",
        "storepass".toCharArray(),
        "soaprecipient",
        null,
        "", null);
SecurityToken st2 = factory.newSecurityToken(X509Token.class, x509callbackHandler2);
WSSEncryption enc = factory.newWSSEncryption(st2);
enc.addEncryptPart(WSSEncryption.BODY_CONTENT);
enc.addEncryptPart(WSSEncryption.SIGNATURE);
enc.setEncryptionMethod(WSSEncryption.AES256);
enc.setKeyEncryptionMethod(WSSEncryption.KW_RSA_OAEP);
gencon.add(enc);
```

In this example, encryption uses a 256 bit key size so you must import the Java Cryptography Extension (JCE) policy file. For more information, read about using the unrestricted JCE policy files in the “Tuning Web Services Security applications” topic.

3. Create the `WSSConsumingContext` object to prepare for response message security header processing; for example:

```
WSSConsumingContext concont = factory.newWSSConsumingContext();

HashMap<Object, Object> map = new HashMap<Object, Object>();

SAMLConsumerCallbackHandler callbackHandler = new SAMLConsumerCallbackHandler(map);
WSSDecryption dec = factory.newWSSDecryption(SAMLToken.class, callbackHandler,
                            "system.wss.consume.saml");
dec.addAllowedEncryptionMethod(WSSDecryption.AES256);
dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_RSA_OAEP);
dec.encryptKey(false);
dec.addRequiredDecryptPart(WSSDecryption.BODY_CONTENT);
concont.add(dec);
X509ConsumeCallbackHandler verHandler = new X509ConsumeCallbackHandler(null,
            "keystores/initiator.jceks",
            "jceks",
            "storepass".toCharArray(),
            "soaprecipient",
            null, null);
WSSVerification ver = factory.newWSSVerification(X509Token.class, verHandler);
ver.addRequiredVerifyPart(WSSVerification.BODY);
concont.add(ver);
```

4. Use the JDK **keytool** utility to generate the `saml-provider.jceks`, `initiator.jceks`, and `recipient.jceks`files that are used to test the example code; for example:

```
keytool -genkey -alias samlissuer -keystore saml-provider.jceks -dname "CN=SAMLIssuer, O=ACME" -storepass storepass -keypass keypass
 -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -genkey -alias soaprecipient -keystore recipient.jceks -dname "CN=SOAPRecipient, O=ACME" -storepass storepass -keypass keypass
 -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -genkey -alias soapinitiator -keystore initiator.jceks -dname "CN=SOAPInitator, O=ACME" -storepass storepass -keypass keypass
 -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -export -alias samlissuer -file issuerpub.cer -keystore saml-provider.jceks -storepass storepass -storetype jceks
keytool -export -alias soaprecipient -file reciptpub.cer -keystore recipient.jceks -storepass storepass -storetype jceks
keytool -export -alias soapinitiator -file initatpub.cer -keystore initiator.jceks -storepass storepass -storetype jceks

keytool -import -alias samlissuer -file issuerpub.cer -keystore initiator.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt
keytool -import -alias soaprecipient -file reciptpub.cer -keystore initiator.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt

keytool -import -alias samlissuer -file issuerpub.cer -keystore recipient.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt
keytool -import -alias soapinitiator -file initatpub.cer -keystore recipient.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt

keytool -import -alias soapinitiator -file initatpub.cer -keystore saml-provider.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt
```

Results

You have learned key building blocks to create a web services client application to send a SAML security token in a SOAP message and to use the asymmetric key that is embedded in SAML security in message level protection.

Example

The following example illustrates the web services provider Web services security policy:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:spe="http://www.ibm.com/xmlns/prod/websphere/200605/ws-securitypolicy-ext">
    <wsp:Policy wsu:Id="response:app_encparts">
        <sp:EncryptedElements>
            <sp:XPath>/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
             and local-name()='Envelope']/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
             and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
             and local-name()='Security']/*[namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and local-name()='Signature']</sp:XPath>
            <sp:XPath>/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
             and local-name()='Envelope']/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
             and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
             and local-name()='Security']/*[namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and local-name()='Signature']</sp:XPath>
        </sp:EncryptedElements>
        <sp:EncryptedParts>
            <sp:Body/>
        </sp:EncryptedParts>
    </wsp:Policy>
    <wsp:Policy wsu:Id="request:req_enc">
        <sp:EncryptedParts>
            <sp:Body/>
        </sp:EncryptedParts>
    </wsp:Policy>
    <wsp:Policy wsu:Id="request:app_signparts">
        <sp:SignedParts>
            <sp:Body/>
            <sp:Header Namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
            <sp:Header Namespace="http://www.w3.org/2005/08/addressing"/>
        </sp:SignedParts>
        <sp:SignedElements>
            <sp:XPath>/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
             and local-name()='Envelope']/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
             and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
             and local-name()='Security']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd'
             and local-name()='Timestamp']</sp:XPath>
            <sp:XPath>/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
            and local-name()='Envelope']/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
            and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
            and local-name()='Security']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd'
            and local-name()='Timestamp']</sp:XPath>
        </sp:SignedElements>
    </wsp:Policy>
    <wsp:Policy wsu:Id="response:resp_sig">
        <sp:SignedParts>
            <sp:Body/>
        </sp:SignedParts>
    </wsp:Policy>
    <sp:AsymmetricBinding>
        <wsp:Policy>
            <sp:InitiatorToken>
                <wsp:Policy>
                    <spe:CustomToken sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/IncludeToken/Always"/>
                        <wsp:Policy>
                            <spe:WssCustomToken localname="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0"/>
                        </wsp:Policy>
                    </spe:CustomToken>
                </wsp:Policy>
            </sp:InitiatorToken>
            <sp:AlgorithmSuite>
                <wsp:Policy>
                    <sp:Basic256/>
                </wsp:Policy>
            </sp:AlgorithmSuite>
            <sp:IncludeTimestamp/>
            <sp:RecipientToken>
                <wsp:Policy>
                    <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/IncludeToken/Always"/>
                        <wsp:Policy>
                            <sp:WssX509V3Token11/>
                        </wsp:Policy>
                    </sp:X509Token>
                <wsp:Policy>
            <sp:RecipientToken>
            <sp:Layout>
                <wsp:Policy>
                    <sp:Strict/>
                </wsp:Policy>
            </sp:Layout>
        <wsp:Policy>
    <sp:AsymmetricBinding>
</wsp:Policy>
```

Requesting SAML bearer tokens from an external STS using WSS APIs and transport level protection:

You can request SAML tokens with the bearer subject confirmation method from an external Security Token Service (STS). After obtaining the SAML bearer token, you can then send these tokens with web services request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API).

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings.

About this task

You can request a SAML token with the bearer subject confirmation method from an external STS and then send the SAML token in web services request messages from a web services client using WSS APIs.

The web services application client used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code snippets from the sample are described in the procedure section, and a complete, ready-to-use web services client sample is provided in the Example section.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to add the Web Services Security API to pass SAML tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

   b. Obtain the JaxWSServicesSamples client code.

      For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

      The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

   c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

   Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS web services thin client; however, you can also use a managed client.

2. Attach the SAML20 Bearer WSHTTPS default policy set to the web services provider. This policy set is used to protect messages using HTTPS transport. Read about configuring client and provider bindings for the SAML Bearer token for details on how to attach the SAML20 Bearer WSHTTPS default policy set to the web services provider.

3. Assign the SAML Bearer Provider sample default general bindings to the sample web services provider. Read about configuring client and provider bindings for the SAML bearer token for details on assigning the SAML Bearer Provider sample default general bindings to your web services application.

4. Verify that the trustStoreType, trustStorePassword and trustStorePath custom properties correspond to the trust store containing the STS signer certificate. Using the administrative console, complete the following steps:
   a. Click **Services** > **Policy sets** > **General provider policy set bindings** > **Saml Bearer Provider sample** > **WS-Security** > **Authentication and protection**.
   b. Click **gen_saml11token** in the Authentication tokens table.
   c. Click **Callback handler**.
   d. In the Custom Properties section, ensure that the trustStoreType, trustStorePassword and trustStorePath custom properties correspond to the trust store containing the STS signer certificate.
5. Request the SAML token from an external STS. The following code snippet illustrates how to request the SAML token and assumes that an external STS is configured to accept a UsernameToken, and to issue a SAML 2.0 token after validation:

```
//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Add the Thin Client for JAX-WS JAR file to the class path. Add the *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.
   b. Use the WSSFactory newSecurityToken method to request a SAML token from an external STS.

      Specify the following method to request the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

      Requesting a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the Policy Tool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken"
```

      The `SAMLToken.class` parameter specifies the type of security token to create.

      The `callbackHandler` object contains parameters that define the characteristics of the SAML token that you are requesting and other parameters required to reach the STS and obtain the SAML token. The SAMLGenerateCallbackHandler object specifies the configuration parameters described in the following table:

Table 102. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the bearer subject confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `Bearer` confirmation method.	Yes

Table 102. SAMLGenerateCallbackHandler properties (continued). This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the bearer subject confirmation method.

Property	Description	Required
SamlConstants.TOKEN_TYPE	Specifies the token type. When a web services client has policy set attachments, this property is not used by Web Services Security runtime environment. Specify the token value type by using the `valueType` attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 2.0 token; however, you can also use the `WSSConstants.SAML.SAML11_VALUE_TYPE` value.	Yes
SamlConstants.STS_ADDRESS	Specifies the Security Token Service address. For the example used in this task topic, the value of this property is set to `https` to specify to use SSL to protect the SAML Token request. You must set the -Dcom.ibm.SSL.ConfigURL property to enable the use of SSL to protect the SAML token request with the STS.	Yes
SamlConstants.SAML_APPLIES_TO	Specifies the target STS address for where you want to use the SAML token.	No
SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST	Specifies whether to request from the STS a single token that is enclosed in a RequestSecurityToken (RST) element or multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element. The default behavior is to request a single token that is enclosed in a RequestSecurityToken (RST) element from the STS. Specifying a `true` value for this property indicates to request multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element from the STS.	No
SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE	Specifies the WS-Trust namespace that is included in the WS-Trust request.	No

A WSSGenerationContext instance and a WSSConsumingContext instance are also set in the SAMLGenerateCallbackHandler object. The WSSGenerationContext instance must contain a UNTGenerateCallbackHandler object with the information to create the UsernameToken that you want to send to the STS.

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
```

c. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

6. Add the SAML token to the SOAP security header of a web services request messages.
   a. Initialize the web services client and configure the SOAPAction properties. The following code snippet illustrates these actions:

```
// Initialize web services client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);
```

```
// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");
```

b. Initialize the WSSGenerationContext. The following code illustrates the use of the WSSGenerationContext interface to initialize a generation context and enable you to insert the `SAMLToken` into the web services request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

Specifically, the `gencont.add(samlToken)` method call specifies to put the SAML token into a request message. Use the Policy Tool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

c. Add the timestamp element in the SOAP messages security header. The SAML20 Bearer WSHTTPS default policy set requires web services requests and response messages to carry a timestamp element in SOAP messages Security header. In the following code snippet, the `factory.newWSSTimestamp()` method call generates the timestamp, and the `gencont.add(timestamp)` method call specifies the timestamp to put into a request message:

```
// Add a timestamp to the request message.
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencont.add(timestamp);

gencont.process(requestContext);
```

d. Attach WSSGenerationContext object to the web services RequestContext object. The WSSGenerationContext object now contains all the security information that is required to format a request message. The `gencont.process(requestContext)` method call attaches the WSSGenerationContext object to the web services RequestContext object to enable the Web Services Security runtime environment to format the required SOAP security header; for example:

```
// Attaches WSSGenerationContext object to the web services RequestContext object.
gencont.process(requestContext);
```

e. Specify SSL transport level message protection using JVM properties.

The SAML20 Bearer WSHTTPS default policy set requires transport-level message protection using SSL. In addition, you can use this same property to enable protection of the SAML token request to the STS using SSL. Specify SSL transport-level message protection using the following JVM property:

```
-Dcom.ibm.SSL.ConfigURL=file:profile_root\properties\ssl.client.props
```

Alternatively, you can define the SSL configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

Results

You have requested a SAML token with the bearer subject confirmation method with transport level protection from an external STS. After obtaining the token, you sent the token with web services request messages using the JAX-WS programming model and WSS APIs.

If you want to request a SAML token with the bearer subject confirmation method with message level protection from an external STS, see the documentation for requesting SAML sender-vouches tokens from an external STS using WSS APIs and message level protection. To use message level protection for SAML tokens with the bearer subject confirmation method, in the step to request the SAML token from an external STS, specify a confirmation method of `Bearer` instead of `sender-vouches`; for example:

```
//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
```

```
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

Additionally, the step to configure the verification of the digital signature in the response message is optional in the case of the bearer token.

Example

The following code sample is a web services client application that demonstrates how to request a SAML token from an external STS and send that SAML token in web services request messages. If your usage scenario requires SAML tokens, but does not require your application to pass the SAML tokens using web services messages, you only need to use the first part of the following sample code, up through the `// Initialize web services client` section.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */

package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;

import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.UsernameToken;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.util.Map;
import java.util.HashMap;

import javax.xml.ws.BindingProvider;

/**
 * SampleClient
 * main entry point for thin client JAR sample
 * and worker class to communicate with the services
 */
public class SampleClient {

 private String urlHost = "localhost";
 private String urlPort = "9443";
 private static final String CONTEXT_BASE = "/WSSampleSei/";
 private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
 private String message = "HELLO";
 private String uriString = "https://" + urlHost + ":" + urlPort;
 private String endpointURL = uriString + ECHO_CONTEXT12;
 private String input = message;

  /**
  * main()
  *
  * see printusage() for command-line arguments
  *
  * @param args
  */
```

```
 public static void main(String[] args) {
  SampleClient sample = new SampleClient();
  sample.CallService();
 }

 /**
  * CallService Parms were already read. Now call the service proxy classes
  *
  */
 void CallService() {
  String response = "ERROR!:";
  try {
      System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
   System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");

//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);


// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());


         // Initialize web services client
   EchoService12PortProxy echo = new EchoService12PortProxy();
   echo._getDescriptor().setEndpoint(endpointURL);

   // Configure SOAPAction properties
   BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
   Map<String, Object> requestContext = bp.getRequestContext();
   requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
   requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
   requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

   // Initialize WSSGenerationContext
   WSSGenerationContext gencont = factory.newWSSGenerationContext();
         gencont.add(samlToken);

         // Add timestamp
         WSSTimestamp timestamp = factory.newWSSTimestamp();
   gencont.add(timestamp);

         gencont.process(requestContext);

         // Build the input object
   EchoStringInput echoParm =
    new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
   echoParm.setEchoInput(input);
   System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

   // Prepare to consume timestamp in response message.
   WSSConsumingContext concont = factory.newWSSConsumingContext();
       concont.add(WSSConsumingContext.TIMESTAMP);
       concont.process(requestContext);

   // Call the service
   response = echo.echoOperation(echoParm).getEchoResponse();

   System.out.println(">> CLIENT: SEI Echo invocation complete.");
   System.out.println(">> CLIENT: SEI Echo response is: " + response);
  } catch (Exception e) {
   System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
   e.printStackTrace();
  }
 }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _191EBC44865015D9AB1270745072344
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to https://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Requesting SAML sender-vouches tokens from an external STS using WSS APIs and message level protection:

You can request SAML tokens with the sender-vouches subject confirmation method from an external Security Token Service (STS). After obtaining the SAML sender-vouches token, you can then send these tokens with web services request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API) with message level protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, SSL transport protection, X.509 security token, and the use of policy sets to configure and administer web services settings.

About this task

You can request a SAML token with the sender-vouches subject confirmation method from an external STS and then send the SAML token in web services request messages from a web services client using WSS APIs with message level protection.

This product does not provide a default policy set that requires SAML tokens with sender-vouches subject confirmation method. Read about configuring client and provider bindings for the SAML sender-vouches token to learn more about how to create a Web Services Security policy to require SAML tokens with sender-vouches subject confirmation and how to create a custom binding configuration. You must attach the policy and binding to the web services provider. The code sample described in this task assumes that the web services provider policy requires that both the SAML tokens and the message bodies are digitally signed by using an X.509 security token.

The web services client application used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code examples from the sample are described in the procedure, and a complete, ready-to-use web services client sample is provided.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to add the Web Services Security API to pass SAML sender-vouches tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

   b. Obtain the JaxWSServicesSamples client code.

      For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS web services thin client; however, you can also use a managed client.

2. Specify to use SSL message-level message protection. Use the following JVM property to specify to use SSL to protect the SAML token request with the STS:

```
-Dcom.ibm.SSL.ConfigURL=file:profile_root\properties\ssl.client.props
```

Alternatively, you can define the SSL configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

3. Add the Thin Client for JAX-WS JAR file to the class path. Add the *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.

4. Request the SAML token from an external STS. The following code snippet illustrates how to request the SAML sender-vouches token and assumes that an external STS is configured to accept a Username token, and to issue a SAML 2.0 token using sender-vouches after validation:

```
//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

a. Use the WSSFactory newSecurityToken method to specify how to request the SAML token from an external STS.

Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

Requesting a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the Policy Tool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken"
```

The `SAMLToken.class` parameter specifies the type of security token to create.

The `callbackHandler` object contains parameters that define the characteristics of the SAMLToken that you are requesting and other parameters required to reach the STS and obtain the SAML token. The SAMLGenerateCallbackHandler object specifies the configuration parameters described in the following table:

Table 103. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the sender-vouches confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `sender-vouches` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Specifies the token type. When a web services client has policy set attachments, this property is not used by the Web Services Security runtime environment. Specify the token value type by using the `valueType` attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 1.1 token; however, you can also use the `WSSConstants.SAML.SAML20_VALUE_TYPE` value.	Yes
SamlConstants.STS_ADDRESS	Specifies the Security Token Service address. For the example used in this task topic, the value of this property is set to `https` to specify to use SSL to protect the SAML Token request. You must set the -Dcom.ibm.SSL.ConfigURL property to enable the use of SSL to protect the SAML token request with the STS.	Yes
SamlConstants.SAML_APPLIES_TO	Specifies the target STS address for where you want to use the SAML token.	No
SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST	Specifies whether to request from the STS a single token that is enclosed in a RequestSecurityToken (RST) element or multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element. The default behavior is to request a single token that is enclosed in a RequestSecurityToken (RST) element from the STS. Specifying a `true` value for this property indicates to request multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element from the STS.	No
SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE	Specifies the WS-Trust namespace that is included in the WS-Trust request. The default value is WSTrust 1.3.	No

A WSSGenerationContext instance and a WSSConsumingContext instance are also set in the SAMLGenerateCallbackHandler object. The WSSGenerationContext instance must contain a UNTGenerateCallbackHandler object with the information to create the UsernameToken that you want to send to the STS.

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
```

b. Obtain the token identifier of the created SAML token.

   Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

5. Add the SAML token to the SOAP security header of web services request messages.

a. Initialize the web services client and configure the SOAPAction properties. The following code example illustrates these actions:

```
// Initialize web services client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);

// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

b. Initialize the WSSGenerationContext. The following code snippet illustrates the use of the gencont.object of the WSSGenerationContext type to initialize a generation context to enable you to insert the `SAMLToken` into a web services request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

Specifically, the `gencont.add(samlToken)` method call specifies to put the SAML token into a request message. This operation requires the client code to have the following Java 2 Security permission:

```
permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

6. Add an X.509 token for message protection using the Web Services Security API.

This sample code uses the dsig-sender.ks key file and the SOAPRequester sample key. You must not use the sample key in a production environment. The following code snippet illustrates adding an X.509 token for message protection:

```
// Add an X.509 Token for message protection
    X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
         null,
         "profile_root/etc/ws-security/samples/dsig-sender.ks",
         "JKS",
         "client".toCharArray(),
         "soaprequester",
         "client".toCharArray(),
         "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

   SecurityToken x509 = factory.newSecurityToken(X509Token.class,
   x509callbackHandler, "system.wss.generate.x509");

   WSSSignature sig = factory.newWSSSignature(x509);
   sig.setSignatureMethod(WSSSignature.RSA_SHA1);

   WSSSignPart sigPart = factory.newWSSSignPart();
   sigPart.setSignPart(samlToken);
   sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
   sig.addSignPart(sigPart);
   sig.addSignPart(WSSSignature.BODY);
```

a. Create a WSSSignature object with the X509 token. The following line of code creates a WSSSignature object with the X509 token:

```
WSSSignature sig = factory.newWSSSignature(x509);
```

b. Add the signed part to use for message protection. The following line of code specifies to add WSSSignature.BODY as the signed part:

```
sig.addSignPart(WSSSignature.BODY);
```

c. Add the timestamp element in the SOAP messages security header. The SAML20 SenderVouches WSHTTPS and SAML11 SenderVouches WSHTTPS policy sets require web services requests and response messages to carry a timestamp element in the SOAP messages Security header. In the following code snippet, the `factory.newWSSTimestamp()` method call generates the timestamp, and the `gencont.add(timestamp)` method call adds the timestamp into the request message:

```
// Add Timestamp
WSSTimestamp timestamp = factory.newWSSTimestamp();
   gencont.add(timestamp);
sig.addSignPart(WSSSignature.TIMESTAMP);

gencont.add(sig);


WSSConsumingContext concont = factory.newWSSConsumingContext();
```

d. Configure the SAML token signature using STR-Transform transform algorithm.

A separate WSSSignPart is needed to specify the SecurityTokenReference transformation algorithm that is represented by the `WSSSignPart.TRANSFORM_STRT10` attribute. A SAML Token cannot be digitally signed directly. This attribute enables the Web Services Security runtime environment to generate a SecurityTokenReference element to reference the SAMLToken and to digitally sign the SAMLToken using the SecurityTokenReference transformation. The following line of code specifies to use the `WSSSignPart.TRANSFORM_STRT10` attribute:

```
WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(samlToken);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
```

e. Attach the WSSGenerationContext object to the web services RequestContext object. The WSSGenerationContext object now contains all the security information that is required to format a request message. The `gencont.process(requestContext)` method call attaches the WSSGenerationContext object to the web services RequestContext object to enable the Web Services Security runtime environment to format the required SOAP security header; for example:

```
// Attaches the WSSGenerationContext object to the web services RequestContext object.
gencont.process(requestContext);
```

7. Use the X.509 token to validate the digital signature and the integrity of the response message. If the provider policy requires the response message to be digitally signed, you must initialize the X.509 token.

a. A X509ConsumeCallbackHandler object is initialized with a truststore, dsig-receiver.ks, and a certificate path object to validate the provider digital signature. The following line of code is used to initialize the X509ConsumeCallbackHandler object:

```
X509ConsumeCallbackHandler callbackHandlerVer = new X509ConsumeCallbackHandler(
 "profile_root/etc/ws-security/samples/dsig-receiver.ks",
          "JKS",
           "server".toCharArray(),
            certList,
                  java.security.Security.getProvider("IBMCertPath"));
```

b. A WSSVerification object is created and the message body is added to the verification object so that the Web Services Security runtime environment validates the digital signature.

The following line of code is used to initialize the WSSVerification object:

```
WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);
```

The WSSConsumingContext object now contains all the security information that is required to format a request message. The `concont.process(requestContext)` method call attaches the WSSConsumingContext object to the response method; for example:

```
// Attaches the WSSConsumingContext object to the web services RequestContext object.
concont.process(requestContext);
```

Results

You have requested a SAML token with the sender-vouches confirmation method from an external STS. After obtaining the token, you sent the token with web services request messages using message level protection using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a complete, ready-to-use web services client application that demonstrates how to request a SAML token from an external STS and send that SAML token in web services request messages with message level protection. This sample code illustrates the procedure steps described previously.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */
package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
```

```
import com.ibm.was.wssample.sei.echo.EchoStringInput;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.UsernameToken;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.WSSException;
import com.ibm.websphere.wssecurity.wssapi.signature.WSSSignPart;
import com.ibm.websphere.wssecurity.wssapi.signature.WSSSignature;
import com.ibm.websphere.wssecurity.wssapi.verification.WSSVerification;
import com.ibm.websphere.wssecurity.wssapi.token.X509Token;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.security.InvalidAlgorithmParameterException;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.cert.CertStore;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.CollectionCertStoreParameters;
import java.security.cert.X509Certificate;
import java.util.HashSet;
import java.util.Set;
import java.util.HashMap;
import java.util.Map;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
 private String urlHost = "localhost";
 private String urlPort = "9080";
 private static final String CONTEXT_BASE = "/WSSampleSei/";
 private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
 private String message = "HELLO";
 private String uriString = "http://" + urlHost + ":" + urlPort;
 private String endpointURL = uriString + ECHO_CONTEXT12;
 private String input = message;

 /**
  * main()
  *
  * see printusage() for command-line arguments
  *
  * @param args
  */
 public static void main(String[] args) {
  SampleSamlSVClient sample = new SampleSamlSVClient();
  sample.CallService();
 }

 /**
  * CallService Parms were already read. Now call the service proxy classes.
  *
  */
 void CallService() {
  String response = "ERROR!:";
  try {
       System.setProperty("com.ibm.SSL.ConfigURL", "profile_root/properties/ssl.client.props");
       System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas.conf");


//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);
```

```
gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());

         // Initialize web services client.
   EchoService12PortProxy echo = new EchoService12PortProxy();
   echo._getDescriptor().setEndpoint(endpointURL);

   // Configure SOAPAction properties
   BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
   Map<String, Object> requestContext = bp.getRequestContext();
   requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
   requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
   requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

   // Initialize WSSGenerationContext
   WSSGenerationContext gencont = factory.newWSSGenerationContext();
         gencont.add(samlToken);

         // Add X.509 Tokens for message protection
            X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
           null,
           "profile_root/etc/ws-security/samples/dsig-sender.ks",
           "JKS",
           "client".toCharArray(),
           "soaprequester",
           "client".toCharArray(),
           "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

            SecurityToken x509 = factory.newSecurityToken(X509Token.class,
            x509callbackHandler, "system.wss.generate.x509");

            WSSSignature sig = factory.newWSSSignature(x509);
            sig.setSignatureMethod(WSSSignature.RSA_SHA1);

            WSSSignPart sigPart = factory.newWSSSignPart();
            sigPart.setSignPart(samlToken);
            sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
            sig.addSignPart(sigPart);
            sig.addSignPart(WSSSignature.BODY);

         // Add timestamp
         WSSTimestamp timestamp = factory.newWSSTimestamp();
   gencont.add(timestamp);
         sig.addSignPart(WSSSignature.TIMESTAMP);

         gencont.add(sig);


         WSSConsumingContext concont = factory.newWSSConsumingContext();

         // Prepare to consume timestamp in response message
       concont.add(WSSConsumingContext.TIMESTAMP);

       // Prepare to verify digital signature in response message
         X509Certificate x509cert = null;
         try {
                 InputStream is = new FileInputStream("profile_root/etc/ws-security/samples/intca2.cer");
                 CertificateFactory cf = CertificateFactory.getInstance("X.509");
                 x509cert = (X509Certificate) cf.generateCertificate(is);
         } catch (FileNotFoundException e1) {
                 throw new WSSException(e1);
         } catch (CertificateException e2) {
                 throw new WSSException(e2);
         }
         Set<Object> eeCerts = new HashSet<Object>();
         eeCerts.add(x509cert);

         java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
         CollectionCertStoreParameters certparam = new CollectionCertStoreParameters(eeCerts);

         CertStore cert = null;
         try {
                 cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
         } catch (NoSuchProviderException e1) {
                 throw new WSSException(e1);
         } catch (InvalidAlgorithmParameterException e2) {
                 throw new WSSException(e2);
         } catch (NoSuchAlgorithmException e3) {
                 throw new WSSException(e3);
         }
         if (certList != null) {
                 certList.add(cert);
         }
```

```
        X509ConsumeCallbackHandler callbackHandlerVer = new X509ConsumeCallbackHandler(
          "profile_root/etc/ws-security/samples/dsig-receiver.ks",
          "JKS",
          "server".toCharArray(),
          certList,
          java.security.Security.getProvider("IBMCertPath"));

        WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);

        ver.addRequiredVerifyPart(WSSVerification.BODY);
        concont.add(ver);

        gencont.process(requestContext);
        concont.process(requestContext);

        // Build the input object
   EchoStringInput echoParm =
    new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
   echoParm.setEchoInput(input);
   System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

   // Call the service
   response = echo.echoOperation(echoParm).getEchoResponse();

   System.out.println(">> CLIENT: SEI Echo invocation complete.");
   System.out.println(">> CLIENT: SEI Echo response is: " + response);
  } catch (Exception e) {
   System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
   e.printStackTrace();
  }
 }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _6CDDF0DBF91C044D211271166233407
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to http://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Requesting SAML sender-vouches tokens from an external STS using WSS APIs and transport level protection:

You can request SAML tokens with the sender-vouches subject confirmation method from an external Security Token Service (STS). After obtaining the SAML sender-vouches token, you can then send these tokens with web services request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API) with transport level protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, SSL transport protection, and the use of policy sets to configure and administer web services settings.

About this task

You can request a SAML token with the sender-vouches subject confirmation method from an external STS and then send the SAML token in web services request messages from a web services client using WSS APIs with transport level protection.

The web services client application used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code examples from the sample are described in the procedure, and a complete, ready-to-use web services client sample is provided.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

To obtain and modify the sample web services client to add the Web Services Security API to pass SAML sender-vouches tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

b. Obtain the JaxWSServicesSamples client code.

   For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

   The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS Web services thin client; however, you can also use a managed client.

2. Create a copy of either the SAML20 Bearer WSHTTPS default policy set or the SAML11 Bearer WSHTTPS default policy set.

   Provide a name for the copy of the policy set; for example SAML20 SenderVouches WSHTTPS or SAML11 SenderVouches WSHTTPS to help you identify that this new policy set uses the sender-vouches confirmation method.

   No additional change is required to the new policy file because the subject confirmation method is specified in the binding configuration and not in the policy.

   The new policy file contains either SAMLToken20Bearer or the SAMLToken11Bearer as the policy identifiers. Change the identifier of the SAMLToken20Bearer policy to SAMLToken20SV or change the identifier of the SAMLToken11Bearer policy to SAMLToken11SV to specify a more descriptive name. Changing the identifier of the policy does not change the policy enforcement in any way; however, adding a descriptive identifier helps you to identify that these policy identifiers use the sender-vouches confirmation method.

   If you want to view the settings of these policies, use the administrative console to complete the following actions:

   a. Click **Services** > **Policy sets** > **Application policy sets** > ***policy_set_name***.
   b. Click the **WS-Security** policy in the policies table.
   c. Click the **Main policy** link or the **Bootstrap policy** link.
   d. Click **Request token policies** from the Policy Details section.

3. Attach the new SAML20 SenderVouches WSHTTPS or SAML11 SenderVouches WSHTTPS policy set to the web services provider application. Read about configuring client and provider bindings for the SAML sender-vouches token for details on attaching this policy set to your web services provider application.

4. Create a copy of the SAML Bearer Provider sample default general bindings.

   a. For the new copy of the default policy set, provide a name that includes sender-vouches, such as `SAML Sender-vouches provider binding`.
   b. In the callback handler of your SAML11 or SAML20 token consumer, change the value of the confirmationMethod property to `sender-vouches` in the token consumer configuration for the intended SAML token version. Ensure that the custom properties trustStoreType, trustStorePassword and trustStorePath correspond to the trust store containing the STS signer

certificate. Read about configuring client and provider bindings for the SAML sender-vouches token for details on modifying the sender-vouches bindings to satisfy the vouching requirement.

5. Assign the new provider binding to the JaxWSServicesSamples provider sample. Read about configuring client and provider bindings for the SAML sender-vouches for details on assigning the SAML sender-vouches provider sample, default general bindings to your web services provider application.
6. Enable the web services provider SSL configuration attribute, clientAuthentication, to require X.509 client certificate authentication.

   The clientAuthentication attribute determines whether SSL client authentication is required. To specify the clientAuthentication attribute, use the administrative console to complete the following actions:

   a. Click **Security** > **SSL certificates and key management** > **Manage endpoint security configurations** > **{Inbound | Outbound}** > ***SSL_configuration***.
   b. Click the **WC_defaulthost_secure** link in the inbound topology.
   c. From Related Items, click the **SSL_configurations** link.
   d. Select the **NodeDefaultSSLSettings** resource.
   e. Click **Quality of protection (QoP) settings** link.
   f. Select **Required** from the menu to specify client authentication.

   Read about creating a secure sockets layer configuration to learn more about configuring the clientAuthentication attribute.
7. Specify to use SSL transport-level message protection. Use the following JVM property to specify to use SSL to protect the SAML token request with the STS:

```
-Dcom.ibm.SSL.ConfigURL=file:profile_root\properties\ssl.client.props
```

   Alternatively, you can define the SSL configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

8. Add the Thin Client for JAX-WS JAR file to the class path. Add the *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.
9. Request the SAML token from an external STS. The following code snippet illustrates how to request the SAML sender-vouches token and assumes that an external STS is configured to accept a UsernameToken, and to issue a SAML 1.1 token using sender-vouches after validation:

```
//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Use the WSSFactory newSecurityToken method to specify how to request the SAML token from an external STS.

      Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

Requesting a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the Policy Tool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken"
```

The `SAMLToken.class` parameter specifies the type of security token to create.

The `callbackHandler` object contains parameters that define the characteristics of the SAMLToken that you are requesting and other parameters required to reach the STS and obtain the SAML token. The SAMLGenerateCallbackHandler object specifies the configuration parameters described in the following table:

Table 104. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the sender-vouches confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `sender-vouches` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Specifies the token type. When a web services client has policy set attachments, this property is not used by the Web Services Security runtime environment. Specify the token value type by using the `valueType` attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 1.1 token; however, you can also use the `WSSConstants.SAML.SAML20_VALUE_TYPE` value.	Yes
SamlConstants.STS_ADDRESS	Specifies the Security Token Service address. For the example used in this task topic, the value of this property is set to `https` to specify to use SSL to protect the SAML Token request. You must set the -Dcom.ibm.SSL.ConfigURL property to enable the use of SSL to protect the SAML token request with the STS.	Yes
SamlConstants.SAML_APPLIES_TO	Specifies the target STS address for where you want to use the SAML token.	No
SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST	Specifies whether to request from the STS a single token that is enclosed in a RequestSecurityToken (RST) element or multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element. The default behavior is to request a single token that is enclosed in a RequestSecurityToken (RST) element from the STS. Specifying a `true` value for this property indicates to request multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element from the STS.	No
SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE	Specifies the WS-Trust namespace that is included in the WS-Trust request. The default value is WSTrust 1.3.	No

A WSSGenerationContext instance and a WSSConsumingContext instance are also set in the SAMLGenerateCallbackHandler object. The WSSGenerationContext instance must contain a UNTGenerateCallbackHandler object with the information to create the UsernameToken that you want to send to the STS.

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
```

b. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

Results

You have requested a SAML token with the sender-vouches confirmation method from an external STS. After obtaining the token, you sent the token with web services request messages using transport protection using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a complete, ready-to-use web services client application that demonstrates how to request a SAML token from an external STS and send that SAML token in web services request messages with transport level protection. This sample code illustrates the procedure steps described previously.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */
package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;

import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.UsernameToken;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.util.Map;
import java.util.HashMap;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
 private String urlHost = "localhost";
 private String urlPort = "9443";
 private static final String CONTEXT_BASE = "/WSSampleSei/";
 private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
 private String message = "HELLO";
 private String uriString = "https://" + urlHost + ":" + urlPort;
 private String endpointURL = uriString + ECHO_CONTEXT12;
 private String input = message;

 /**
  * main()
  *
  * see printusage() for command-line arguments
  *
  * @param args
  */
 public static void main(String[] args) {
  SampleSamlSVClient sample = new SampleSamlSVClient();
  sample.CallService();
 }

 /**
  * CallService Parms were already read. Now call the service proxy classes.
  *
  */
 void CallService() {
  String response = "ERROR!:";
  try {

        System.setProperty("com.ibm.SSL.ConfigURL", "profile_root//properties/ssl.client.props");
        System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");


//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
```

```
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());


         // Initialize web services client
   EchoService12PortProxy echo = new EchoService12PortProxy();
   echo._getDescriptor().setEndpoint(endpointURL);

   // Configure SOAPAction properties
   BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
   Map<String, Object> requestContext = bp.getRequestContext();
   requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
   requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
   requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

   // Initialize WSSGenerationContext
   WSSGenerationContext gencont = factory.newWSSGenerationContext();
         gencont.add(samlToken);


         // Add timestamp
         WSSTimestamp timestamp = factory.newWSSTimestamp();
   gencont.add(timestamp);

         gencont.process(requestContext);

         // Build the input object
   EchoStringInput echoParm =
    new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
   echoParm.setEchoInput(input);
   System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

   // Prepare to consume timestamp in response message
   WSSConsumingContext concont = factory.newWSSConsumingContext();
       concont.add(WSSConsumingContext.TIMESTAMP);
       concont.process(requestContext);

   // Call the service
   response = echo.echoOperation(echoParm).getEchoResponse();

   System.out.println(">> CLIENT: SEI Echo invocation complete.");
   System.out.println(">> CLIENT: SEI Echo response is: " + response);
  } catch (Exception e) {
   System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
   e.printStackTrace();
     }
 }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _6CDDF0DBF91C044D211271166233407
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to http://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Requesting SAML holder-of-key tokens with symmetric key from external security token service using WSS APIs:

You can request an external security token service (STS) to issue SAML tokens with the holder-of-key subject confirmation method with symmetric key that is encrypted for a target service. Use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to complete this task.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about propagating self-issued SAML holder-of-key tokens with symmetric key by using WSS APIs.
- Become familiar with using embedded key materials in SAML tokens for message protection by using WSS APIs. Your usage scenario requires requesting SAML tokens from an external STS instead of using self-issued SAML tokens.
- Read about requesting SAML sender-vouches tokens from an external STS to propagate by using WSS APIs with message level protection.
- Read about requesting SAML sender-vouches tokens from an external STS to propagate by using WSS APIs with transport level protection.
- Read about requesting SAML bearer tokens from an external STS to propagate by using WSS APIs with transport level protection.
- Be familiar with accessing an external STS by using WSS APIs.

About this task

This task shows example code to request SAML tokens from an external STS, with holder-of-key subject confirmation method and embedded symmetric key that is encrypted for the target service by using WSS APIs. This task focuses on sending a WS-Trust request message to an external STS to request SAML holder-of-key tokens with symmetric keys.

Procedure

1. Specify an STS from which to request a SAML security token that contains holder-of-key subject confirmation method; for example:

```
com.ibm.websphere.wssecurity.wssapi.WSSFactory factory =
    com.ibm.websphere.wssecurity.wssapi.WSSFactory.getInstance();
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, "https://www.example.com/sts");  //STS URL
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, "http://myhost:9080/myService");  //Target Service
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_SOAP_VERSION, "1.1");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
                  "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST,
              "true");   //RST or RSTC
cbackMap1.put(SamlConstants.TOKEN_TYPE,
              "http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0");
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "holder-of-key");
```

   To request a holder-of-key SAML security token from the STS, you must specify whether to embed a symmetric key or a public key by way of a `KeyType` element in a trust request. This example requires a symmetric key type as shown in the next step.

2. Specify the symmetric key to be embedded in SAML security tokens; for example:

```
cbackMap1.put(SamlConstants.KEY_TYPE,
             "http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);
cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);

SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class,
                                               cbHandler1, "system.wss.generate.saml");
```

The requested SAML token contains a symmetric key that is encrypted for the target service. The STS also returns the unencrypted symmetric key through the `WS-Trust RequestedProofToken` element. See the following example.

```
<wst:RequestedProofToken>
    <wst:BinarySecret
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
      wsu:Id="_27325D34CE4BCC83141288966548620">n68rFQba+XTZLNBfEc4prg==</wst:BinarySecret>
</wst:RequestedProofToken>
```

The `RequestedProofToken` element is shown here for your information. The detailed processing is not exposed to WSS APIs users. The `RequestedProofToken` element and the symmetric key are handled by the Web Services Security runtime environment, or more precisely by the `SAMLGenerateLoginModule` that is specified in the `system.wss.geenrate.saml` JAAS login configuration.

Results

You have learned key building blocks for requesting SAML tokens with holder-of-key subject confirmation method and symmetric key from an external STS by using WSS APIs. To use the SAML token to sign request messages, review the example code in the "Propagating self-issued SAML holder-of-key tokens with symmetric key by using WSS APIs" topic.

Requesting SAML holder-of-key tokens with asymmetric key from External Security Token Service using WSS APIs:

You can request an external Security Token Service (STS) to issue SAML tokens with the holder-of-key subject confirmation method with a public key in an X.509 certificate with the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs).

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about propagating self-issued SAML holder-of-key tokens with asymmetric key by using WSS APIs.
- Become familiar with using embedded key materials in SAML tokens for message protection by using WSS APIs. Your usage scenario requires requesting SAML tokens from an external STS instead of using self-issued SAML tokens.
- Read about requesting SAML sender-vouches tokens from an external STS to propagate by using WSS APIs with message level protection.
- Read about requesting SAML sender-vouches tokens from an external STS to propagate by using WSS APIs with transport level protection.
- Read about requesting SAML bearer tokens from an external STS, which you propagate by using WSS APIs with transport level protection.
- Become familiar with accessing an external STS by using WSS APIs.

About this task

This task shows example code to request SAML tokens with the holder-of-key subject confirmation method and the embedded public key in an X.509 certificate by using WSS APIs, from an external STS. This task focuses on sending an X.509 certificate to an external STS when requesting SAML holder-of-key tokens.

Procedure

1. Specify an STS from which to request a SAML security token that contains holder-of-key subject confirmation method; for example:

```
com.ibm.websphere.wssecurity.wssapi.WSSFactory factory =
    com.ibm.websphere.wssecurity.wssapi.WSSFactory.getInstance();
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, "https://www.example.com/sts");
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, "http://myhost:9080/myService");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_SOAP_VERSION, "1.1");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
              "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST,
              "true");   //RST or RSTC
cbackMap1.put(SamlConstants.TOKEN_TYPE,
              "http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0");
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "holder-of-key");
```

For the holder-of-key subject confirmation method, you must specify whether a public key or a symmetric key is embedded in SAML tokens. This example specifies a public key type. It then specifies the location of a certificate that contains the public key, and the location of the corresponding private key for the sender to digitally sign elements of SOAP messages to satisfy the holder-of-key subject confirmation requirements.

2. Specify the location of an X.509 certificate to embed in SAML tokens and a corresponding private key for using to digitally sign message elements; for example:

```
cbackMap1.put(SamlConstants.KEY_TYPE,
              "http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey");
cbackMap1.put(SamlConstants.KEY_ALIAS, "soapinitiator" );
cbackMap1.put(SamlConstants.KEY_NAME, "CN=SOAPInitator, O=Example");
cbackMap1.put(SamlConstants.KEY_PASSWORD, "keypass");
cbackMap1.put(SamlConstants.KEY_STORE_PATH, "keystores/initiator.jceks");
cbackMap1.put(SamlConstants.KEY_STORE_PASSWORD, "storepass");
cbackMap1.put(SamlConstants.KEY_STORE_TYPE, "jceks");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);
cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);

SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class,
                                                   cbHandler1, "system.wss.generate.saml");
```

The specified X.509 certificate is sent in WS-Trust requests to the external STS in the `trust:UseKey` element. For more information read about SAML assertions defined in the SAML Token Profile standard. SSL is used to protect integrity and confidentiality of WS-Trust request and response messages in this example.

Results

You have learned key building blocks to request SAML tokens with the holder-of-key subject confirmation method and asymmetric key from an external STS using WSS APIs. To use the SAML token to sign request messages, become familiar with the example code in the "Propagating self-issued SAML holder-of-key tokens with asymmetric key by using WSS APIs” topic.

Sending a security token using WSSAPIs with a generic security token login module:

You can request an authentication token from an external *Security Token Service (STS)*, and then send the token with web service request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API), with message or transport level protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, WebSphere web service security generic security token login modules, SSL transport protection, message level protection, and the use of policy sets to configure and administer web services settings.

About this task

The web service client application used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code examples from the sample are described in the procedure, and a complete, ready-to-use web service client sample is provided.

Complete the following steps to request a SAML Bearer authentication token from an external STS and send the token:

Procedure

1. Identify and obtain the web service client that you want to use to invoke a web service provider. Use this client to request and to insert authentication tokens in the SOAP request messages programmatically using WSS APIs. The web service client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web service sample application.

   Complete the following steps to obtain and modify the sample web service client to add the Web Services Security API to pass a security token in the SOAP request message programmatically using WSS APIs:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

   b. Obtain the JaxWSServicesSamples client code. For the purpose of this example, the procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web service Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

      The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

   c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web service client code against the sample application.

      Instead of using the PolicySet for the protection of the web service client sample, you can choose to add the code snippets to pass authentication tokens in the SOAP request message programmatically using WSS APIs in your own web service client application. The example in this procedure uses a JAX-WS web service thin client; however, you can also use a managed client.

   d. Attach the SAML11 Bearer WSHTTPS default policy set to the web services provider. This policy set is used to protect messages using HTTPS transport. Read about configuring client and provider bindings for the SAML Bearer token for details on how to attach the SAML11 Bearer WSHTTPS default policy set to the web services provider.

   e. Assign the SAML Bearer Provider sample default general bindings to the sample web services provider. Read about configuring client and provider bindings for the SAML bearer token for details on assigning the SAML Bearer Provider sample default general bindings to your web services application.

   f. Verify that the trustStoreType, trustStorePassword and trustStorePath custom properties correspond to the trust store containing the STS signer certificate. Complete the following steps by using the administrative console:

      1) Click **Services** > **Policy sets** > **General provider policy set bindings** > **Saml Bearer Provider sample** > **WS-Security** > **Authentication and protection**.

      2) Click **con_saml11token** in the Authentication tokens table.

      3) Click **Callback handler**.

      4) In the Custom Properties section, ensure that the trustStoreType, trustStorePassword and trustStorePath custom properties correspond to the trust store containing the STS signer certificate.

2. If you are using SSL Transport-level protection to protect the web service request or the WS-Trust request, use the following Java virtual machine (JVM) property to set up the SSL configuration.

```
-Dcom.ibm.SSL.ConfigURL=file:<profile_root>\properties\ssl.client.props
```

   Alternatively, you can define the SSL configuration file using a Java system property in the sample client code:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

3. Add the JAR file for the JAX-WS thin client to the class path: `app_server_root/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar`. See the testing web service-enabled clients information for more information about adding this JAR file to the class path.
4. Request the authentication token from an external STS. The following code snippet illustrates how to request the authentication token to be used with WebSphere generic SecurityToken login module, and assumes that an external STS is configured to accept a Username token as authentication token, and to issue a SAML 1.1 token.

```
//Request SecurityToken from external STS:
WSSFactory factory = WSSFactory.getInstance();
//STS URL that issues the requested token
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
//Web services endpoint that receives the issued token
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";

//Begin sample code 1 (Using WS-Trust Issue to request the token from
//the STS in which authentication token is send over WS-Security head):
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(IssuedTokenConfigConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(IssuedTokenConfigConstants.APPLIES_TO, ENDPOINT_URL);
//The following property specifies that the ws-trust request should be
//compliance with WS-Trust 1.3 spec
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
"http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
//This request is made with WS-TRust Issue only (without the use of
//WS-Trust Validate)
cbackMap1.put(IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT, "false");

GenericIssuedTokenGenerateCallbackHandler cbHandler1 =
new GenericIssuedTokenGenerateCallbackHandler (cbackMap1);

//Create the context object for WS-Trust request:
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
// Use UNT for trust request authentication
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);
gencont1.add(ut);
cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
//End of sample code 1.

//Begin sample code 2 (using WS-Trust Validate to request a token by
//exchanging a token in RunAs Subject).
//If web service client has RunAs Subject , for example an
//authenticated intermediate server acts as a client to invoke the
//downstream service, you can program the client to use the token from
//the RunAs subject to exchange with the STS by using WS-Trust validate.
//To do so, you replace sample code 1 with the following:
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(IssuedTokenConfigConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(IssuedTokenConfigConstants.APPLIES_TO, ENDPOINT_URL);
//This request is made with WS-Trust 1.3
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
"http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");

//add the next line if you do not want to fallback to WS-Trust Issue if
//token exchange fails.
cbackMap1.put(IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT_ONLY, "true");

//add the next line to specify the token type in the RunAs subject that
//will be used to exchange the requested token. For example, you use
//the LTPA token to exchange for a SAML token. If the exchanged token
//in the RunAs subject has the same value type as the requested token,
//setting IssuedTokenConfigConstants.USE_TOKEN is not required.
cbackMap1.put(IssuedTokenConfigConstants.USE_TOKEN, LTPAToken.ValueType);

GenericIssuedTokenGenerateCallbackHandler cbHandler1 =
    new GenericIssuedTokenGenerateCallbackHandler (cbackMap1);
//The following codes are added if Authentication token in ws-security
//head or Message level security protection is required. If there is no
//Message level protection or additional authentication token for
//WS-Trust Validate, do not create the context object shown below.
//Context object for WS-Trust request:
```

```
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
// Use UNT for trust request authentication
UNTGenerateCallbackHandler utCallbackHandler =
    new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);
gencont1.add(ut);
cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);

//End of sample code 2.

GenericSecurityToken token = (GenericSecurityToken) factory.newSecurityToken
(GenericSecurityToken.class, cbHandler1, "system.wss.generate.issuedToken");

//The following step to set ValueType is required..
//The parameter is always the QName of the requested token's valueType.
//QName for SAML1.1:
QName Saml11ValueType = new QName(WSSConstants.SAML.SAML11_VALUE_TYPE);
            token.setValueType(Saml11ValueType);

//This article includes QName definitions for  SAML11, SAML20, TAM
//token, and Pass ticket token.
//QName for SAML 2.0:
QName Saml20ValueType = new QName(WSSConstants.SAML.SAML20_VALUE_TYPE);
token.setValueType(Saml11ValueType);
//QName for TAM token:
QName TamValueType = new QName("http://ibm.com/2004/01/itfim/ivcred");
//QName for PassTicket token:
QName PassTicketValueType = new QName("http://docs.oasis-open.org/wss/
2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken");

//You can use the Token interface to get the token ValueType QName for
//all other tokens. For example, a Username Token's QName is //UsernameToken.ValueType.
```

The `GenericIssuedTokenGenerateCallbackHandler` object contains parameters that define the characteristics of the security token that you are requesting, as well as other parameters required to reach the STS and to obtain the security token. The `GenericIssuedTokenGenerateCallbackHandler` object specifies the configuration parameters described in the following table:

Table 105. `GenericIssuedTokenGenerateCallbackHandler` properties. This table describes the configuration parameters for the `GenericIssuedTokenGenerateCallbackHandler` object, and specifies whether or not the property is required.

Property	Description	Required
IssuedTokenConfigConstants.STS_ADDRESS	Specifies the http address of the STS. When communication to the STS is protected with SSL, you must set the -Dcom.ibm.SSL.ConfigURL property. SSL connection to the STS is indicated with an https:// address prefix.	Yes
IssuedTokenConfigConstants.APPLIES_TO	Specifies the target service address for where you want to use the token.	No
IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST	Specifies whether to request a single token from the STS that is enclosed in a RequestSecurityToken (RST) element or multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element. The default behavior is to request a single token that is enclosed in a RequestSecurityToken (RST) element from the STS. Specifying a true value for this property indicates a request for multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element from the STS. The default value is false.	No
IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE	Specifies the WS-Trust namespace that is included in the WS-Trust request. The default value is WSTrust 1.3.	No

Table 105. GenericIssuedTokenGenerateCallbackHandler properties (continued). This table describes the configuration parameters for the GenericIssuedTokenGenerateCallbackHandler object, and specifies whether or not the property is required.

Property	Description	Required
IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT	Specify if you want WS-Security to use the token from the RunAs subject to exchange the requested token first by using WS-Trust Validate. If set to `false`, WS-Security will use WS-Trust Issue to request the token. The default value is `true`.	No
IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT_ONLY	Specify if you do not want WS-Security to use WS-Trust Issue to the requested token if token exchange fails. The default value is `false`.	No
IssuedTokenConfigConstants.USE_TOKEN	Use this value to choose a token from the RunAs subject to exchange the requested token. The default value is the requested token's `ValueType`.	No

A `WSSGenerationContext` instance and a `WSSConsumingContext` instance are also set in the `GenericIssuedTokenGenerateCallbackHandler` object. In this example, the `WSSGenerationContext` instance contains a `UNTGenerateCallbackHandler` object with the information to create the `UsernameToken` that you want to send to the STS.

The `system.wss.generate.issuedToken` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the generic security token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
```

5. Add the requested authentication token from the STS to the SOAP security header of web services request messages.
   a. Initialize the web service client and configure the `SOAPAction` properties. The following code illustrates these actions:

```
// Initialize web service client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);

// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(token);
```

   b. Initialize the `WSSGenerationContext` object. The following code illustrates using the `WSSFactory.newWSSGenerationContext` method to obtain a `WSSGenerationContext` object. The `WSSGenerationContext` object is then used to insert the token into a web service request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(token);
```

   The `WSSGenerationContext.add` method requires the client code to have the following Java 2 Security permission:

```
permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

6. Add an X.509 token for message protection (skip this step if the web service is protected with SSL Transport level protection only). The following sample code uses the `dsig-sender.ks` key file and the

SOAPRequester sample key. You must not use the sample key in a production environment. The following code illustrates adding an X.509 token for message protection:

```
//Add an X.509 Token for message protection
X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
    null,
    "profile_root/etc/ws-security/samples/dsig-sender.ks",
    "JKS",
    "client".toCharArray(),
    "soaprequester",
    "client".toCharArray(),
    "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

SecurityToken x509 = factory.newSecurityToken(X509Token.class,
x509callbackHandler, "system.wss.generate.x509");

WSSSignature sig = factory.newWSSSignature(x509);
sig.setSignatureMethod(WSSSignature.RSA_SHA1);

WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(token);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
sig.addSignPart(sigPart);
sig.addSignPart(WSSSignature.BODY);
```

a. Create a WSSSignature object with the X.509 token. The following line of code creates a WSSSignature object with the X.509 token:

```
WSSSignature sig = factory.newWSSSignature(x509);
```

b. Add the signed part to use for message protection. The following line of code specifies to add WSSSignature.BODY as the signed part:

```
sig.addSignPart(WSSSignature.BODY);
```

c. Add the Timestamp element in the SOAP Security header. The SAML20 SenderVouches WSHTTPS and SAML11 SenderVouches WSHTTPS policy sets require web service requests and responses to contain a Timestamp element in the SOAP Security header. In the following code, the WSSFactory.newWSSTimestamp() method generates a Timestamp element, and the WSSGenerationContext.add(timestamp) method adds the Timestamp element to the request message:

```
// Add Timestamp element
WSSTimestamp timestamp = factory.newWSSTimestamp();
    gencont.add(timestamp);
sig.addSignPart(WSSSignature.TIMESTAMP);

gencont.add(sig);

WSSConsumingContext concont = factory.newWSSConsumingContext();
```

d. Skip this step if token signature is not required. If the requested security token needs to be signed with the *STR Dereference Transform* reference option, follow step 1. Otherwise, follow step 2. The *STR Dereference Transform* reference option is commonly known as *STR-Transform*.

Step 1: Some tokens, including SAML Tokens, cannot be digitally signed directly. You must sign the token using *STR-Transform*. In order for a token to be signed with *STR-Transform*, it must be referenced by a <wsse:SecurityTokenReference> element in the <wsse:Security> header block. To sign a security token with *STR-Transform*, a separate WSSSignPart is created to specify the SecurityTokenReference with a transformation algorithm that is represented by the WSSSignPart.TRANSFORM_STRT10 attribute. This attribute enables the WS-Security runtime environment to generate a SecurityTokenReference element to reference the token, and to digitally sign the token using the *STR Dereference* reference option. The following code illustrates the use of the WSSSignPart.TRANSFORM_STRT10 attribute:

```
WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(token);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
```

Step 2: If the requested signed token is not a SAML token, or *STR-Transform* is not used, use the following code instead:

```
sig.addSignPart(token);
```

e. Attach the WSSGenerationContext object to the web service RequestContext object. The WSSGenerationContext object now contains all of the security information required to format a request message. The WSSGenerationContext.process(requestContext) method attaches the

WSSGenerationContext object to the web service RequestContext object to enable the WS-Security runtime environment to format the required SOAP Security header; for example:

```
// Attaches the WSSGenerationContext object to the web service RequestContext object.
gencont.process(requestContext);
```

7. Use the X.509 token to validate the digital signature and the integrity of the response message if the provider policy requires the response message to be digitally signed. Skip this step if using SSL Transport level protection.
   a. An X509ConsumeCallbackHandler object is initialized with a trust store and a List of certificate path objects to validate the digital signature in a response message. The following code initializes the X509ConsumeCallbackHandler object with dsig-receiver.ks trust store and a certificate path object called certList:

```
ArrayList certList = new ArrayList();
java.security.cert.CertStore certStore = java.security.cert.CertStore.getDefaultType();
certList.add(certStore);

X509ConsumeCallbackHandler callbackHandlerVer = new
    X509ConsumeCallbackHandler("profile_root/etc/ws-security/samples/dsig-receiver.ks",
    "JKS",
    "server".toCharArray(),
    certList,
    java.security.Security.getProvider("IBMCertPath"));
```

   b. A WSSVerification object is created and the message body is added to the verification object so that the WS-Security runtime environment validates the digital signature. The following code is used to initialize the WSSVerification object:

```
WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);
```

   The WSSConsumingContext object now contains all the security information that is required to format a request message. The WSSConsumingContext.process(requestContext) method attaches the WSSConsumingContext object to the response method; for example:

```
// Attaches the WSSConsumingContext object to the web service RequestContext object.
concont.process(requestContext);
```

Results

You have requested a security token from an external STS. After obtaining the token, you sent the token with web services request messages using message level protection using the JAX-WS programming model and WSS APIs.

Example

The following code example is a web service client application that demonstrates how to request a SAML Bearer token from an external STS and send that SAML token in a web service request message. If your usage scenario requires SAML tokens, but does not require your application to pass the SAML tokens using web service messages, you only need to use the first part of the following sample code, up through the // Initialize web service client section.

```
package com.ibm.was.wssample.sei.cli;
import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.websphere.wssecurity.wssapi.token.UsernameToken;
import com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.core.config.IssuedTokenConfigConstants;
import com.ibm.websphere.wssecurity.callbackhandler.GenericIssuedTokenGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.GenericSecurityToken;
import javax.xml.namespace.QName;
import java.util.HashMap;
import java.util.Map;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
    private String urlHost = "yourhost";
    private String urlPort = "9444";
```

```
private static final String CONTEXT_BASE = "/WSSampleSei/";
private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
private String message = "HELLO";
private String uriString = "https://" + urlHost + ":" + urlPort;
private String endpointURL = uriString + ECHO_CONTEXT12;
private String input = message;

/**
 * main()
 *
 * see printusage() for command-line arguments
 *
 * @param args
 */
public static void main(String[] args) {
    SampleSamlSVClient sample = new SampleSamlSVClient();
    sample.CallService();
}

/**
 * CallService Parms were already read. Now call the service proxy classes
 *
 */
void CallService() {
    String response = "ERROR!:";
    try {
        System.setProperty("java.security.auth.login.config",
            "file:/opt/IBM/WebSphere/AppServer/profiles/AppSrv01/properties/wsjaas_client.conf ");
        System.setProperty("com.ibm.SSL.ConfigURL",
            "file:/opt/IBM/WebSphere/AppServer/profiles/AppSrv01/properties/ssl.client.props");

        //Request the SAML Token from external STS
        WSSFactory factory = WSSFactory.getInstance();
        String STS_URI = "https://yourhost:9443/TrustServerWST13/services/RequestSecurityToken";
        String ENDPOINT_URL = "http://localhost:9081/WSSampleSei/EchoService12";

        HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
        cbackMap1.put(IssuedTokenConfigConstants.STS_ADDRESS, STS_URI);
        cbackMap1.put(IssuedTokenConfigConstants.APPLIES_TO, ENDPOINT_URL);
        cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
            "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
        cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
        cbackMap1.put(IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT, "false");

        GenericIssuedTokenGenerateCallbackHandler cbHandler1 =
            new GenericIssuedTokenGenerateCallbackHandler (cbackMap1);

        //Context object for WS-Trust request:
        WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
        WSSConsumingContext concont1 = factory.newWSSConsumingContext();

        // Use UNT for trust request authentication
        UNTGenerateCallbackHandler utCallbackHandler = new
            UNTGenerateCallbackHandler("testuser", "testuserpwd");
        SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);
        gencont1.add(ut);
        cbHandler1.setWSSConsumingContextForTrustClient(concont1);
        cbHandler1.setWSSGenerationContextForTrustClient(gencont1);

        //get generic security token
        GenericSecurityToken token = (GenericSecurityToken) factory.newSecurityToken
            (GenericSecurityToken.class, cbHandler1, "system.wss.generate.issuedToken");
        QName Saml11ValueType = new QName(WSSConstants.SAML.SAML11_VALUE_TYPE);
        token.setValueType(Saml11ValueType);
        System.out.println("SAMLToken id = " + token.getId());

        // Initialize web services client
        EchoService12PortProxy echo = new EchoService12PortProxy();
        echo._getDescriptor().setEndpoint(endpointURL);

        // Configure SOAPAction properties
        BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
        Map<String, Object> requestContext = bp.getRequestContext();
        requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
        requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
        requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

        // Initialize WSSGenerationContext
        WSSGenerationContext gencont = factory.newWSSGenerationContext();
        gencont.add(token);

        // Add timestamp
        WSSTimestamp timestamp = factory.newWSSTimestamp();
        gencont.add(timestamp);
        gencont.process(requestContext);

        // Build the input object
        EchoStringInput echoParm =
            new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
        echoParm.setEchoInput(input);
```

```
            System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

            // Prepare to consume timestamp in response message.
            WSSConsumingContext concont = factory.newWSSConsumingContext();
            concont.add(WSSConsumingContext.TIMESTAMP);
            concont.process(requestContext);

            // Call the service
            response = echo.echoOperation(echoParm).getEchoResponse();

            System.out.println(">> CLIENT: SEI Echo invocation complete.");
            System.out.println(">> CLIENT: SEI Echo response is: " + response);
        } catch (Exception e) {
            System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
            e.printStackTrace();
        }
    }
}
```

Securing messages at the response consumer using WSS APIs:

You can secure SOAP messages with signature verification, decryption, and consumer tokens to protect message integrity, confidentiality, and authenticity, respectively. The response consumer (client-side) configuration defines the Web Services Security requirements for the incoming SOAP response.

About this task

To secure web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. You must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

The response consumer (client-side) configuration requirements involve verifying that the integrity parts are signed and that the signature is verified, verifying that the required confidential parts are encrypted and that the parts are decrypted; and validating the security tokens.

You can use the following methods to configure Web Services Security and to define policy types to secure the SOAP messages:
- Use the administrative console to configure policy sets.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

The following high-level steps use the WSS APIs:

Procedure
- Verify consumer signing information to protect message integrity.
- Configure decryption to protect message confidentiality.
- Validate consumer tokens to protect message authenticity.

Results

After completing these procedures, you have secured messages at the response consumer level.

What to do next

Next, if not already configured, secure messages with signing information, encryption, and generator tokens at the response (client-side) generator level.

Configuring decryption methods to protect message confidentiality using the WSS APIs:

You can configure decryption method information for the response consumer (client side) section of the binding file. Decryption information is used to specify how the consumers (receivers) decrypt incoming

SOAP messages. To configure decryption, specify which message parts to decrypt and specify which algorithm methods and security tokens are to be used for decryption.

Before you begin

Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. Prior to configuring decryption, familiarize yourself with XML encryption.

About this task

For decryption, you must specify the following:
- Which parts of the message are to be decrypted.
- Which decryption algorithms to specify.

To configure decryption and decrypted parts on the client side, use the WSSDecryption and WSSDecryptPart APIs, or configure policy sets using the administrative console.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

WebSphere Application Server uses decryption information for the default consumer to decrypt parts of the SOAP message. The WSSDecryption API configures the following required parts as decrypted parts.

Table 106. Required decrypted parts. Use the decryption information to specify how incoming messages are decrypted.

Decryption parts	Description
Keywords	Keywords are used to add the decrypted parts to the SOAP message.
XPath expression	XPath expressions are used to add the decrypted parts to the SOAP message.
WSSDencryptPart object	This object adds the decrypted parts to the SOAP message.
WSSVerification object	This object adds the signature verification component as a decrypted part.
Header	This part adds the header in the SOAP header, specified by QName, as a decrypted part.
Security token object	This object adds the security token as a decrypted part.

Web Services Security API (WSS API) supports symmetric encryption, by using a shared key, only when Web Services Secure Conversation (WS-SecureConversation) is used.

The WSS APIs allow the use of either keywords or an XPath expression to specify the parts of the SOAP message that are to be decrypted. WebSphere Application Server supports the use of the following keywords:

Table 107. Supported decryption keywords. Use the keywords to decrypt incoming messages.

Keyword	References
BODY_CONTENT	The keyword for the body contents of the SOAP message body as a decryption target.
SIGNATURE	The keyword for the signature element as a decryption target.
USERNAME_TOKEN,	The keyword for the Username token element as a decryption target.

If configuring using the WSS APIs, the WSSDecryption and WSSDecryptPart APIs complete these high-level steps:

Procedure

1. Use the WSSDecryption API to configure encryption. The WSSDecryption API performs these tasks by default:

a. Generates the callback handler.
b. Generates the consumer security token object.
c. Adds the security token reference type.
d. Adds the WSSEncryptPart object.
e. Adds the parts to be encrypted. Adds the default parts for decryption by using keywords and XPath expressions.
f. Adds the verification component.
g. Adds the header in the SOAP message, specified by QName.
h. Sets the default data encryption method.
i. Specifies whether the key is to be decrypted using a Boolean value. Calls this method when the shared key is encrypted.
j. Sets the default key encryption method.

2. Use the WSSEncryptPart API to configure encrypted parts or add a transform method. The WSSEncryptPart API performs these tasks by default:
   a. Sets the encrypted parts specified by using keywords or an XPath expression.
   b. Sets the encrypted parts specified by an XPath expression.
   c. Sets the signature component object, WSSSignature.
   d. Sets the header in the SOAP message, specified by QName.
   e. Sets the generator security token.
   f. Adds the transform method, if needed.
3. Change from the default values for algorithm or message parts, as needed. For example: you could change one or more of the following items:
   - Add USERNAME_TOKEN as a target of decryption.
   - Change the data encryption algorithm from the default value of AES 128.
   - Change the key encryption algorithm from the default value of KW_RSA_OAEP.
   - Specify to not encrypt the encryption key (false).
   - Change the security token type from the default value of X.509 token.
   - Only use BODY_CONTENT as an encryption part and not use SIGNATURE also.

Results

The decryption information is configured for the consumer binding.

Example

The following is an example of the WSSDecryption API:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();
    X509ConsumeCallbackHandler callbackhandler = generateCallbackHandler();
// see X509ConsumeCallbackHandler
    WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
        callbackhandler);

 concont.add(dec);
```

What to do next

You must configure similar encryption information for the client-side request generator (sender) bindings, if you have not already configured the information.

Next, review the WSSDecryption API process.

Decrypting SOAP messages using the WSSDecryption API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the client for decryption on the response (client) consumer side, use the WSSDecryption API to decrypt the SOAP messages. The WSSDecryption API specifies which request SOAP message parts to decrypt when configuring the client.

Before you begin

You can use the WSS API or use policy sets on the administrative console to enable decryption and add consumer security tokens in the SOAP message. To secure SOAP messages, you must have completed the following decryption tasks:

- Encrypted the SOAP message.
- Chosen the decryption method.

About this task

The decryption information on the consumer side is used for decrypting an incoming SOAP message for the response consumer (client side) bindings. The client consumer configuration must match the configuration for the provider generator.

Confidentiality settings require that confidentiality constraints be applied to generated messages.

The following decryption parts can be configured:

Table 108. Decryption parts. Use the decryption parts to secure messages.

Decryption parts	Description
part	Adds the WSSDecryptPart object as a target of the decryption part.
keyword	Adds the decryption part using keywords. WebSphere Application Server supports the following keywords: • BODY_CONTENT • SIGNATURE • USERNAME_TOKEN
xpath	Adds the decryption part using an XPath expression.
verification	Adds the WSSVerification instance as a target of the decryption part.
header	Adds the SOAP header, specified by QName, as a target of the decryption part.

For decryption, certain default behaviors occur. The simplest way to use the WSS API for decryption is to use the default behavior (see the example code). WSSDecryption provides defaults for the key encryption algorithm, the data encryption algorithm, and the decryption parts such as the SOAP body content and the signature. The decryption default behaviors include:

Table 109. Decryption decisions. Several decryption part characteristics are configured by default.

Decryption decisions	Default behavior
Which parts to decrypt	The default decryption parts are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • WSSDecryption.BODY_CONTENT • WSSDecryption.SIGNATURE • WSSDecryption.USERNAME_TOKEN After you specify which message parts to decrypt, you must specify which method to use when decrypting the consumer request message. For example, if both signature and body content are applied for encryption, then the SOAP message parts that are decrypted include the same parts.
Whether to encrypt the key (isEncrypt)	The default value is to encrypt the key (`true`).

Table 109. Decryption decisions (continued). Several decryption part characteristics are configured by default.

Decryption decisions	Default behavior
Which data decryption algorithm to choose (method)	The default data decryption algorithm method is AES128. WebSphere Application Server supports these data encryption methods: • WSSDecryption.AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc • WSSDecryption.AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc • WSSDecryption.AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc • WSSDecryption.TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc
Which key decryption method to choose (algorithm)	The default key decryption algorithm method is key wrap RSA OAEP. WebSphere Application Server supports these key encryption methods: • WSSDecryption.KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128 • WSSDecryption.KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192 • WSSDecryption.KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256 • WSSDecryption.KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p • WSSDecryption.KW_RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5 • WSSDecryption.KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes
Which security token to specify	The default security token type is the X509 token. WebSphere Application Server provides the following pre-configured consumer token types: • Derived key token • X509 tokens

Procedure

1. To decrypt the SOAP message using the WSSDecryption API, first ensure that the application server is installed.
2. The WSS API process for decryption performs these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. The WSSConsumingContext must always be called in a JAX-WS client application.
   c. Creates the callback handler for the consumer side.
   d. Creates WSSDecryption with the class for the security token and the callback handler from the WSSFactory instance. The default behavior of WSSDecryption is to assume that the body content and the signature are encrypted.
   e. Adds the parts to be decrypted, if the default is not appropriate.
   f. Adds the candidates of the data encryption methods to use for decryption.
   g. Adds the candidates of the key encryption methods to use for decryption.
   h. Adds the candidates of the security token to use for decryption.
   i. Calls WSSDecryption.encryptKey(false) if the application does not want the key to be encrypted in the incoming message.
   j. Adds WSSDecryption to WSSConsumingContext.
   k. Calls WSSConsumingContext.process() with the SOAPMessageContext

Results

If there is an error condition during decryption, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code for decrypting the SOAP message body content:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();
```

```
// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext gencont = factory.newWSSConsumingContext();

// Generate the callback handler (step: c)
   X509ConsumeCallbackHandler callbackHandler = new
       X509ConsumeCallbackHandler(
                                "",
                                "enc-sender.jceks",
                                "jceks",
                                "storepass".toCharArray(),
                                "alice",
                                "keypass".toCharArray(),
 "CN=Alice, O=IBM, C=US");

// Generate the WSSDecryption instance (step: d)
   WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
                                              callbackHandler);

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part specified by the keyword (step: e)
      dec.addRequiredDecryptPart(WSSDecryption.BODY_CONTENT);

// Set the part in the SOAP Header specified by QName (step: e)
    dec.addRequiredDecryptHeader(new
       QName("http://www.w3.org/2005/08/addressing",
       "MessageID"));

// Set the part specified by WSSVerification  (step: e)
   X509ConsumeCallbackHandler verifyCallbackHandler =
      getCallbackHandler();
   WSSVerification ver = factory.newWSSVerification(X509Token.class,
                                                  verifyCallbackHandler);
      dec.addRequiredDecryptPart(ver);

// Set the part specified by XPath expression (step: e)
      StringBuffer sb = new StringBuffer();
         sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Envelope']");
         sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Body']");
         sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Ping']");
         sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Text']");
      dec.addRequiredDecryptPartByXPath(sb.toString());

// Set the part in the SOAP header to be decrypted specified by QName (step: e)
      dec.addRequiredDecryptHeader(new
          QName("http://www.w3.org/2005/08/addressing",
          "MessageID"));

// Set the candidates for the data encryption method (step: f)
// DEFAULT : WSSDecryption.AES128
      dec.addAllowedEncryptionMethod(WSSDecryption.AES128);
      dec.addAllowedEncryptionMethod(WSSDecryption.AES192);

// Set the candidates for the key encryption method (step: g)
// DEFAULT : WSSDecryption.KW_RSA_OAEP
      dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_TRIPLE_DES);

// Set the candidate security token to used for the decryption (step: h)
   X509ConsumeCallbackHandler callbackHandler2 = getCallbackHandler2();
      dec.addToken(X509Token.class, callbackHandler2);

// Set whether or not the key should be encrypted in the incoming SOAP message (step: i)
// DEFAULT: true
      dec.encryptKey(true);

// Add the WSSDecryption to the WSSConsumingContext (step: j)
   concont.add(dec);

// Validate the WS-Security header (step: k)
concont.process(msgcontext);
```

What to do next

Next, use the WSSDecryptPart API or configure the policy sets using the administrative console to add decrypted parts for the consumer message.

Choosing decryption methods for the consumer binding:

To configure the client for response decryption for the consumer binding, specify which data and transform algorithm methods to use when the client decrypts the SOAP messages.

Before you begin

Prior to completing these steps, read the XML encryption information to become familiar with encrypting and decrypting SOAP messages.

To complete decryption configuration to secure SOAP messages, you must complete the following tasks:
- Configure decryption of the SOAP message parts
- Specify the decryption methods.

You can configure the decryption methods using the WSSDecryption and WSSDecryptPart APIs. Or you can also configure policy sets using the administrative console to configure the decryption methods.

About this task

Some of the encryption-related definitions are based on the XML-Encryption specification. The following information defines some data encryption-related terms:

Data encryption method algorithm

Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. This algorithm encrypts and decrypts data in fixed size, multiple octet blocks.

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Key encryption method algorithm

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. The algorithm represents public key encryption algorithms that are specified for encrypting and decrypting keys.

By default, the RSA_OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:
- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA_OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW_AES256 and the KW_AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

To complete the decryption configuration, you must specify the algorithm uniform resource identifier (URI) and its usage type. If the URI is used for multiple usage types, then you must define the URI to each usage type. WebSphere Application Server supports the following decryption usage types:

Table 110. Decryption usage types. These decryption types are supported by WebSphere Application Server.

Usage types	Description
Data encryption	Specifies the algorithm URI that is used for both encrypting and decrypting data. Encrypts and decrypts data in fixed size, multiple octet blocks.
Key encryption	Specifies the algorithm URI that is used for encrypting and decrypting the encryption key.

To configure the decryption and decrypted part algorithms, use the WSSDecryption and WSSDecryptPart APIs, or configure policy sets using the administrative console.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints that use policy sets.

If you are using the WSS APIs, the WSSDecryption and WSSDecryptPart APIs specify which algorithm methods are used when the client decrypts the SOAP messages.

- Use the WSSDecryption API to configure the data encryption algorithm and the key encryption algorithm methods.
- Use the WSSDecryptPart API to configure a transform algorithm method.

The WSS API process completes the following high-level steps to specify which decryption and decrypted part algorithm methods to use when configuring the client for response decryption:

Procedure

1. Using the WSSDecryption API, adds the required data encryption algorithm. The data encryption algorithm is used for encrypting or decrypting parts of a SOAP message. Data decryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method.

   The default data encryption algorithm is AES 128. The data encryption name is AES128, and the URI of the data encryption algorithm, is http://www.w3.org/2001/04/xmlenc#aes128-cbc. WebSphere Application Server supports the following pre-configured data decryption algorithms:

   - AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc

     The AES 128 algorithm is the default data algorithm method.
   - AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc

     Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

     To use this AES 192-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

2. As needed, changes the WSSEncryption API method to specify another data encryption algorithm. For example, you might add the following code to change from the default AES 128 algorithm to the Triple DES algorithm:

```
dec.addAllowedKeyEncryptionMethod(WSSDecryption.TRIPLE_DES);
```

3. Using the WSSDecryption API, adds the required key encryption algorithm. The key encryption algorithm is used for encrypting the key that is used for encrypting the message parts within the SOAP message. If no key for encrypting the data is needed, then you must specify WSSDecryption.encryptKey(false).

   The key encryption algorithm that you select for the consumer side must match the key encryption method that you select for the generator side.

   The default key encryption algorithm value is key wrap RSA_OAEP. The key encryption name is KW_RSA_OAEP, and the URI of the key encryption algorithm is http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p. WebSphere Application Server supports the following pre-configured key encryption algorithms:

   - KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128
   - KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192

     To use this key wrap AES 192 algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

     Restriction: Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
   - KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256

     To use this key wrap AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

     The KW_RSA_OAEP algorithm is the default key algorithm method.

     When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this algorithm. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5. See more information at http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
   - KW_RSA_15: http://www.w3.org/2001/04/xmlenc#rsa-1_5
   - KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

   Note: For Web Services Secure Conversation, the WSSEncryption API might specify addition key-related information, such as the:
   - algorithmName
   - keyLength

4. As needed, uses the WSSDecryption API method to change to other key encryption algorithms. For example, you might add the following code to change from the default key encryption algorithm KW_RSA_OAEP to the TRIPLE_DES algorithm:

```
dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_TRIPLE_DES);
```

5. Using the WSSDecryptPart API, adds a transform algorithm, as needed. There is no default transform algorithm. However, WebSphere Application Server provides a pre-configured decrypted part, `WSSDecryptPart.TRANSFORM_ATTACHMENT_CIPHERTEXT`, that can be added.

Results

If there is an error condition, a WSSException is provided. If successful, the API calls the `WSSConsumerContext.process()` method, the WS-Security header is validated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample WSS API code for decrypting the body content as well as changing the data encryption and key encryption algorithms from the default values:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance
   WSSConsumingContext gencont = factory.newWSSConsumingContext();

// Generate the callback handler
   X509ConsumeCallbackHandler callbackHandler = new
      X509ConsumeCallbackHandler(
                               "",
                               "enc-sender.jceks",
                               "jceks",
                               "storepass".toCharArray(),
                               "alice",
                               "keypass".toCharArray(),
                               "CN=Alice, O=IBM, C=US");

// Generate WSSDecryption instance
   WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
                                                callbackHandler);

// Set the candidates for the data encryption method
// DEFAULT : WSSDecryption.AES128
   dec.addAllowedEncryptionMethod(WSSDecryption.AES128);
   dec.addAllowedEncryptionMethod(WSSDecryption.AES192);

// Set the candidates for the key encryption method
// DEFAULT : WSSDecryption.KW_RSA_OAEP
   dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_TRIPLE_DES);

// Add the WSSDecryption to WSSConsumingContext
   concont.add(dec);

// Validate the WS-Security header
concont.process(msgcontext);
```

Adding decrypted parts using the WSSDecryptPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure decrypted parts for the response consumer (client side) bindings, use the WSSDecryptPart API to define and add to the listing of elements in the decrypted part. WSSDecryptPart is an interface that is part of the com.ibm.websphere.wssecurity.wssapi.decryption package.

Before you begin

You can use either the WSS APIs or configure the policy sets using the administrative console to configure and add new encrypted parts. To secure SOAP messages using the WSSDecryptPart APIs, you must configure the decrypted parts for the response consumer bindings.

About this task

Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted and decrypted, and which message parts to attach encrypted elements to.

The WSSDecryptPart API specifies information related to decryption and sets the decrypted parts that have been added for message confidentiality protection. Use the WSSDecryptPart to set the transform method and to specify the part to which the transform method is to be applied. Sets the transform method only if using SOAP with Attachments. The WSSDecryptPart is usually not needed except, in some case for tasks such as setting the transform method.

The decrypted parts displayed in the following table are used to protect the confidentiality of messages.

Table 111. Decrypted Parts. Use the decrypted parts to secure messages.

Decrypted parts	Description
keyword	Sets the decrypted part using keywords. The default decrypted parts that you can add using keywords are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports the following keywords: • BODY_CONTENT • SIGNATURE • USERNAME_TOKEN
xpath	Sets the decrypted part by using an XPath expression.
verification	Sets the WSSVerification component as a decrypted part. The WSSVerification part is applicable only if the SOAP message contains a signature element.
header	Sets the header, specified by QName, as a decrypted part.

For decrypted parts, certain default behaviors occur. The simplest way to use the WSSDecryptPart API is to use the default behavior (see the example code).

WSSDecryptPart provides defaults for setting the transform algorithm, adding a transform method, setting objects as targets, whether an element, and the encrypted parts, such as: the SOAP body content and the signature.

Table 112. Decrypted part decisions. Several characteristics of decrypted parts are configured by default.

Decryption decisions	Default behavior
Which SOAP message parts to decrypt using keywords	Specifies which keywords to use for the decrypted parts. WebSphere Application Server sets the following SOAP message parts by default for decryption: • WSSDecryption.BODY_CONTENT • WSSDecryption.SIGNATURE
Which transform algorithm to use (algorithm)	WebSphere Application Server does not specify any transform algorithm by default. Specify a transform method only if using SOAP with Attachments.

Procedure

1. To decrypt the SOAP message parts using the WSSDecryptPart API, first ensure that the application server is installed.
2. The WSS API process using WSSDecryptPart follows these steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. Note that the WSSConsumingContext must always be called in a JAX-WS client application.
   c. Creates the SecurityToken from WSSFactory to configure decryption.
   d. Creates WSSDecryption from the WSSFactory instance using SecurityToken.
   e. Creates WSSDecryptPart from the WSSFactory instance. The default behavior of WSSDecryptPart is to assume that the body content and signature are encrypted.
   f. Adds the parts to be decrypted and to be applied with the transform in WSSDecryptPart. WebSphere Application Server sets these encrypted parts by default for WSSDecryptPart: the BODY_CONTENT and SIGNATURE. After you add other decrypted parts, the default values are no longer valid. For example, if you call addDecryptPart(securityToken, false), only the security token is encrypted, and not the signature and body content. So if you want to decrypt the security token, the signature, and the body content, you must call addDecryptPart(securityToken, false), addDecryptPart(WSSDecryption.SIGNATURE), and addDecryptPart(WSSDecryption.BODY_CONTENT).
   g. Sets the transform method.
   h. Adds WSSDecryptPart to WSSDecryption.
   i. Adds WSSDecryption to WSSConsumingContext.
   j. Calls WSSConsumingContext.process() with the SOAPMessageContext

Results

If there is an error condition when decrypting the message, a WSSException is provided. If successful, the API calls the WSSConsumingContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

What to do next

After enabling decrypted parts for the response consumer (client side) binding, specify the generator and consumer tokens, if the security tokens have not already been specified.

Decryption methods:

The decryption algorithms specify the data and key encryption algorithms that are used to decrypt the SOAP message. The WSS API for decryption (WSSDecryption) specifies the algorithm uniform resource identifier (URI) of the data and key encryption methods. The WSSDecryption interface is part of the `com.ibm.websphere.wssecurity.wssapi.decryption` package.

Data encryption algorithms

The data encryption algorithms are the algorithms that are used to encrypt and decrypt data. This algorithm type is used for encrypting data to encrypt and decrypt various parts of the message, including the body content and the signature.

Data decryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. WebSphere Application Server supports the following pre-configured data decryption algorithms:

Table 113. Supported pre-configured data decryption algorithms. The algorithms are used to decrypt SOAP messages.

WSS API	URI
WSSDecryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSDecryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSDecryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSDecryption.TRIPLE_DES	A URI of data encryption algorithm, TRIPLE DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The data encryption algorithm must match the data decryption algorithm that is configured for the consumer.

Key encryption algorithms

The key encryption algorithms are the algorithms that are used to encrypt and decrypt keys.

This key information is used to specify the configuration that is needed to generate the key for digital signature and encryption. The signing information and encryption information configurations can share the key information. The key information on the consumer side is used for specifying the information about the key that is used for validating the digital signature in the received message or for decrypting the encrypted parts of the message. The request generator is configured for the client.

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. WebSphere Application Server supports the following pre-configured key encryption algorithms:

Table 114. Supported pre-configured key encryption algorithms. The algorithms are used to decrypt SOAP messages.

WSS API	URI
WSSDecryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128
WSSDecryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSDecryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256
WSSDecryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSDecryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSDecryption.KW_TRIPLE_DES	A URI of data encryption algorithm, key wrap TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method: `http://www.w3.org/2001/04/xmlenc#sha256` `http://www.w3.org/2001/04/xmlenc#sha512`

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the kw-aes256 and the kw-aes192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The key encryption algorithm for the generator and the consumer must match.

The following example provides a sample of the WSS API code for the default algorithms that are used for WebSphere Application Server decryption:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();

// Required to attach username token into the message.
X509ConsumeCallbackHandler callbackHandler =
       new X509ConsumeCallbackHandler("",
                                    "enc-sender.jceks",
```

```
                                   "JCEKS",
                                   "storepass".toCharArray(),
                                   "alice",
                                   "keypass".toCharArray(),
                                   "CN=Alice, O=IBM, C=US");
// Set the decrypt component.
// Default encrypted part: Body-Content
// Default data encryption algorithm: AES128
// Default key encryption algorithm: KW-RSA-OAEP
WSSDecryption dec = factory.newWSSDecryption(X509Token.Type,
callbackHandler);
concont.add(dec);

// validate the WS-Security header.
concont.process(msgctx);
```

Verifying consumer signing information to protect message integrity using WSS APIs:

You can verify the signing information to protect message integrity for the response (client side) consumer binding. Signing information includes the signature and the signed parts for the generator side as well as signature verification and verify parts for the consumer side. To keep the integrity of the message, digital signatures are typically applied.

Before you begin

Ensure that the signature and signed parts information has been configured. The signature verification information must match what was configured on the generator side.

About this task

Integrity refers to digital signature while confidentiality refers to encryption. Integrity is provided by applying a digital signature to a SOAP message. To configure the signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity decreases the risk of data modification when you transmit data across a network.

Also, message integrity is provided by verifying the digitally signed body, time stamp, and WS-Addressing headers using the signature verification algorithm methods. The WSS APIs specify which algorithm is to be used to verify the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature verification method. WebSphere Application Server supports several pre-configured verification algorithm methods.

You can use the following interfaces to configure Web Services Security and to protect SOAP message integrity:
- Use the administrative console to configure policy sets for signature verification.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

Perform the following verification tasks, using the WSS APIs, to configure the signing information and to protect message integrity for the consumer binding.

Procedure

- Configure the signing information using the WSSSignature API. Configure the signature verification information for the consumer binding using the WSSVerification API. Signature verification information is used to verify parts of a message including the SOAP body, the time stamp, and the WS-Addressing headers. Both verifying and decryption can be applied to the same message parts, such as the SOAP body.
- Add or change verify parts using the WSSVerifyPart API.

- Configure the client for request signing methods using the WSSVerification or WSSVerifyPart APIs. To configure the client for response verification, choose the verification methods. Use the WSSVerification API to configure the canonicalization and signature methods. Use the WSSVerifyPart API to configure the digest and transform methods.

Results

By completing the steps in these tasks, you have configured the consumer verification information to protect the integrity of messages.

Verifying signing information for the consumer binding using the WSS APIs:

You can configure the signing information for the client-side response consumer (receiver) bindings. Signing information is used to sign and validate parts of a message including the SOAP body, the timestamp information, and the Username token.

Before you begin

WebSphere Application Server uses XML digital signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method. Prior to completing these steps, read the information about XML digital signature to become familiar with signing and verifying digital signatures for digital content.

By including XML signature in SOAP messages, the following issues are realized: message integrity and authentication. *Integrity* refers to digital signature whereas confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet.

Before you can verify the signature and SOAP message signed parts, you must have completed the following tasks:
- Configured the signature.
- Added signed parts, as needed.
- Chosen the signature and signed parts methods.

About this task

Use the Web Services Security APIs (WSS API) to configure the signing verification information for the response consumer (client side) section of the bindings file. Use the WSSVerification or WSSVerifyPart APIs to configure the client for request signature verification and to specify which digitally signed message parts to verify.

WebSphere Application Server uses the signing information on the consumer side to verify the integrity of the received SOAP message by validating that the message parts (such as the body, time stamp, and Username token) are signed.

On the client side, use the WSS APIs, or configure policy sets using the administrative console to specify which parts of the message are signed and to configure the key information that is referenced by the key information references. To verify the signature and signed parts, use the WSSVerification and WSSVerifyPart APIs.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

The WSSVerification and WSSVerifyPart APIs complete the following steps to specify which digitally signed message parts to verify when configuring the client for response consumer signing:

Procedure

1. The WSSVerification API adds the required verify parts of the SOAP message.

   The part reference refers to the message part that is digitally signed. The part attribute refers to the name of the <Integrity> element when the <PartReference> element is specified for the signature. You can specify multiple <PartReference> elements within the <SigningInfo> element. The <PartReference> element has two child elements when it is specified for the signature: <DigestTransform> and <Transform>.

   The WSSVerification API configures the following parts as verification parts:

Verification part	Description
Security token	Adds information for the security token that is used for the signature verification.
SOAP header and the QName as a target	Adds the SOAP header, specified by QName, as a verification part.

   The WSS APIs allow the use of keywords or an XPath expression to specify which parts of the message are to be verified. WebSphere Application Server supports the use of the following keywords:

Keyword	References
WSSVerification.ADDRESSING_HEADERS	The Web Services Addressing (WS-Addressing) headers.
WSSVerification.BODY	The SOAP message body. The body is the user data portion of the message.
WSSVerification.TIMESTAMP	The creation and expiration timestamp information.

2. The WSSVerification API adds the required header to the SOAP message. The header, specified by QName, is a required verification header.
3. The WSSVerification API adds a security token. Adds information about the security token that is to be used for the signature verification, such as:
   - The class for security token.
   - The callback handler
   - The name of the JAAS login configuration.
4. The WSSVerification API adds the signature method algorithm. The signature method is the algorithm that is used to convert the canonicalized <SignedInfo> element in the binding file into the <SignatureValue> element. The algorithm that is specified for the consumer, which is the response consumer configuration, must match the algorithm specified for the request generator configuration. WebSphere Application Server supports the following pre-configured signature algorithms:
   - WSSVerification.RSA_SHA1:http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSVerification.HMAC_SHA1:http://www.w3.org/2000/09/xmldsig#hmac-sha1

   WebSphere Application Server does not support the following algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1. You cannot use the DSA-SHA1 algorithm if you want to be compliant with the Basic Security Profile (BSP).
5. The WSSVerification API adds a canonicalization method. The canonicalization method algorithm is used to canonicalize the <SignedInfo> element before it is incorporated as part of the digital signature operation. The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer.

   WebSphere Application Server supports the following pre-configured canonicalization algorithms:
   - WSSVerification.EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerification.C14N: http://www.w3.org/TR/xml-c14n
6. The WSSVerification API verifies whether a signature confirmation is required. The OASIS Web Services Security (WS-Security) Version 1.1 specification defines the use of signature confirmation. If you are using WS-Security Version 1.0, this function is not available.

The signature confirmation value is stored in order to validate the signature confirmation with it after the receiving message is returned. This method is called if the response message is expected to attach the signature confirmation into the SOAP message.

7. The WSSVerifyPart API adds a digest method. For each part reference in the signing information, the API specifies both a digest method algorithm and a transform algorithm.

   WebSphere Application Server supports the following pre-configured digest algorithms:
   - WSSVerifyPart.SHA1: http://www.w3.org/2000/09/xmldsig#sha1
   - WSSVerifyPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSVerifyPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
8. The WSSVerifyPart API adds a transform method. For each part reference in the signing information, the API specifies both a digest method algorithm and a transform algorithm.

   WebSphere Application Server supports the following pre-configured transform algorithms:
   - WSSVerifyPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerifyPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSVerifyPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support these algorithms:
   - http://www.w3.org/2002/07/decrypt#XML
   - http://www.w3.org/TR/1999/REC-xpath-19991116

   The transform algorithm for the consumer must match the transform algorithm for the generator.

Results

You have completed the steps to configure the signing information for the client-side response consumer sections of the bindings files.

Example

The following example shows WSS API sample code to verify the signature and to verify the X.509 token type as the security token:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();
// Generate the X.509 Callback Handler on the consumer side
    X509ConsumeCallbackHandler callbackhandler = generateCallbackHandler();
    WSSVerification ver = factory.newWSSVerification(X509Token.class,
        callbackhandler);
concont.add(ver);
```

What to do next

If not already configured, specify a similar signing information configuration for the generator bindings.

Next, if already configured, configure the encryption and decryption information, or configure the consumer and generator tokens.

Verifying the signature using the WSSVerification API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To verify the signing information for the consumer binding sections for the client side request, use the WSSVerification API. You must also specify which algorithm methods and which signature parts of the SOAP message are to be verified. The WSSVerification API is part of the com.ibm.websphere.wssecurity.wssapi.verification package.

Before you begin

Use the WSS APIs, or configure the policy sets by using the administrative console to verify the signing information. To secure SOAP messages, you must complete the following signature tasks:

- Configure the signature information.
- Choose the algorithm methods for signature and signature verification.
- Verify the signature information.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing algorithms such as RSA-SHA1 and HMAC-SHA1.

XML signature defines many methods for describing key information and enables the definition of a new method. XML canonicalization (C14N) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. The C14N process is used to canonicalize XML information. Select an appropriate C14N algorithm because the information that is canonicalized depends on this algorithm.

The following table shows the required and optional binding information when the digital signature security constraint (integrity) is defined.

Table 115. Signature verification parts. Use the signature verification parts to secure messages.

Verification parts	Description
keywords	Adds required signature parts as targets of verification by using keywords . Different message parts can be specified in the message protection for request on the generator side. Use the following keywords for the required signature verification parts: • ADDRESSING_HEADERS • BODY • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds verification parts by using an XPath expression.
part	Adds the WSSVerifyPart object as a verification part.
header	Adds the header, specified by QName, as a verification part.

For signature verification information, certain default behaviors occur. The simplest way to use the WSSVerification API is to use the default behavior.

The default values are defined by the WSS API for the digest method, the transform method, the security token, and the required verification parts.

Table 116. Signature verification default behaviors. Several characteristics of the signature verification parts are configured by default.

Signature verification decisions	Default behavior
Which signature method to use (algorithm)	Sets the signature algorithm method. Both the data encryption and the signature and the canonicalization can be specified. The default signature method is RSA SHA1. WebSphere Application Server supports the following pre-configured signature methods: • WSSVerification.RSA_SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1 • WSSVerification.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1 The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported.
Which canonicalization method to use (algorithm)	Sets the canonicalization algorithm method. Both the data encryption and the signature and the canonicalization can be specified. The default signature method is EXC_C14N. WebSphere Application Server supports the following pre-configured canonicalization methods: • WSSVerification.EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n# • WSSVerification.C14N: http://www.w3.org/2001/10/xml-c14n#

Table 116. Signature verification default behaviors (continued). Several characteristics of the signature verification parts are configured by default.

Signature verification decisions	Default behavior
Whether signature confirmation is required	If the WSSSignature API specifies that signature confirmation is required, then the WSSVerification API verifies the signature confirmation value in the response message that has the signature confirmation value attached to it when received. Signature confirmation is defined in the OASIS Web Services Security Version 1.1 specification. The default signature confirmation is false.
Which security token to specify (securityToken)	Adds the securityToken object as a signature part. WebSphere Application Server sets the token information to use for verification. WebSphere Application Server supports the following pre-configured tokens for signing: • X.509 Token • Derived Key Token Information required for tokens include the class for the token, the callback handler information, and the name of the JAAS login module.

Procedure

1. To verify the signature in a SOAP message by using the WSSVerification API, first ensure that the application server is installed.
2. Use the WSSVerification API to set the message parts to be verified and to specify the algorithms in a SOAP message. The WSS API process for signature verification follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance.
   c. Ensures that WSSConsumingContext is called in the JAX-WS Provider implementation class. Due to the nature of the JAX-WS programming model, a JAX-WS provider needs to be implemented and must call the WSSConsumingContext to verify the SOAP message signature.
   d. Creates WSSVerification from the WSSFactory instance.
   e. Adds the part to be verified. If the digest method or the transform method are changed, create WSSVerifyPart and set it into WSSVerification.
   f. Sets the candidates of the canonicalization method, if the default is not appropriate.
   g. Sets the candidates of the signature method, if the default is not appropriate.
   h. Sets the candidate security token, if the default is not appropriate.
   i. Calls the requireSignatureConfirmation(), if the signature confirmation is applied.
   j. Adds WSSVerification to WSSConsumingContext.
   k. Calls WSSConsumingContext.process() with the SOAP message context.

Results

You have completed the steps to verify the signature for the consumer section of the bindings. If there is an error condition, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses methods that are defined in the WSSVerification API:

```
// Get the message context
  Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext concont = factory.newWSSConsumingContext();
```

```
// Generate the certificate list
  String certpath = "c:/WebSphere/AppServer/etc/ws-security/samples/intca2.cer";
// The location of the X509 certificate file
  X509Certificate x509cert = null;
     try {
      InputStream is = new FileInputStream(certpath);
      CertificateFactory cf = CertificateFactory.getInstance("X.509");
      x509cert = (X509Certificate)cf.generateCertificate(is);
       } catch(FileNotFoundException e1){
        throw new WSSException(e1);
       } catch (CertificateException e2) {
        throw new WSSException(e2);
       }
      Set<Object> eeCerts = new HashSet<Object>();
       eeCerts.add(x509cert);
// Create the certificate store
   java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
   CollectionCertStoreParameters certparam = new CollectionCertStoreParameters(eeCerts);
   CertStore cert = null;
   try {
    cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
   } catch (NoSuchProviderException e1) {
    throw new WSSException(e1);
   } catch (InvalidAlgorithmParameterException e2) {
    throw new WSSException(e2);
   } catch (NoSuchAlgorithmException e3) {
    throw new WSSException (e3);
   }
   if(certList != null ){
    certList.add(cert);
   }
// Generate the callback handler
   X509ConsumeCallbackHandler callbackHandler = new X509ConsumeCallbackHandler(
     "dsig-receiver.ks",
     "jks",
     "server".toCharArray(),
     certList,
     java.security.Security.getProvider("IBMCertPath")
   );

// Generate the WSSVerification instance (step: d)
  WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandler);

// Set the part to be verified (step: e)
// DEFAULT: WSSVerification.BODY, WSSSignature.ADDRESSING_HEADERS,
// and WSSSignature.TIMESTAMP.

// Set the part in the SOAP header to be specified by QName (step: e)
     ver.addRequiredVerifyHeader(new QName("http://www.w3.org/2005/08/addressing", "MessageID"));

// Set the part to be specified by the keyword (step: e)
     ver.addRequiredVerifyPart(WSSVerification.BODY);

// Set the part to be specified by WSSVerifyPart (step: e)
  WSSVerifyPart verPart = factory.newWSSVerifyPart();
     verPart.setRequiredVerifyPart(WSSVerification.BODY);
     verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);
     ver.addRequiredVerifyPart(verPart);

// Set the part specified by XPath expression (step: e)
  StringBuffer sb = new StringBuffer();
        sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Envelope']");
        sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Body']");
        sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Ping']");
        sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Text']");
     ver.addRequiredVerifyPartByXPath(sb.toString());


// Set one or more canonicalization method candidates for verification (step: f)
// DEFAULT : WSSVerification.EXC_C14N
    ver.addAllowedCanonicalizationMethod(WSSVerification.C14N);
    ver.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

// Set one or more signature method candidates for verification (step: g)
// DEFAULT : WSSVerification.RSA_SHA1
    ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);

// Set the candidate security token to used for the verification (step: h)
  X509ConsumeCallbackHandler callbackHandler2 = getCallbackHandler2();
    ver.addToken(X509Token.class, callbackHandler2);


// Set the flag to require the signature confirmation (step: i)
   ver.requireSignatureConfirmation();
```

```
// Add the WSSVerification to the WSSConsumingContext (step: j)
  concont.add(ver);

//Validate the WS-Security header (step: k)
concont.process(msgcontext);
```

What to do next

After verifying the signature and setting algorithm methods for the SOAP message, you can set either the digest method or the transform method. If you want to set these methods, use the WSSVerifyPart API, or configure policy sets using the administrative console.

Verifying signed parts using the WSSVerifyPart API:

To secure SOAP messages on the consumer side, use the Web Services Security APIs (WSS API) to configure the verify parts information for the consumer binding on the response consumer (client side). You can specify which algorithm methods and which parts of the SOAP message are to be verified. Use the WSSVerifyPart API to change the digest method or the transform method. The WSSVerifyPart API is part of the com.ibm.websphere.wssecurity.wssapi.verification package.

Before you begin

To secure SOAP messages using the signing verification information, you must complete one of the following tasks:

- Configure the signature verification information using the WSSVerification API.
- Configure verify parts using the WSSVerifyPart API, as needed.

The WSSVerifyPart is used for specify the transform or digest methods for the verification. Use the WSSVerifyPart API or configure policy sets using the administrative console.

About this task

WebSphere Application Server uses the signing information for the default consumer to verify the signed parts of the message. The WSSVerifyPart API is only supported on the response consumer (requester).

The following table shows the required verification parts when the digital signature security constraint (integrity) is defined:

Table 117. Verify parts information. Use the verify parts to secure messages with signing verification information.

Verify parts information	Description
keyword	Sets the verify parts using the following keywords: • BODY • ADDRESSING_HEADERS • TIMESTAMP The WS-Addressing headers are not decrypted but can be signed and verified.
xpath	Sets the verify parts using an XPath expression.
header	Sets the header, specified by QName, as a required verify part.

For signature verification, certain default behaviors occur. The simplest way to use the WSSVerification API is to use the default behavior (see the example code). The default values are defined by the WSS API for the signing algorithm and the canonicalization algorithm, and the verify parts.

Table 118. Verify parts default behaviors. Several characteristics of verify parts are configured by default.

Verify parts decisions	Default behavior
Which keywords to specify	The different SOAP message parts to be signed and used for message protection. WebSphere Application Server supports the following keywords: • WSSVerification.BODY • WSSVerification.ADDRESSING_HEADERS • WSSVerification.TIMESTAMP
Which transform method to use (algorithm)	Adds the transform method. The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signature. The default transform method is TRANSFORM_EXC_C14N. WebSphere Application Server supports the following pre-configured transform algorithms: • WSSVerifyPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n# • WSSVerifyPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2 Use this transform method to ensure compliance with the Basic Security Profile (BSP). • WSSVerifyPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform • WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature
Which digest method to use (algorithm)	Sets the digest algorithm method. The digest method algorithm that is specified within the <DigestMethod> element is used in the <SigningInfo> element. The default digest method is SHA1. WebSphere Application Server supports the following digest method algorithms: • WSSVerifyPart.SHA1: http://www.w3.org/2000/09/xmldsig#sha1 • WSSVerifyPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256 • WSSVerifyPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512

Procedure

1. To verify signed parts by using the WSSVerifyPart API, first ensure that the application server is installed.
2. Use the Web Services Security API to verify the verification in a SOAP message. The WSS API process for verifying the signature follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. Ensures that WSSConsumingContext is called in the JAX-WS Provider implementation class. Due to the nature of the JAX-WS programming model, a JAX-WS provider needs to be implemented and must call the WSSConsumingContext to verify the SOAP message signature.
   c. Creates the CallbackHandler to use for verification.
   d. Create the WSSVerification object from the WSSFactory instance.
   e. Creates WSSVerifyPart from the WSSFactory instance.
   f. Sets the part to be verified, if the default is not appropriate.
   g. Sets the candidates for the digest method, if the default is not appropriate.
   h. Sets the candidates for the transform method, if the default is not appropriate.
   i. Adds WSSVerifyPart to WSSVerification.
   j. Adds WSSVerification to WSSConsumingContext.
   k. Calls WSSConsumingContext.process() with the SOAPMessageContext.

Results

You have completed the steps to verify to verify the signed parts on the consumer side. If there is an error condition when verifying the signing information, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and Web Services Security is verified for the SOAP message.

Example

The following example provides sample code for the WSSVerification API process for verifying the signing information in a SOAP message:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext concont = factory.newWSSConsumingContext();

// Generate the certificate list
   String certpath =
   "c:/WebSphere/AppServer/etc/ws-security/samples/intca2.cer";
// The location of the X509 certificate file
   X509Certificate x509cert = null;
      try {
          InputStream is = new FileInputStream(certpath);
          CertificateFactory cf = CertificateFactory.getInstance("X.509");
          x509cert = (X509Certificate)cf.generateCertificate(is);
      } catch(FileNotFoundException e1){
            throw new WSSException(e1);
      } catch (CertificateException e2) {
            throw new WSSException(e2);
      }

      Set<Object> eeCerts = new HashSet<Object>();
      eeCerts.add(x509cert);
// create certStore
      java.util.List<CertStore> certList = new
          java.util.ArrayList<CertStore>();
      CollectionCertStoreParameters certparam = new
          CollectionCertStoreParameters(eeCerts);
      CertStore cert = null;
      try {
         cert = CertStore.getInstance("Collection",
              certparam, "IBMCertPath");
      } catch (NoSuchProviderException e1) {
           throw new WSSException(e1);
      } catch (InvalidAlgorithmParameterException e2) {
           throw new WSSException(e2);
      } catch (NoSuchAlgorithmException e3) {
           throw new WSSException (e3);
      }
      if(certList != null ){
      certList.add(cert);
      }

// generate callback handler (step: c)
   X509ConsumeCallbackHandler callbackHandler = new
       X509ConsumeCallbackHandler(
                                   "dsig-receiver.ks",
                                   "jks",
                                   "server".toCharArray(),
                                   certList,
       java.security.Security.getProvider("IBMCertPath")
);

// Generate the WSSVerification instance (step: d)
   WSSVerification ver = factory.newWSSVerification(X509Token.class,
       callbackHandler);

// Set the part to be specified by WSSVerifyPart (step: e)
   WSSVerifyPart verPart = factory.newWSSVerifyPart();

// Set the part to be specified by the keyword (step: f)
   verPart.setRequiredVerifyPart(WSSVerification.BODY);

// Set the candidates for the digest method for verification (step: g)
// DEFAULT : WSSVerifyPart.SHA1
   verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);

// Set the candidates for the transform method for verification (step: h)
// DEFAULT : WSSVerifypart.TRANSFORM_EXC_C14N : String
   verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_STRT10);

// Set WSSVerifyPart to WSSVerification (step: i)
   ver.addRequiredVerifyPart(verPart);

// Add WSSVerification to WSSConsumingContext (step: j)
   concont.add(ver);

//Validate the WS-Security header (step: k)
concont.process(msgcontext);
```

What to do next

You have completed configuring the signed part to be verified.

Configuring response signature verification methods for the client:

Use the WSSVerification and WSSVerifyPart APIs to choose the signing verification methods. The request signing verification methods include the digest algorithm and the transport methods.

Before you begin

To complete configuration of the signature verification information to secure SOAP messages, you must perform the following algorithm tasks:

- Use the WSSVerification API to configure the canonicalization and signature methods.
- Use the WSSVerifyPart API to configure the digest and transform methods.

to configure the algorithm methods to use when configuring the client for request signing.

About this task

The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following website http://www.w3.org/TR/xmldsig-core.

Table 119. Signing verification methods. Use signing verification information to secure messages.

Name of method	Purpose
Digest algorithm	Applies to the data after transforms are applied, if specified, to yield the `<DigestValue>` element. Signing the `<DigestValue>` element binds the resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the client request receiver configuration.
Transform algorithm	Applies to the `<Transform>` element.
Signature algorithm	Specifies the Uniform Resource Identifiers (URI) of the signature verification method.
Canonicalization algorithm	Specifies the Uniform Resource Identifiers (URI) of the canonicalization method.

After configuring the client to digitally sign the message, you must configure the client to verify the digital signature. You can use the WSS APIs or configure policy sets using the administrative console to verify the digital signature and to choose the verification and verify part algorithms. If using the WSS APIs to configure, use the WSSVerification and WSSVerifyPart APIs to specify which digitally signed message parts to verify and to specify which algorithm methods to use when configuring the client for request signing.

The WSSVerification and WSSVerifyPart APIs perform the following steps to configure the signature verification and verify parts algorithm methods:

Procedure

1. For the consumer binding, the WSSVerification API specifies the signature methods to allow for the signature verification. WebSphere Application Server supports the following pre-configured signature methods:
   - WSSVerification.RSA_SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSVerification.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1

   The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported.

2. For the consumer binding, the WSSVerification API specifies the canonicalization method to allow for the signature verification. WebSphere Application Server supports the following pre-configured canonicalization methods by default:
   - WSSVerification.EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerification.C14N: http://www.w3.org/2001/10/xml-c14n#
3. For the consumer binding, the WSSVerifyPart API specifies the digest method, as needed. WebSphere Application Server supports the following digest method algorithms for signed parts verification:
   - WSSVerifyPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1
   - WSSVerifyPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSVerifyPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
4. For the consumer binding, the WSSVerifyPart API specifies the transform method. WebSphere Application Server supports the following transform algorithms for verify parts:
   - WSSVerifyPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerifyPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSVerifyPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support these algorithms:
   - http://www.w3.org/2002/07/decrypt#XML
   - http://www.w3.org/TR/1999/REC-xpath-19991116

Results

You have specified which method to use when verifying a digital signature when the client sends a message.

Example

The following example provides sample WSS API code that specifies the verification information, the body as a part to be verified, the HMAC_SHA1 as a signature method, C14N and EXC_C14N as the candidates of canonicalization methods, TRANSFORM_STRT10 as a transform method, and SHA256 as a digest method.

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance
   WSSConsumingContext concont = factory.newWSSConsumingContext();

// Generate the certificate list
   String certpath = "intca2.cer";
// The location of the X509 certificate file
      X509Certificate x509cert = null;
      try {
            InputStream is = new FileInputStream(certpath);
            CertificateFactory cf = CertificateFactory.getInstance("X.509");
           x509cert = (X509Certificate)cf.generateCertificate(is);
      } catch(FileNotFoundException e1){
          throw new WSSException(e1);
      } catch (CertificateException e2) {
          throw new WSSException(e2);
      }

      Set<Object> eeCerts = new HashSet<Object>();
      eeCerts.add(x509cert);
// Create the certStore
   java.util.List<CertStore> certList = new
        java.util.ArrayList<CertStore>();
   CollectionCertStoreParameters certparam = new
```

```
        CollectionCertStoreParameters(eeCerts);
   CertStore cert = null;
      try {
            cert = CertStore.getInstance("Collection",
                                         certparam,
                                        "IBMCertPath");
      } catch (NoSuchProviderException e1) {
           throw new WSSException(e1);
      } catch (InvalidAlgorithmParameterException e2) {
           throw new WSSException(e2);
      } catch (NoSuchAlgorithmException e3) {
           throw new WSSException (e3);
      }
      if(certList != null ){
            certList.add(cert);
      }

// Generate the callback handler
   X509ConsumeCallbackHandler callbackHandler = new
      X509ConsumeCallbackHandler(
                               "dsig-receiver.ks",
                               "jks",
                               "server".toCharArray(),
                               certList,
                               java.security.Security.getProvider(
                                     "IBMCertPath")
                               );

// Generate the WSSVerification instance
   WSSVerification ver = factory.newWSSVerification(X509Token.class,
                                                    callbackHandler);

// Set one or more candidates of the signature method used for
//    verification (step. 1)
// DEFAULT : WSSVerification.RSA_SHA1
      ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);

// Set one or more candidates of the canonicalization method used for
//    verification (step. 2)
// DEFAULT : WSSVerification.EXC_C14N
      ver.addAllowedCanonicalizationMethod(WSSVerification.C14N);
      ver.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

// Set the part to be specified by WSSVerifyPart
   WSSVerifyPart verPart = factory.newWSSVerifyPart();

// Set the part to be specified by the keyword
      verPart.setRequiredVerifyPart(WSSVerification.BODY);

// Set the candidates of digest methods to use for verification (step. 3)
// DEFAULT : WSSVerifypart.TRANSFORM_EXC_C14N : String
      verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_STRT10);

// Set the candidates of digest methods to use for verification (step. 4)
// DEFAULT : WSSVerifyPart.SHA1
      verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);

// Set WSSVerifyPart to WSSVerification
      ver.addRequiredVerifyPart(verPart);

// Add the WSSVerification to the WSSConsumingContext
   concont.add(ver);

// Validate the WS-Security header
concont.process(msgcontext);
```

What to do next

You have completed configuring the signature verification algorithms. Next, configure the encryption or decryption algorithms, if not already configured. Or, configure the security token information, as needed.

Signature verification methods using the WSSVerification API:

You can verify the signing or signature information using the WSS API for the consumer binding. The signature and canonicalization algorithm methods are used for the generator binding. The WSSVerification API is provided in the com.ibm.websphere.wssecurity.wssapi.verification package.

To configure consumer signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

Methods

Methods that are used for the signature verification include the:

Signature method
Sets the signature algorithm method.

Canonicalization method
Sets the canonicalization algorithm method.

The algorithm that is specified for the request generator configuration must match the algorithm that is specified for the response consumer configuration.

Signature algorithms

The signature algorithms specify the signature verification algorithm that is used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature verification method. WebSphere Application Server supports the following pre-configured algorithms:

Table 120. Signature verification algorithms. The algorithms include the signature methods.

Algorithm	Description
WSSVerification.HMAC_SHA1	A URI of the signature algorithm, HMAC: http://www.w3.org/2000/09/xmldsig#hmac-sha1
WSSVerification.RSA_SHA1 (the default value)	A URI of the signature algorithm, RSA: http://www.w3.org/2000/09/xmldsig#rsa-sha1

WebSphere Application Server does not support the algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1

Canonicalization algorithms

The canonicalization algorithms specify the Uniform Resource Identifiers (URI) of the canonicalization method. WebSphere Application Server supports the following pre-configured algorithms:

Table 121. Verification canonicalization algorithms. The algorithms include the canonicalization methods.

Algorithm	Description
WSSVerification.C14N	A URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#
WSSVerification.EXC_C14N (the default value)	A URI of the exclusive canonicalization algorithm EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#

The following example provides sample WSS API code that specifies the X.509 token security token for signature verification:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();

// X509ConsumeCallbackHandler
X509ConsumeCallbackHandler callbackHandler = new
   X509ConsumeCallbackHandler("dsig-receiver.ks",
                              "jks",
                              "server".toCharArray(),
                              certList,
   java.security.Security.getProvider("IBMCertPath")46 );

// Set the verification component
// DEFAULT verification parts: Body, WS-Addressing header, and Timestamp
// DEFAULT data encryption algorithm: RSA-SHA1
// DEFAULT digest algorithm: SHA1
// DEFAULT canonicalization algorithm: exc-c14n
WSSVerification ver = factory.newWSSVerification(X509Token.class,
                                                 callbackhandler);
concont.add(ver);

// Validate the WS-Security header
concont.validate(msgctx);
```

Choosing the verify parts methods using the WSSVerifyPart API:

You can configure the signing verification information for the consumer binding using the WSS API. The transform algorithm and digest methods are used for the consumer binding. Use the WSSVerifyPart API to configure the algorithm methods. The WSSVerifyPart API is provided in the com.ibm.websphere.wssecurity.wssapi.verification package.

To configure consumer verify parts information to protect message integrity, you must first digitally sign and then verify the signature and signed parts for the SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

Methods

Methods that are used for the signing information include the:

Digest method
Sets the digest method.

Transform method
Sets the transform algorithm method.

Digest algorithms

The digest method algorithm is specified within the element is used in the <Digest> element. WebSphere Application Server supports the following pre-configured digest algorithms:

Table 122. Verify parts digest methods. Use the verify parts to protect message integrity.

Digest method	Description
WSSVerifyPart.SHA1 (the default value)	A URI of the digest algorithm, SHA1: http://www.w3.org/2000/09/xmldsig#sha1
WSSVerifyPart.SHA256	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha256
WSSVerifyPart.SHA512	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha512

Transform algorithms

The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signed part. WebSphere Application Server supports the following pre-configured transform algorithms:

Table 123. Verify parts transform methods. Use the verify parts to protect message integrity.

Digest method	Description
WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE	A URI of the transform algorithm, enveloped signature: http://www.w3.org/2000/09/xmldsig#enveloped-signature
WSSVerifyPart.TRANSFORM_STRT10	A URI of the transform algorithm, STR-Transform: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
WSSVerifyPart.TRANSFORM_EXC_C14N (the default value)	A URI of the transform algorithm, Exc-C14N: http://www.w3.org/2001/10/xml-exc-c14n#
WSSVerifyPart.TRANSFORM_XPATH2_FILTER	A URI of the transform algorithm, XPath2 filter: http://www.w3.org/2002/06/xmldsig-filter2

For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:

- http://www.w3.org/TR/1999/REC-xpath-19991116
- http://www.w3.org/2002/07/decrypt#XML

The following example provides sample WSS API code that verifies the body using SHA256 as the digest method and TRANSFORM_EXC_14N and TRANSFORM_STRT10 as the transform methods:

```
// get the message context
Object msgcontext = getMessageContext();

// generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

// generate WSSConsumingContext instance
WSSConsumingContext concont = factory.newWSSConsumingContext();


// generate the cert list
String certpath = "intca2.cer";// The location of the X509
   certificate file X509Certificate x509cert = null;
try {
 InputStream is = new FileInputStream(certpath);
 CertificateFactory cf = CertificateFactory.getInstance("X.509");
 x509cert = (X509Certificate)cf.generateCertificate(is);
} catch(FileNotFoundException e1){
 throw new WSSException(e1);
} catch (CertificateException e2) {
 throw new WSSException(e2);
}

Set<Object> eeCerts = new HashSet<Object>();
eeCerts.add(x509cert);
// create certStore
java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
CollectionCertStoreParameters certparam = new
      CollectionCertStoreParameters(eeCerts);
CertStore cert = null;
try {
 cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
} catch (NoSuchProviderException e1) {
 throw new WSSException(e1);
} catch (InvalidAlgorithmParameterException e2) {
 throw new WSSException(e2);
} catch (NoSuchAlgorithmException e3) {
 throw new WSSException (e3);
}
if(certList != null ){
 certList.add(cert);
}

// generate callback handler
X509ConsumeCallbackHandler callbackHandler = new
      X509ConsumeCallbackHandler(
  "dsig-receiver.ks",
  "jks",
  "server".toCharArray(),
  certList,
  java.security.Security.getProvider("IBMCertPath")
);

//generate WSSVerification instance
WSSVerification ver = factory.newWSSVerification(X509Token.class,
     callbackHandler);

//set one or more candidates of the signature method used for the
 //verification (step. 1)
// DEFAULT : WSSVerification.RSA_SHA1
ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);

//set one or more candidates of the canonicalization method used
 //for the verification (step. 2)
// DEFAULT : WSSVerification.EXC_C14N
ver.addAllowedCanonicalizationMethod(WSSVerification.C14N);
ver.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

//set the part to be specified by WSSVerifyPart
WSSVerifyPart verPart = factory.newWSSVerifyPart();

//set the part to be specified by the keyword
verPart.setRequiredVerifyPart(WSSVerification.BODY);

//set the candidates of digest methods to use for verification (step. 3)
// DEFAULT : WSSVerifypart.TRANSFORM_EXC_C14N
verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_EXC_C14N);
verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_STRT10);

//set the candidates of digest methods to use for verification (step. 4)
// DEFAULT : WSSVerifyPart.SHA1
verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);

//set WSSVerifyPart to WSSVerification
ver.addRequiredVerifyPart(verPart);
```

```
//add the WSSVerification to the WSSConsumingContext
concont.add(ver);

//validate the WS-Security header
concont.process(msgcontext);
```

Validating the consumer token to protect message authenticity:

The token consumer information is used on the consumer side to incorporate and validate the security token. The Username token, X509 tokens, and LTPA tokens by default are used for message authenticity.

Before you begin

The token processing and pluggable token architecture in the Web Services Security run time reuses the same security token interface and Java Authentication and Authorization Service (JAAS) Login Module from the Web Services Security APIs (WSS API). The same implementation of token creation and validation can be used in both the WSS API and the WSS SPI in the Web Services Security run time.

Restriction: The `com.ibm.wsspi.wssecurity.token.TokenConsumingComponent` interface is not used with JAX-WS web services. If you are using JAX-RPC web services, this interface is still valid.

Note that the key name (KeyName) element is not supported because there is no KeyName policy assertion defined in the current OASIS Web Services Security draft specification.

About this task

The JAAS callback handler (CallbackHandler) and the JAAS login module (LoginModule) are responsible for creating the security token on the generator side and validating (authenticating) the security token on the consumer side.

For example, on the generator side, the Username token is created by the JAAS LoginModule and using the JAAS CallbackHandler to pass the authentication data. The JAAS LoginModule creates the Username SecurityToken object and passes it to the Web Services Security run time.

Then, on the consumer side, the Username Token XML format is passed to the JAAS LoginModule for validation or authentication and the JAAS CallbackHandler is used to pass authentication data from the Web Services Security run time to the LoginModule. After the token is authenticated, a Username SecurityToken object is created and passed it to the Web Services Security run time.

Note: WebSphere Application Server does not support a stackable login module with the WebSphere Application Server default login module implementation, meaning adding the login module before or after the WebSphere Application Server login module implementation. If you want to stack the login module implementations, you must develop the required login modules because there is no default implementation.

The com.ibm.websphere.wssecurity.wssapi.token package provided by WebSphere Application Server includes support for these classes:

- Security token (SecurityTokenImpl)
- Binary security token (BinarySecurityTokenImpl)

In addition, WebSphere Application Server provides the following pre-configured sub-interfaces for security tokens:

- Derived key token
- Security context token (SCT)
- Username token
- LTPA token propagation

- LTPA token
- X509PKCS7 token
- X509PKIPath token
- X509v3 token
- Kerberos v5 token

The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity. The derived key token and the X.509 tokens are used by default for signing and encryption.

The WSS API and WSS SPI are only supported on the client. To specify the security token type on the consumer side, you can also configure policy sets using the administrative console. You can also use the WSS APIs or policy sets for matching generator security tokens.

The default Login Module and Callback implementations are designed to be used as a pair, meaning both a generator and a consumer part. To use the default implementations, select the appropriate generator and consumer security token in a pair. For example, select `system.wss.generate.x509` in the token generator and `system.wss.consume.x509` in the token consumer when the X.509 token is required.

To configure the consumer-side security token, use the appropriate pre-configured token consumer interface from the WSS APIs to complete the following token configuration process steps:

Procedure

1. Generate the wssFactory instance.
2. Generate the wssConsumingContext instance.

   The WSSConsumingContext interface stores the components for consuming Web Services Security (WS-Security), such as verification, decryption, the security token, and the time stamp. When the validate() method is called, all of these components are validated.
3. Create the consumer-side components, such as the WSSVerification and the WSSDecryption objects.
4. Specify a JAAS configuration by specifying the name of the JAAS login configuration. The Java Authentication and Authorization Service (JAAS) configuration specifies the name of the JAAS configuration. The JAAS configuration specifies how the token logs in on the consumer side. Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module.
5. Specify a token consumer class name. The token consumer class name specifies the required information to validate the SecurityToken. The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity.
6. Specify the settings for the callback handler by specifying a callback handler class name and also specifies the callback handler keys. This class name is the name of the callback handler implementation class that is used for the plug-in to the security token framework.

   WebSphere Application Server provides the following default callback handler implementations for the consumer side:

   com.ibm.websphere.wssecurity.callbackhandler.PropertyCallback
   This class is a callback for handling the name-value pair in elements in the Web Services Security (WS-Security) configuration XMI files.

   ccom.ibm.websphere.wssecurity.callbackhandler.UNTConsumeCallbackHandler
   This class is a callback handler for the Username token on the consumer side. This instance is used to set into WSSConsumingContext object to validate a Username token. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

   com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler
   This class is a callback handler that is used to validate the X.509 certificate that is inserted in

the Web Services Security header within the SOAP message as a binary security token on the consumer side. This instance is used to generate the WSSVerification object and WSSDecryption objects, set the objects into WSSConsumingContext object to validate the X.509 binary security tokens. A keystore and a key definition are required for this callback handler. If you use this implementation, a key store password, path, and type must have been provided on the generator side.

com.ibm.websphere.wssecurity.callbackhandler.LTPAConsumeCallbackHandler
This class is a callback handler for the Lightweight Third Party Authentication (LTPA) tokens on the consumer side. This instance is used to generate the WSSVerification and WSSDecryption objects to validate an LTPA token.

This callback handler is used to validate the LTPA security token inserted in the Web Services Security header within the SOAP message as a binary security token. However, if the user name and password are specified, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the Run As Subject. Use this callback handler only when the web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client. If you use this implementation, a basic authentication user ID and password must have been provided on the generator side.

com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler
This class is a callback handler for the Kerberos v5 token on the consumer side. This instance is used to set the WSSConsumingContext object to consume the Kerberos v5 AP-REQ as a binary security token. The instance is also used to generate the WSSVerification and WSSDecryption objects to use the Kerberos session key or derived key in the SOAP message verification and decryption.

7. If a X.509 token is specified, additional token information is also specified.

Table 124. Information for the X.509 token. Use the X.509 token to authenticate messages.

Token Information	Description
keyStoreRef	The reference name of the keystore that is used for the key locator.
keyStorePath	The keystore file path from which the keystore is loaded, if needed. It is recommended that you use the `${USER_INSTALL_ROOT}` in the path name as this variable expands to the WebSphere Application Server path on your machine. This path is required when you use the X.509 tokens callback handler implementations.
keyStorePassword	The password that is used to check the integrity of the keystore, or the keystore password that is used to unlock the keystore and to access the keystore file. The keystore and its configuration are used for some of the default callback handler implementations that are provided by WebSphere Application Server.
keyStoreType	The keystore type of keystore that is used for the key locator. This selection indicates the format that is used by the keystore file. The following values are available for selection: **JKS** Use this option if the keystore uses the Java Keystore (JKS) format. **JCEKS** Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in WebSphere Application Server. This option provides stronger protection for stored private keys by using Triple DES encryption. **JCERACFKS** Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only). **PKCS11KS (PKCS11)** Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might contain encrypt keys that use cryptographic hardware to ensure protection. **PKCS12KS (PKCS12)** Use this option if your keystore uses the PKCS#12 file format.
alias	The key alias name. The key alias is used by the key locator to find the key within the keystore file.
keyPassword	The key password that is used for recovering the key. This password is needed to access the key object within the keystore file.
keyName	The name of the key. For digital signatures, the key name is used by the request generator or response consumer signing information to determine which key is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption. The key name must be a fully qualified, distinguished name (DN). For example, `CN=Bob,O=IBM,C=US`.
trustAnchorPath	The file path from which the trust anchor is loaded.

Table 124. Information for the X.509 token (continued). Use the X.509 token to authenticate messages.

Token Information	Description
trustAnchorType	The type of trust anchor.
trustAnchorPassword	The password that is used to check the integrity of the trust anchor or the password used to unlock the keystore.
certStores	A list of certificate stores. A collection certificate store includes a list of untrusted, intermediary certificates and certificate revocation lists (CRLs). The collection certificate store is used to validate the certificate path of the incoming X.509-formatted security tokens.
provider	The security provider.

The following can be specified for a X.509 token:

a. Without any keystore.

b. With a trust anchor. A trust anchor specifies a list of keystore configurations that contain trusted root certificates. These configurations are used to validate the certificate path of incoming X.509-formatted security tokens. For example, when you select the trust anchor or the certificate store of a trusted certificate, you must configure the trust anchor and the certificate store before setting the certificate path.

c. With a keystore that is used for the key locator.

First, you must have created the keystore file, by using a key tool utility, for example. The keystore is used to retrieve the X.509 certificate. This entry specifies the password that is used to access the keystore file. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trustworthiness of a certificate chain. The names of the trust anchor and the collection certificate store are created in the certificate path under your token consumer.

d. With a keystore that is used for the key locator and the trust anchor.

e. With a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the consumer token, and represents the token type of this class:

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3
Specifies an X.509 certificate token.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1
Specifies X.509 certificates in a public key infrastructure (PKI) path. This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used. If you use this implementation, you must provide a key store password, path, and type on this panel.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7
Specifies a list of X.509 certificates and certificate revocation lists in a PKCS#7 format. This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web Services Security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You can specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. If you use this implementation, you must provide a key store password, path, and type.

For some tokens, WebSphere Application Server provides a predefined local name for the value type. When you specify the following local name, you do not need to specify a value type URI:

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
For an LTPA token, you can use `LTPA` for the value type local name. This local name

causes `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` to be specified for the value type Uniform Resource Identifier (URI).

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
For LTPA token propagation, you can use `LTPA_PROPAGATION` for the value type local name. This local name causes`http://www.ibm.com/websphere/appserver/tokentype` to be specified for the value type Uniform Resource Identifier (URI).

8. If the Username token is specified as the token consumer class name, the following token information can be specified:
   a. Whether to specify the nonce.

      This option indicates whether a Nonce is included for the token consumer. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of Username tokens. Nonce is valid only when the validating token type is a Username token, and it is available only for the response consumer binding.
   b. Specifies the keyword of the time stamp. This option indicates whether to verify a time stamp in the Username token. The time stamp is valid only when the incorporated token type is a Username token.
   c. Specifies a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the consumer token, and represents the token type of this class:

      URI value type: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
      Specifies a Username token.
9. If a Kerberos v5 token is specified as the token generator class name, the following token information can be specified:

Token Information	Description	Default Value
tokenValueType	Kerberos token value type in QName defined by Oasis Kerberos Token Profile v1.1 specification.	http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ
requireDKT	A boolean value to require a derived key for message protection.	false
clabel	The client label for the derived key.	WS-SecureConversation Specify null to use the default value.
slabel	The service label for the derived key.	WS-SecureConversation Specify null to use the default value.
keylen	The length of the derived key.	16 Specify zero to use the default value
supportTokenRequireSHA1	A boolean value to require a SHA1 key that is used in subsequent request messages when the Kerberos token is used as a supporting token.	false SHA1 key is consumed only if the supporting Kerberos token is protected. If set to true, the SHA1 key is always consumed.

Token Information	Description	Default Value
decComponent	An instance of WSSDecryption .	Set decComponent and verComponent to null to initialize this first for either the decryption or verification component. Then, use the initialized component only in the callback handler constructor for the second component.
verComponent	An instance of WSSVerfication.	Set decComponent and verComponent to null to initialize this first for either the decryption or verification component. Then, use the initialized component only in the callback handler constructor for the second component.

Additional token value types are defined in the OASIS Kerberos Token Profile v1.1 specification. Specify the token value type as the local name. It is not necessary to specify the value type URI for the Kerberos v5 token.

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

10. If secure conversation is used for message protection, then the following information must be specified:

Information	Description
EncryptionAlgorithm	This determines the key size.
cLabel	The client label used when creating the derived key.
sLabel	The server label used when creating the derived key.

11. Set the components into the wssConsumingContext object.
12. Invoke the wssConsumingContext.process() method.

Results

Using the WSS APIs, you have configured the token consumer.

What to do next

You must specify a similar token generator configuration, if not already completed.

Configuring the consumer security tokens using the WSS API:

You can secure the SOAP messages, without using policy sets, by using the Web Services Security APIs. To configure the token on the consumer side, use the Web Services Security APIs (WSS API). The consumer security tokens are part of the com.ibm.websphere.wssecurity.wssapi.token interface package.

Before you begin

The pluggable token framework in WebSphere Application Server has been redesigned so that the same framework from the WSS API can be reused. The same implementation of creating and validating security token can be used both for the Web Services Security run time and for the WSS API application code. The redesigned framework also simplifies the SPI programming model and will make it easier to add security token types.

You can use the WSS API or you can configure the tokens by using the administrative console. To configure tokens, you must have completed the following token task: configure the generator tokens, as needed.

About this task

On the generator side, the JAAS CallbackHandler and JAAS LoginModule are responsible for creating the security token. The token is created by using the JAAS LoginModule and by using JAAS CallbackHandler to pass authentication data. Then, the JAAS LoginModule creates the securityToken object, such as the UsernameToken, and passes it to the Web Services Security run time.

On the consumer side, the XML format is passed to the JAAS LoginModule for validation or authentication. then the JAAS CallbackHandler is used to pass authentication data from the Web Services Security run time to the LoginModule. After the token is authenticated and a security token object is created, then the token is passed it to the Web Services Security run time.

When using the WSS API for consumer token validation, certain default behaviors occur. The simplest way to use the WSS API is to use the default JAAS login module and callback handler. The example uses the default for them so the example does not specify the JAAS login module name.

The simplest way to use the WSS API is to use the default behavior (see the example code). The WSS API provide defaults for the token type, the token value, and the JAAS configuration name. The default token behaviors include:

Table 125. Default token behaviors. Several token characteristics are configured by default.

Consumer token decisions	Default behavior
Which token type to use	The token type specifies which type of token to use for signing and validating messages. The X.509 token is the default token type. WebSphere Application Server provides the following pre-configured consumer token types: • Security context token • Derived key token • X509 tokens You can also create custom token types, as needed.
What JAAS login configuration name to specify	The JAAS login configuration name specifies which JAAS login configuration name to use.
Which configuration type to use	The JAAS login module configuration type. Only the pre-configured consumer configuration types can be used for consumer token types.

The SecurityToken class (com.ibm.websphere.wssecurity.wssapi.token.SecurityToken) is the generic token class and represents the security token that has methods to get the identity, XML format, and cryptographic keys. Using the SecurityToken class, you can apply both the signature and encryption to the SOAP message. However, to apply both, you must have two SecurityToken objects, one for the signature and one for encryption, respectively.

The following token types are subclasses of the generic security token class:

Table 126. Subclasses of the SecurityToken. Use the subclasses to represent the security token.

Token type	JAAS login configuration name
Security context token	system.wss.consume.sct
Derived key token	system.wss.consume.dkt

The following token types are subclasses of the binary security token class:

Table 127. Subclasses to the BinarySecurityToken. Use the subclasses to represent the binary security token.

Token type	JAAS login configuration name
X.509 token	system.wss.consume.x509
X.509 PKI Path token	system.wss.consume.pkiPath
X.509 PKCS7 token	system.wss.consume.pkcs7

Note:

- For each JAAS login token consumer configuration name, there is a respective token generator configuration name. For example, for the X509Token, the respective token generator configuration name is system.wss.generate.x509.
- The LTPA and LTPA propagation tokens are only available to a requester that is running as a server-based client. The LTPA and LTPA propagation tokens are not supported for the Java SE 6 or Java EE application client.

To validate the X509Token to the SOAP message on the consumer side, the <X509Token> element must be in the <wsse:Security> element.

Procedure

1. To validate the securityToken package, com.ibm.websphere.wssecurity.wssapi.token, first ensure that the application server is installed.
2. *If using the default values*, configures the tokens for the Web Services Security token consumer process. , for each token type, the process is similar to the following token consumer process:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. Note that the WSSConsumingContext must always be called in a JAX-WS client application.
   c. Creates a JAAS CallbackHandler with information that is required to validate the security token. Review the token class information for which parameters are required or optional. For example, for an X.509 token, you could configure the following:

Table 128. X.509 token options. Use the X.509 configuration options to control the behavior of the token.

Token Information	Description
keyStoreRef	Indicates the reference name of the keystore that is stored in the cryptographic card. It can be specified when the card is set to the hardware.
keyStorePath	Indicates the path of the keystore file. It is not necessary to specify the keyStorePath if the keyStoreRef is set.
keyStorePassword	Indicates the password of the keystore file.
keyStoreType	Indicates the type of keystore file.
alias	Indicates the alias of the key.
keyPassword	Indicates the password of the key.
keyName	Indicates the subject name of the key.

   d. Sets the callback handler into WSSDecryption, WSSVerification, or WSSConsumingContext.
   e. If the callback handler is set into the WSSDecryption or WSSVerification, adds either one into WSSConsumingContext.
   f. Calls WSSConsumingContext.process().

3. *If using other than the default values*, configures the tokens for the Web Services Security token consumer process. For each token type, the process is similar to the following token consumer process:
   a. If you do not use the default JAAS login module and callback handler, you need to prepare a custom one and register the name of JAAS login configuration using the administrative console in advance.
   b. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   c. Creates the WSSConsumingContext instance from the WSSFactory instance. Note that the WSSConsumingContext must always be called in a JAX-WS client application.
   d. Creates a callback handler with information that is required to validate the security token. Review the token class information for which parameters are required or optional. For example, for a X.509 token, you can configure the following:

Table 129. X.509 token options. Use the X.509 configuration options to control the behavior of the token.

Token Information	Description
keyStoreRef	Indicates the reference name of the keystore that is stored in the cryptographic card. It can be specified when the card is set to the hardware.
keyStorePath	Indicates the path of the keystore file. It is not necessary to specify the keyStorePath if the keyStoreRef is set.
keyStorePassword	Indicates the password of the keystore file.
keyStoreType	Indicates the type of keystore file.
alias	Indicates the alias of the key.
keyPassword	Indicates the password of the key.
keyName	Indicates the subject name of the key.

   e. Sets JAAS configuration name and callback handler into WSSDecryption or WSSVerification, or WSSConsumingContext.
   f. If JAAS configuration name and callback handler are set into the WSSDecryption or WSSVerification, adds either one into WSSConsumingContext.
   g. Calls WSSConsumingContext.process().

Results

If there is an error condition, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and the security token on the consumer side is validated (authenticated).

Example

The following sample code provides the WSS API example code for decryption using the default JAAS login module and callback handler:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext gencont = factory.newWSSConsumingContext();

// Generate the callback handler (step: c)
   X509ConsumeCallbackHandler callbackHandler = new
         X509ConsumeCallbackHandler(
                                 "",
                                 "enc-sender.jceks",
                                 "jceks",
                                 "storepass".toCharArray(),
                                 "alice",
                                 "keypass".toCharArray(),
                                 "CN=Alice, O=IBM, C=US");

// Generate the WSSDecryption instance (step: d)
   WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
```

```
                                               callbackHandler);

// Add WSSDecryption to WSSConsumingContext (step: e)
   concont.add(dec);

// Validate the WS-Security header (step: f)
concont.process(msgcontext);
```

What to do next

For each token type, configure the token using the WSS APIs or using the administrative console. Next, specify the similar generator tokens if you have not done so.

If both the generator and consumer tokens are configured, continue securing SOAP messages at the response consumer using the WSS APIs or configure the tokens using the administrative console.

If both the generator and consumer tokens are configured, continue securing SOAP messages either by verifying the signature or by decrypting the message, as needed. You can use either the WSS APIs or the administrative console to secure the SOAP messages.

Configuring Web Services Security using the WSS APIs:

The Web Services Security application programming interfaces (WSS API) provide support for securing SOAP message.

Before you begin

Web Service Security supports the following programming models:

- Programming API for securing SOAP message with Web Services Security (WSS API).

  The API programming model design has been redesigned. The new design is an interface-based programming model and is based on Web Services Security Version 1.1 standards but the design also includes support for Web Services Security Version 1.0 for securing the SOAP message. The WSS API programming model implementation is a simplified version, which is based on an early draft proposal of JSR-183, which is the JSR for defining Java API binding for Web Services Security. By design, because the application code is programmed to the interface, any application code that is programmed with the open source implementation should be able to run on the WebSphere Application Server with minimal changes or no changes at all.
- Service Programming Interfaces (SPI) for a service provider

  Similarly, the Web Services Security run time token generation and token consuming SPI have been redesign so that the same security token interface and JAAS Login Module implementation can be used for both the WSS API and the SPI. The WSS SPI for the service provider extend the security token types and provide keys and deriving keys for signing, signature verification, encryption and decryption.

Usage statement: You must use the IBM implementation of the WS-Security standards in the context of web services.

About this task

These programming models extend the following functions :

- Security token types and deriving keys for signing
- Signature and verification
- Encryption and decryption

The following figure demonstrates how to use the simplified WSS APIs to secure a SOAP message by using XML digital signature and XML encryption.

The configuration model for web services has also been redesigned from a deployment descriptor model to a policy set model. The configuration programming model is based on configuring policy sets using a security policy to specify security constraints.

The functions provided by the policy set configurations are the same as the functions supported by the WSS API for the Web Services Security run time. However, the security policy that is defined using policy sets has a higher priority over the WSS API. When the WSS API and the policy set are both used in the application, the default behavior is for the security policy from the policy set to be enforced and the WSS API to be ignored. To use the WSS API in the application, you must make sure that there is no policy set attached to the application or to the application resources, or make sure there is no security policy in the attached policy set.

Web Service Security can be enabled by either using a policy set that is configured by using the administrative console, or by using the WSS API for configuration.

Using the WSS API, complete the following high-level steps to secure the SOAP message:

Procedure

1. Use the WSSSignature API to configure the signing information for the request generator (client side) binding. Different message parts can be specified in the message protection for a request on the generator side. The default required parts are BODY, ADDRESSING_HEADERS, and TIMESTAMP.

   The WSSSignature API also specifies the different algorithm methods to be used with the signature for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.
2. Use the WSSSignPart API if you want to add or change the signed parts to be used for message protection. The default signed parts are WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS, and WSSSignature.TIMESTAMP.

   The WSSSignPart API also specifies the different algorithm methods to be used if you added or changed the signed parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N. For example, use the WSSSignPart API if you want to generate the signature for the SOAP message using the SHA256 digest method instead of the default value of SHA1.
3. Use the WSSEncryption API to configure the encryption information on the request generator side. The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The default targets of encryption are BODY_CONTENT and SIGNATURE

   The WSSEncryption API also specifies the different algorithm methods to be used to protect message confidentiality. The default data encryption method is AES128. The default key encryption method is KW_RSA_OAEP.
4. Use the WSSEncryptPart API if you want to add or change the encrypted parts to be used for message confidentiality. For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

   No algorithm methods are required for encrypted parts.
5. Use the WSS API to attach the token on the generator side. The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for creating the security token on the generator side. Different stand-alone tokens can be sent in request or response. The default token is the X509Token. The other token that can be used for signing is the DerivedKeyToken, which is used only with Web Services Secure Conversation (WS-SecureConversation).
6. Use the WSSVerification API to verify the signature for the response consumer (client side) binding. Different message parts can be specified in the message protection for a response on the consumer side. The required targets for verification are BODY, ADDRESSING_HEADERS, and TIMESTAMP.

The WSSVerification API also specifies the different algorithm methods to be used for verifying the signature and for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.

7. Use the WSSVerifyPart API to add or change the verify signed parts to be used for message protection. The required verify parts are WSSVerification.BODY, WSSVerification.ADDRESSING_HEADERS, and WSSVerification.TIMESTAMP.

   The WSSVerifyPart API also specifies the different algorithm methods to be used if you added or changed the verification parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N.

8. Use the WSSDecryption API to configure the decryption information for the response consumer (client side) binding. The decryption information on the consumer side is used for decrypting an incoming SOAP message. The default targets of decryption are BODY_CONTENT and SIGNATURE. The default data encryption method is AES128. The default key encryption method is KW_RSA_OAEP.

   No algorithm methods are required for decryption.

9. Use the WSSDecryptPart API if you want to add or change the decrypted parts to be used for message confidentiality. For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

   No algorithm methods are required for decrypted parts.

10. Use the WSS API to configure the token on the consumer side. The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for validating (authenticating) the security token on the consumer side. Different stand-alone tokens can be sent in request or response.

    The WSS API adds the information for the candidate token that is used for decryption. The default token is X509Token.

Results

What to do next

The Web Services Security run time token generation and token consuming Service Programming Interfaces (SPI) have been redesign so that the same Security Token interface and JAAS Login Module implementation can be used in both the WSS API and the SPI. See the SPI information for detail descriptions.

Web Services Security APIs:

The Web Services Security programming model provides application programming interfaces (WSS API) for securing the SOAP message. The WSS API model is based on Web Services Security Version 1.1 standards but also includes support for Web Services Security Version 1.0.

The Web Services Security APIs (WSS APIs) can generate and process the following SOAP-related bindings for XML security:

- XML signature and signature verification
- XML encryption and decryption

The token processing and pluggable token architecture in the Web Service Security run time has been redesign to reuse the same Security Token interface and the JAAS Login Module as those used for the WSS APIs.

The following table lists the WSS API interfaces that are provided with WebSphere Application Server and used to configure signing and encryption information in the SOAP bindings for the generator and consumer bindings.

Table 130. WSS API interfaces. Use the interfaces to configure security information in the bindings.

WSS API interfaces	Description
WSSDecryption	Package: com.ibm.websphere.wssecurity.wssapi.decryption This interface is responsible for specifying decryption. The default values for decryption include: • Targets: BODY_CONTENT, SIGNATURE • Data encryption method: AES128 • Key encryption method: KW_RSA_OAEP • Security token: X.509
WSSDecryptPart	Package: com.ibm.websphere.wssecurity.wssapi.decryption This interface is responsible for adding decrypted parts, as needed. If specified, the default values for decrypted parts include: • Security token: X.509 • Transform method: N/A (not applicable)
WSSEncryption	Package: com.ibm.websphere.wssecurity.wssapi.encryption This interface is responsible for the encryption component. The default values for encryption include: • Targets: BODY_CONTENT, SIGNATURE • Data encryption method: AES128 • Key encryption method: KW_RSA_OAEP • Security token: X.509 • refType: SecurityToken.REF_KEYID • mtomOptimize: false
WSSEncryptPart	Package: com.ibm.websphere.wssecurity.wssapi.encryption This interface is responsible for adding encrypted parts, as needed. If specified, the default values for encrypted parts include: • Transform method: N/A (not applicable)
WSSSignature	Package: com.ibm.websphere.wssecurity.wssapi.signature This interface is responsible for specifying the signature. The default values for signature include: • Targets: BODY, ADDRESSING_HEADERS, TIMESTAMP • Signature method: RSA_SHA1 • Canonicalization method: EXC_C14N • Security token: X.509 • Type of token reference: SecurityToken.REF_STR
WSSSignPart	Package: com.ibm.websphere.wssecurity.wssapi.signature This interface is responsible for adding signed parts, as needed. If specified, the default values for signed parts include: • Transform method : TRANSFORM_EXC_C14N • Digest method: SHA1
WSSVerification	Package: com.ibm.websphere.wssecurity.wssapi.verification This interface is responsible for specifying the signature verification. The default values for verification include: • Targets: BODY, ADDRESSING_HEADERS, TIMESTAMP • Signature method: RSA_SHA1 • Canonicalization method: EXC_C14N • Security token: X.509
WSSVerifyPart	Package: com.ibm.websphere.wssecurity.wssapi.verification This interface is responsible for adding verify parts, as needed. If specified, the default values for verify parts include: • Digest method: SHA1 • Transform method: TRANSFORM_EXC_C14N

Also see the information about pre-configured generator and consumer tokens.

Web Services Security configuration considerations when using the WSS API:

To secure Web Services Security for WebSphere Application Server, you can specify several different configurations using the Web Services Security APIs (WSS API). The Web Services Security specification provides a flexible way to secure web services messages using XML digital signature, XML encryption, and attaching security tokens. You can enable Web Services Security by either configuring a policy set or by using the Web Services Security APIs (WSS API). The implementation for WSS API has default values for which message parts are to be signed or encrypted. The default values for the WSS APIs help end users to enable Web Services Security quickly.

Different message parts can be specified in the message protection for request or response, and different stand-alone tokens can be sent in request or response. However, there is only one symmetric or one asymmetric binding assertion to describe the token type and the algorithm that is used for message protection.

Using the WSS API, you can override any default values. However, when you alter the protection parts, note that all the default protection parts are cleared. For example, if you specify that you want to encrypt the Username token instead of the default X.509 token, all the default values of the encrypting protection parts are cleared.

The following table shows an example of the relationships between each of the configurations:

Table 131. Request generator and response consumer configurations. Use the table to determine the mapping between the configurations and the default values.

Type of configuration	Configuration name	Configurations and default values
Request generator	Signing information	• Canonicalization method: WSSSignature.EXC_C14N • Signature method: WSSSignature.RSA_SHA1 • Digest method: WSSSignPart.SHA1 • Transform method: WSSSignPart.TRANSFORM_EXC_C14N • Signed part - Body: WSSSignature.BODY • Signed part - Addressing: WSSSignature.ADDRESSING_HEADERS • Signed part - Timestamp: WSSSignature.TIMESTAMP • Token reference: SecurityToken.REF_STR • Token - Value type: X509Token.ValueType • Token - JAAS login configuration name: system.wss.generate.x509
Response consumer	Signature verification information	• Canonicalization method: WSSVerification.EXC_C14N • Signature method: WSSVerification.RSA_SHA1 • Transform method: WSSVerifyPart.TRANSFORM_EXC_C14N • Signed part - Body: WSSVerification.BODY • Signed part - Addressing: WSSVerification.ADDRESSING_HEADERS • Signed part - Timestamp: WSSVerification.TIMESTAMP • Token - Value type: X509Token.ValueType • Token - JAAS login configuration name: system.wss.consume.x509
Request generator	Encryption information	• Encrypted key: true • Key encryption method: WSSEncryption.KW_RSA_OAEP • Data encryption method: WSSEncryption.AES128 • Encryption part: WSSEncryption.BODY_CONTENT • Token reference: SecurityToken.REF_KEYID • Token - Value type: X509Token.ValueType • Token - JAAS login configuration name: system.wss.generate.x509
Response consumer	Decryption information	• Encrypted key: true • Key decryption method: WSSDecryption.KW_RSA_OAEP • Data decryption method: WSSDecryption.AES128 • Decryption part: WSSDecryption.BODY_CONTENT • Token - Value type: 509Token.ValueType • Token - JAAS login configuration name: system.wss.consume.x509

Encrypted SOAP headers:

The encrypted header element provides a standard way of encrypting SOAP headers. As one of the extensions to the OASIS SOAP message security specification, the encrypted header element indicates that the responder has processed the request. Encrypting SOAP headers and parts help to provide more secure message-level security.

The EncryptedHeader or <wsse11:EncryptedHeader> element is a part of the updated Web Services Security Version 1.1 standard and enables interoperability with other vendors that support the Version 1.1 standards, such as Microsoft .NET and DataPower.

Use the EncryptedHeader element for encrypting SOAP header blocks. The EncryptedHeader element allows Web Services Security to be compliant with the SOAP mustUnderstand processing guidelines and to prevent disclosure of information that is contained in attributes on a SOAP header block.

The <wsse11:EncryptedHeader> element must contain one <xenc:EncryptedData> element. Only one <xenc:EncryptedData> element per encrypted header element is permitted.

Encrypted data element

Normally, the programming model, such as JAX-WS, deserializes the SOAP message to a Java binding object before dispatching the call to the application code. However, if the SOAP message is encrypted, the deserialization fails because, before encryption, the original content is replaced with the EncryptedData XML element from the XML Encryption standard.

In certain cases, it might be desirable for the token that is included in the <wsse:Security> header to be encrypted for the recipient processing role.

Follow these guidelines when using the EncryptedData element:
- The EncryptedHeader element must contain one EncryptedData element.
- The <xenc:EncryptedData> element may be used to contain a security token and include it in the <wsse:Security> header.
- The <xenc:EncryptedData> must not include an XML ID for referencing the contained security token.
- All <xenc:EncryptedData> tokens must either have an embedded encryption key or must be referenced by a separate encryption key.
- If compliance with Basic Security Profile 1.1 is desired, the <xenc:EncryptedData> element must have an Id attribute.

Policy assertion for encrypted parts

The EncryptedParts policy assertion specifies which header is to be encrypted in the security policy. The following table describes the elements and attributes that can be used for EncryptedParts.

Table 132. Attributes and elements of the EncryptedParts element. Use encrypted parts to provide more secure message-level security.

Element or attribute	Description
/sp:EncryptedParts/sp:Header	Optional. Presence of this optional element indicates that a specific SOAP header (or set of such headers) must be protected. You can have multiple sp:Header elements within a single EncryptedParts element. Each header (or set of headers) must be encrypted, and this encryption will encrypt the elements by using Web Services Security Version 1.1 encrypted headers. As such, if WS-Secuirty 1.1 Encrypted Headers are not supported by a service, then the headers cannot be encrypted by using message-level security. If multiple SOAP headers with the same local name but different namespace names are to be encrypted, multiple sp:Header elements are required, either as part of a single sp:EncryptedParts assertion or as part of separate sp:EncryptedParts assertions.

Table 132. Attributes and elements of the EncryptedParts element (continued). Use encrypted parts to provide more secure message-level security.

Element or attribute	Description
/sp:EncryptedParts/sp:Header/@Name	Optional. This attribute indicates the local name of the SOAP header to be confidentiality protected. If this attribute is not specified, all SOAP headers whose namespace matches the Namespace attribute are to be protected.
/sp:EncryptedParts/sp:Header/@Namespace	Required. This attribute indicates the namespace of the SOAP headers to be confidentiality protected.

The following message example shows what the EncryptedHeader element looks like on a message where the EncryptedParts policy assertion for the encrypted header has been specified on the policy:

```
<S:Envelope xmlns:S="..." xmlns:wsse="..." xmlns:wsse11="..." xmlns:wsu="..."
   xmlns:xenc="..." xmlns:ds="...">
   <S:Header>
     <wsse:Security>
      <!-- Tokens etc. -->
       <xenc:EncryptedKey>
         <xenc:EncryptionMethod Algorithm="..."/>
           <ds:KeyInfo>
             ... ...
           </ds:KeyInfo>
          <xenc:CipherData>
            <xenc:CipherValue>...</xenc:CipherValue>
          </xenc:CipherData>
          <xenc:ReferenceList>
            <xenc:DataReference URI="#hdrID"/>
          </xenc:ReferenceList>
        </xenc:EncryptedKey>
     </wsse:Security>
     <wsse11:EncryptedHeader wsu:Id="hdrID">
       <xenc:EncryptedData Id="encDataID">
         <xenc:CipherData>
           <xenc:CipherValue>...</xenc:CipherValue>
         </xenc:CipherData>
         ...
       </xenc:EncryptedData>
     </wsse11:EncryptedHeader>
   </S:Header>
   <S:Body>
     ...
   </S:Body>
</S:Envelope>
```

To encrypt headers in the Web Services Security Version 1.0 specification format, specify the com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.0 property with a value of `true` on the <encryptionInfo> element in the binding. When this property is specified, the target header for encryption is replaced by an <EncryptedData> element, instead of an <EncryptedHeader> element that contains an <EncryptedData> element.

For Web Services Security Version 1.1 behavior that is equivalent to WebSphere Application Server versions prior to version 7.0, specify the com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.1.pre.V7 property with a value of `true` on the <encryptionInfo> element in the binding. When this property is specified, the <EncryptedHeader> element includes a wsu:Id parameter and the <EncryptedData> element omits the Id parameter. This property should only be used if compliance with Basic Security Profile 1.1 is not required.

For complete information about the EncryptedHeader element and the EncryptedData element, see the Web Services Security Version 1.1 specification.

Signature confirmation:

Web Services Security signature confirmation is an enhanced XML digital signature, and it is included in the Web Services Security standard. XML digital signature is used for signing elements of the SOAP envelope.

As one of the extensions to the OASIS SOAP message security specification, the signature confirmation element incorporates the elements that are needed within the response message in order to confirm the signature that is contained in a request message. XML digital signature and signature confirmation help to provide more secure message-level security.

Web Services Security Version 1.0 for SOAP message security did not provide any guidance on how to confirm mutual understanding of the request that prompted this response. The SignatureConfirmation or <wsse11:SignatureConfirmation> element has been added to the Web Services Security Version 1.1 specification. The <wsse11:SignatureConfirmation> element ensures that the signature is processed by the intended recipient and indicates that the responder has processed the signature in the request. The signature confirmation element is part of the updated Web Services Security standard and enables interoperability with other vendors that support the Version 1.1 standards, such as Microsoft .NET and DataPower.

Because of the stateless nature of web services and due to different message exchange patterns (MEPs), consider the following assumptions:

- Assume that session affinity is enabled if a cluster is enabled for the clients that are running in WebSphere Application Server. When session affinity is enabled, it implies that the response is sent back to the initiating client of the server.
- Assume WS-Addressing is enabled for asynchronous message exchange patterns. When WS-Addressing is enabled, it allows the run time to relate the response back to the request. An asynchronous response is sent back to the application of the initiating WebSphere Application Server.

Syntax

The SignatureConfirmation element indicates that the responder has processed the signature in the request. When this element is not present in a response, the initiator interprets that the responder is not compliant.

The format for the signature confirmation element is as follows:

```
<wsse11:SignatureConfirmation wsu:Id="..." Value="..." />
```

where:

wsu:Id
The identifier that is used when referencing this element in the <ds:SignedInfo> reference list of the signature of the associated response message. This attribute is required so that unambiguous references are made to this <wsse11:SignatureConfirmation> element.

Value
This attribute is optional and contains the contents of a <ds:SignatureValue> that is copied from the associated request. If the request is unsigned, this attribute must not be present. If this attribute is specified without a value (empty), the initiator interprets this as incorrect behavior and processes it accordingly. When this attribute is not present, the initiator interprets this to mean that the response is based on a request that was not signed.

Configuration

To configure signature confirmation, configure the policy file using the administrative console, and select **Require signature confirmation**. To process Signature Confirmation correctly, the initiator of the request needs to preserve the signatures during request generator processing and later needs to retrieve the signatures for confirmation checks.

Response generation rules

Additional SOAP security elements for the SOAP responder are used to confirm that the response is in relationship to a particular request. The responder must include the contents of the <ds:SignatureValue> element of the request signature as the value of the @Value attribute of the <wsse11:SignatureConfirmation> element.

The following response generation rules apply when using the SignatureConfirmation policy assertion:
- If there are no signatures on the request, the response contains one SignatureConfirmation element, without a value. For MEPs where there are multiple requests (all without signatures) and one response, the response contains one SignatureConfirmation element without a value.
- If there are signatures on the request, the response contains a SignatureConfirmation element for each signature, with a value that matches the signature value on the request. For MEPs where there are multiple requests, with at least one containing a signature, and one response, the response contains a SignatureConfirmation element for each signature that is found on the requests, with a value that matches the signature value on the request.
- For MEPs where there is one request and multiple responses, each response contains the appropriate SignatureConfirmation elements as noted in the first and second bullets.
- If the SOAP request contains multiple signatures, the requester will find all of the signature confirmation elements contained in the response, and will check the values of the value fields of the signature confirmation elements against the values of the signatures in the original SOAP request.

Developing JAX-WS based web services client applications that retrieve security tokens:

The security handlers are responsible for propagating security tokens. These security tokens are embedded in the SOAP security header and passed to downstream servers.

About this task

This information applies only to Java API for XML-based Web Services (JAX-WS) .

The security tokens are encapsulated in the implementation classes for the com.ibm.wsspi.wssecurity.auth.token.Token interface. You can retrieve the security token data from either a server application or a client application.

With a client application, the application serves as the request generator and the response consumer and runs as the Java Platform, Enterprise Edition (Java EE) client application. The consumer component for Web Services Security stores the security tokens that it receives in one of the properties of the MessageContext object for the current web services call. You can retrieve a set of token objects through the javax.xml.rpc.Stub interface of that web services call. You must know which security tokens to retrieve and their token IDs in case multiple security tokens are included in the SOAP security header. Complete the following steps to retrieve the security token data from a client application:

Procedure

1. Use the com.ibm.wsspi.wssecurity.token.tokenPropergation key string to obtain the Hashtable for the tokens through a property value in the javax.xml.ws.Stub interface. The following example shows how to obtain the Hashtable:

   ```
   java.util.Hashtable t;

   javax.xml.ws.Service serv = ...;
   serv.addPort(...);
   javax.xml.ws.Dispatch<Object> dispatch = svc.createDispatch(...);

   Map<String, Object> requestContext = dispatch.getRequestContext();
   requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, ..);
   requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, ..);
   ```

```
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, ..);

String response = dispatch.invoke(body.toString());

Map<String, Object> responseContext = dispatch.getResponseContext();

t = (Hashtable) responseContext.get(
com.ibm.wsspi.wssecurity.Constants.WSSECURITY_TOKEN_PROPERGATION);
```

2. Search the targeting token objects in the Hashtable. Each token object in the Hashtable is set with its token ID as a key. You must have prior knowledge of the security token IDs to retrieve the security tokens. The following example shows how to retrieve a username token from the security header with a certain token ID value:

```
com.ibm.wsspi.wssecurity.auth.token.UsernameToken unt;
if (t != null) {
  unt = (com.ibm.wsspi.wssecuty.auth.token.UsernameToken)t.get("...");
}
```

Results

After completing these steps, you have retrieved the security tokens that are processed by the Web Services Security handler in a client application.

Developing JAX-WS based web services server applications that retrieve security tokens:

With a server application, the application acts as the request consumer, and the response generator is deployed and runs in the Java Platform, Enterprise Edition (Java EE) container. The consumer component for Web Services Security stores the security tokens that it receives in the Java Authentication and Authorization Service (JAAS) Subject of the current thread. You can retrieve the security tokens from the JAAS Subject that is maintained as a local thread in the container.

About this task

This information applies only to Java API for XML-based Web Services (JAX-WS).

The security handlers are responsible for propagating security tokens. These security tokens are embedded in the SOAP security header and passed to downstream servers. The security tokens are encapsulated in the implementation classes for the com.ibm.wsspi.wssecurity.auth.token.Token interface. You can retrieve the security token data from either a server application or a client application.

Complete the following steps to retrieve the security token data from a server application:

Procedure

1. Obtain the JAAS Subject of the current thread using the WSSubject API. If you enable Java 2 Security on the Global security panel in the administrative console, access to the JAAS Subject is denied if the application code is not granted the javax.security.auth.AuthPermission("wssecurity.getCallerSubject") permission. The following code sample shows how to obtain the JAAS subject:

```
javax.security.auth.Subject subject;

try {
subject = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();
} catch (com.ibm.websphere.security.WSSecurityException e) {
  ...
}
```

2. Obtain a set of private credentials from the Subject. For more information, see the application programming interface (API) com.ibm.websphere.security.auth.WSSubject class through the information center . To access this information within the information center, click **Reference >**

Developer > API Documentation > Application Programming Interfaces. In the Application Programming Interfaces article, click **com.ibm.websphere.security.auth > WSSubject**.

Attention: When Java 2 Security is enabled, you might need to use the AccessController class to avoid a security violation that is caused by operating the security objects in the Java EE container.

The following code sample shows how to set the AccessController class and obtain the private credentials:

```
Set s = (Set) AccessController.doPrivileged(new PrivilegedAction() {
   public Object run() {
   return subj.getPrivateCredentials();
   }
      });
```

3. Search the targeting token class in the private credentials. You can search the targeting token class by using the java.util.Iterator interface. The following example shows how to retrieve a username token with a certain token ID value in the security header. You can also use other method calls to retrieve security tokens. For more information, see the application programming interface (API) documents for the com.ibm.wsspi.wssecurity.auth.token.Token interface or custom token classes.

```
com.ibm.wsspi.wssecurity.auth.token.UsernameToken unt;
Iterator it = s.iterator();
while (it.hasNext()) {
  Object obj = it.next();
  if (obj != null &&
obj instanceOf com.ibm.wsspi.wssecurity.auth.token.UsernameToken) {
    unt =(com.ibm.wsspi.wssecurity.auth.token.UsernameToken) obj;
if (unt.getId().equals("...")) break;
else continue;
  }
}
```

Results

After completing these steps, you have retrieved the security tokens from the JAAS Subject in a server application.

Developing message-level security for JAX-RPC web services

IBM® WebSphere® Application Server supports the Java™ API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model.

Developing web services clients that retrieve tokens from the JAAS Subject in an application:

The security handlers are responsible for propagating security tokens. These security tokens are embedded in the SOAP security header and passed to downstream servers.

About this task

This information applies only to Java API for XML-based RPC (JAX-RPC) Web services.

The security tokens are encapsulated in the implementation classes for the com.ibm.wsspi.wssecurity.auth.token.Token interface. You can retrieve the security token data from either a server application or a client application.

With a client application, the application serves as the request generator and the response consumer and runs as the Java Platform, Enterprise Edition (Java EE) client application. The consumer component for Web Services Security stores the security tokens that it receives in one of the properties of the MessageContext object for the current Web services call. You can retrieve a set of token objects through the javax.xml.rpc.Stub interface of that web services call. You must know which security tokens to retrieve and their token IDs in case multiple security tokens are included in the SOAP security header. Complete the following steps to retrieve the security token data from a client application:

Procedure

1. Use the com.ibm.wsspi.wssecurity.token.tokenPropergation key string to obtain the Hashtable for the tokens through a property value in the javax.xml.rpc.Stub interface. The following example shows how to obtain the Hashtable:

```
java.util.Hashtable t;
javax.xml.rpc.Service serv = ...;
MyWSPortType pt = (MyWSPortType)serv.getPort(MyWSPortType.class);
t = (Hashtable)((javax.xml.rpc.Stub)pt)._getProperty(
com.ibm.wsspi.wssecurity.Constants.WSSECURITY_TOKEN_PROPERGATION);
```

2. Search the targeting token objects in the Hashtable. Each token object in the Hashtable is set with its token ID as a key. You must have prior knowledge of the security token IDs to retrieve the security tokens. The following example shows how to retrieve a username token from the security header with a certain token ID value:

```
com.ibm.wsspi.wssecurity.auth.token.UsernameToken unt;
if (t != null) {
  unt = (com.ibm.wsspi.wssecuty.auth.token.UsernameToken)t.get("...");
}
```

Results

After completing these steps, you have retrieved the security tokens from the JAAS Subject in a client application

Developing web services applications that retrieve tokens from the JAAS Subject in a server application:

With a server application, the application acts as the request consumer, and the response generator is deployed and runs in the Java Platform, Enterprise Edition (Java EE) container. The consumer component for Web Services Security stores the security tokens that it receives in the Java Authentication and Authorization Service (JAAS) Subject of the current thread. You can retrieve the security tokens from the JAAS Subject that is maintained as a local thread in the container.

About this task

This information applies only to Java API for XML-based RPC (JAX-RPC) Web services.

The security handlers are responsible for propagating security tokens. These security tokens are embedded in the SOAP security header and passed to downstream servers. The security tokens are encapsulated in the implementation classes for the com.ibm.wsspi.wssecurity.auth.token.Token interface. You can retrieve the security token data from either a server application or a client application.

Complete the following steps to retrieve the security token data from a server application:

Procedure

1. Obtain the JAAS Subject of the current thread using the WSSubject utility class. If you enable Java 2 Security on the Global security panel in the administrative console, access to the JAAS Subject is denied if the application code is not granted the javax.security.auth.AuthPermission("wssecurity.getCallerAsSubject") permission. The following code sample shows how to obtain the JAAS subject:

```
javax.security.auth.Subject subj;
try {
subj = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();
} catch (com.ibm.websphere.security.WSSecurityException e) {
  ...
}
```

2. Obtain a set of private credentials from the Subject. For more information, see the application programming interface (API) com.ibm.websphere.security.auth.WSSubject class through the

information center . To access this information within the information center, click **Reference > Developer > API Documentation > Application Programming Interfaces**. In the Application Programming Interfaces article, click **com.ibm.websphere.security.auth > WSSubject**.

Attention: When Java 2 Security is enabled, you might need to use the AccessController class to avoid a security violation that is caused by operating the security objects in the Java EE container.

The following code sample shows how to set the AccessController class and obtain the private credentials:

```
Set s = (Set) AccessController.doPrivileged(new PrivilegedAction() {
   public Object run() {
   return subj.getPrivateCredentials();
   }
      });
```

3. Search the targeting token class in the private credentials. You can search the targeting token class by using the java.util.Iterator interface. The following example shows how to retrieve a username token with a certain token ID value in the security header. You can also use other method calls to retrieve security tokens. For more information, see the application programming interface (API) documents for the com.ibm.wsspi.wssecurity.auth.token.Token interface or custom token classes.

```
com.ibm.wsspi.wssecurity.auth.token.UsernameToken unt;
Iterator it = s.iterator();
while (it.hasNext()) {
  Object obj = it.next();
  if (obj != null &&
obj instanceOf com.ibm.wsspi.wssecurity.auth.token.UsernameToken) {
    unt =(com.ibm.wsspi.wssecurity.auth.token.UsernameToken) obj;
if (unt.getId().equals("...")) break;
else continue;
  }
}
```

Results

After completing these steps, you have retrieved the security tokens from the JAAS Subject in a server application

Web Services Security service provider programming interfaces

Several Service Provider Interfaces (SPIs) are provided to extend the capability of the Web Services Security runtime.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

The following list contains the SPIs that are available for WebSphere Application Server:

Procedure

- com.ibm.wsspi.wssecurity.config.KeyLocator is an abstract for obtaining the keys for digital signature and encryption. The following list contains the default implementations:
  1. com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator implements the Java key store.
  2. com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator povides a mapping of the authenticated identity to a key for encryption or, the implementation uses the default key that is specified.

3. com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator Provides the capability of using the signer key for encryption in the response message. This implementation is typically used in the response sender configuration.

- com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator is an interface that is used to evaluate the trust for identity assertion. The default implementation is com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl, which enables you to define a list of trusted identities.
- The Java Authentication and Authorization Service (JAAS) CallbackHandler application programming interfaces (APIs) are used for token generation by the request sender. This interface can be extended to generate a custom token that can be inserted in the Web Services Security header. The following list contains the default implementations that are provided by WebSphere Application Server:
  1. com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler presents a login prompt to gather the basic authentication data. Use this implementation in the client environment only.
  2. com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler collects the basic authentication data in the standard in (stdin) prompt. Use this implementation in the client environment only.

     Restriction: If you have a multi-threaded client and multiple threads attempt to read from standard in at the same time, all the threads will not successfully obtain the user name and password information. Therefore, you cannot use the com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler implementation with a multi-threaded client where multiple threads might attempt to obtain data from standard in concurrently.
  3. com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler reads the basic authentication data from the application binding file. This implementation might be used on the server side to generate a user name token.
  4. com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler Generates a Lightweight Third Party Authentication (LTPA) token in the Web Services Security header as a binary security token. If basic authentication data is defined in the application binding file, it is used to perform a login, to extract the LTPA token from the WebSphere credentials, and to insert the token in the Web Services Security header. Otherwise, it will extract the LTPA security token from the invocation credentials (RunAs identity) and insert the token in the Web Services Security header.

What to do next

The JAAS LoginModule API is used for token validation on the request receiver side of the message. You can implement a custom LoginModule API to perform validation of the custom token on the request receiver of the message. After the token is verified and validated, the token is set as the caller and then run as the identity in the WebSphere Application Server runtime. The identity is used for authorization checks by the containers before a Java Platform, Enterprise Edition (Java EE) resource is invoked. The following list presents the default AuthMethod configurations provided by WebSphere Application Server:

BasicAuth
Validates a user name token.

Signature
Maps the distinguished name (DN) of a verified certificate to a Java Authentication and Authorization Service (JAAS) subject.

IDAssertion
Maps a trusted identity to a JAAS subject.

LTPA Validates an LTPA token that is received in the message and creates a JAAS subject.

Configuring Web Services Security during application assembly

If you configure Web Services Security with an assembly tool, the Web Services Security binding information is modified

Configuring HTTP outbound transport level security with an assembly tool

You can configure the HTTP outbound transport level security with an assembly tool.

Before you begin

You can configure HTTP outbound transport level security with assembly tools provided with WebSphere Application Server.

This task is one of several ways that you can configure the HTTP outbound transport level security for a web service acting as a client to another web service server. You can also configure the HTTP outbound transport level security with the administrative console or by using the Java properties. If you do not configure the HTTP outbound transport level security, the web services runtime defers to the Java Platform, Enterprise Edition (Java EE) security runtime in the WebSphere product for an effective Secure Sockets Layer (SSL) configuration. If there is no SSL configuration with the Java EE security runtime in the WebSphere product, the Java Secure Socket Extension (JSSE) system properties are used.

About this task

If you configure the HTTP outbound transport level security with assembly tool or with the administrative console, the Web Services Security binding information is modified. If you have not yet installed the web services application into WebSphere Application Server, you can configure the HTTP SSL configuration with an assembly tool. This task assumes that you have not deployed the web services application into the WebSphere product.

If you configure the HTTP outbound transport level security using the standard Java properties for JSSE, the properties are configured as system properties. The configuration that is specified in the binding takes precedence over the Java properties. However, the configurations that are specified by the Java EE security programming model, or are associated with the Dynamic selection, have a higher precedence.

To learn more, see the secure communications using Secure Sockets Layer information.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate the web application archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
4. Configure the HTTP outbound transport level security. Read about enabling web service endpoints in the Rational Application Developer documentation.

Results

You have configured the HTTP outbound transport level security for a web service acting as a client to another web service with an assembly tool.

Configuring HTTP basic authentication for JAX-RPC web services with an assembly tool

You can configure HTTP basic authentication for Java API for XML-based RPC (JAX-RPC) web services with an assembly tool.

Before you begin

You can configure HTTP basic authentication with assembly tools provided with WebSphere Application Server.

About this task

This task is one of three ways that you can configure HTTP basic authentication. You can also configure HTTP basic authentication with the administrative console or by modifying the HTTP properties programmatically.

If you choose to configure the HTTP basic authentication with an assembly tool or with the administrative console, the Web Services Security binding information is modified. You can use an assembly tool to configure HTTP basic authentication before you deploy or install the web services application into WebSphere Application Server. This task assumes that you have not deployed the web services application into the WebSphere product.

If you configure HTTP basic authentication programmatically, the properties are configured in the Stub or Call instance. The values set programmatically take precedence over the values defined in the binding.

The HTTP basic authentication that is discussed in this topic is orthogonal to WS-Security and is distinct from basic authentication that WS-Security supports. WS-Security supports basic authentication token, not HTTP basic authentication.

To configure HTTP basic authentication, use the WebSphere Application Server tools to modify the binding information.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate the web application archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
4. Configure the HTTP basic authentication in the Web Services Client Port Binding page for a web service or a web service client. The Web Services Client Port Binding page is available after double-clicking the client deployment descriptor file. Read about Web Services Client Port Bindings in the Rational Application Developer documentation.

Configuring XML digital signature for Version 5.x web services with an assembly tool

XML digital signature is one of the methods WebSphere® Application Server provides to secure your web services. It provides message integrity and authentication capabilities when used with SOAP messages.

Configuring trust anchors using an assembly tool:

Use an assembly tool to configure trust anchors (that specify keystores which contain trusted root certificates to validate the signer certificate) or trust stores at the application level.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This document describes how to configure trust anchors or trust stores at the application level. It does not describe how to configure trust anchors at the server or cell level. Trust anchors defined at the application level have a higher precedence over trust anchors defined at the server or cell level. You can configure an application-level trust anchor using an assembly tool or the administrative console. This document describes how to configure the application-level trust anchor using an assembly tool.

About this task

A trust anchor specifies keystores that contain trusted root certificates, which validate the signer certificate. These keystores are used by the request receiver (as defined in the `ibm-webservices-bnd.xmi` file) and the response receiver (as defined in the `application-client.xml` file when web services are acting as client) to validate the signer certificate of the digital signature. The keystores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these keystores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `application-client.xml` file.

Complete the following steps to configure trust anchors using an assembly tool.

Procedure

1. Configure an assembly tool to work with a Java Platform, Enterprise Edition (Java EE) enterprise application. For more information, see the related information on Assembly Tools.
2. Create a web services-enabled Java EE enterprise application.
3. Configure the client-side response receiver, which is defined in the `ibm-webservicesclient-bnd.xmi` bindings extensions file.
   a. Use an assembly tool to import your Java EE application.
   b. Click **Window** > **Open Perspective** > **Other** > **J2EE**.
   c. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**
   d. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Binding** tab. The Client Deployment Descriptor is displayed.
   e. Locate the Port qualified name binding section and either select an existing entry or click **Add**, to add a new port binding. The web services client port binding editor displays for the selected port.
   f. Locate the Trust anchor section and click **Add**. The Trust anchor window is displayed.
      1) Enter a unique name within the port binding for the **Trust anchor name**.
         The name is used to reference the trust anchor that is defined.
      2) Enter the keystore password, path, and keystore type.
         The supported keystore types are the Java Cryptography Extension (JCE) and Java Cryptography Extension Keystores (JCEKS) types.

      Click **Edit** to edit the selected trust anchor.

      Click **Remove** to remove the selected trust anchor.

      When you start the application, the configuration is validated in the run time while the binding information is loading.
   g. Save the changes.

4. Configure the server-side request receiver, which is defined in the `ibm-webservices-bnd.xmi` bindings extensions file.
   a. Click **Window** > **Open perspective** > **J2EE**.
   b. Select the web services enabled Enterprise JavaBeans (EJB) or web module.
   c. In the Package Explorer window, click the `META-INF` directory for an EJB module or the `WEB-INF` directory for a web module.
   d. Right-click the `webservices.xml` file, select **Open with** > **Web services editor**, and click the bindings tab. The web services bindings editor is displayed.
   e. Locate the web service description bindings section and either select an existing entry or click **Add** to add a new web services descriptor.
   f. Click **Binding configurations**. The web services binding configurations editor is displayed for the selected web services descriptor.
   g. Locate the Trust anchor section and click **Add**. The Trust anchor dialog box is displayed.
      1) Enter a unique name within the binding for the **Trust anchor name**.

         This unique name is used to reference the trust anchor defined.
      2) Enter the keystore password, path, and keystore type. The supported keystore types are `JCE` and `JCEKS`.

      Click **Edit** to edit the selected trust anchor.

      Click **Remove** to remove the selected trust anchor.

      When you start the application, the configuration is validated in the run time while the binding information is loading.
   h. Save the changes.

Results

This procedure defines trust anchors that can be used by the request receiver or the response receiver (if the web services is acting as client) to verify the signer certificate.

Example

The request receiver or the response receiver (if the web service is acting as a client) uses the defined trust anchor to verify the signer certificate. The trust anchor is referenced using the trust anchor name.

What to do next

To complete the signing information configuration process for request receiver, complete the following tasks:
1. "Configuring the server for request digital signature verification: Verifying the message parts" on page 613
2. "Configuring the server for request digital signature verification: choosing the verification method" on page 614

To complete the process for the response receiver, if the web services is acting as a client, complete the following tasks:
1. "Configuring the client for response digital signature verification: verifying the message parts" on page 620
2. "Configuring the client for response digital signature verification: choosing the verification method" on page 622

Configuring the client-side collection certificate store using an assembly tool:

You can configure the client-side collection certificate store using the assembly tool.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using an assembly tool or the WebSphere Application Server administrative console. Complete the following steps to configure the client-side collection certificate store using the assembly tool.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Binding** tab, which is located at the bottom of deployment descriptor editor within the assembly tool. The Client Deployment Descriptor is displayed.
5. Click the **Port binding** tab in deployment descriptor editor within the assembly tool. The web services client port binding window is displayed.
6. Select one of the port-qualified name binding entries.
7. Expand the **Security response receiver binding configuration** > **certificate store list** > **Collection certificate store** section.
8. Click **Add** to create a new collection certificate store, click **Edit** to edit an existing certificate store, or click **Remove** to delete an existing certificate store.
9. Enter a name in the **Name** field. This name is referenced in the Certificate store reference field in the Signing info dialog box.
10. Leave the Provider field as `IBMCertPath`.
11. Click **Add** to enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`. If you have additional certificate store paths, click **Add** to add the paths.
12. Click **OK** when you finish adding paths.

Configuring the server-side collection certificate store using an assembly tool:

A *collection certificate store* is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collections of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message. You can configure the server-side collection certificate store by using an assembly tool.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

You can configure the collection certificate either by using an assembly tool or by using the WebSphere Application Server administrative console. Complete the following steps to configure the server-side collection certificate store using an assembly tool.

Procedure

1. Start an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, select **Open with** > **Web Services Editor**.
5. Click the Binding configurations tab in the web services editor within the assembly tool. The Web Service Binding Configuration window is displayed.
6. Select one of the web service description binding entries under the Port Component Binding section.
7. Expand the **Request receiver binding configuration details** > **Certificate store list** > **Collection certificate store** section.
8. Click **Add** to create a new collection certificate store, click **Edit** to edit an existing certificate store, or click **Remove** to delete an existing certification store.
9. Enter a name in the **Name** field. This name is referenced in the **Certificate store reference** field in the Signing info dialog.
10. Leave the **Provider** field as `IBMCertPath`.
11. Click **Add** to enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT]/etc/ws-security/samples/intca2.cer`. If you have additional certificate store paths, click **Add** to add the paths.
12. Click **OK** when you finish adding paths.

Configuring key locators using an assembly tool:

The following information provides instructions on how to configure key locators using an assembly tool.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

You can configure key locators in various locations within the assembly tool. The following procedure provides instructions on how to configure key locators at any of these locations because the concept is the same.

Procedure

1. Start an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Binding** tab. The Client Deployment Descriptor is displayed.
5. Click the **WS Binding** tab in deployment descriptor editor within the assembly tool or the **Binding configurations** tab in the Web services editor within the assembly tool.
6. Expand one of the **Binding configuration** sections.
7. Expand the **Key locators** section.

8. Click **Add** to create a new key locator, click **Edit** to edit an existing key locator, or click **Remove** to delete an existing key locator.
9. Enter a key locator name. The name entered for the **Key locator name** is used to refer to the key locator from the Encryption information and Signing Information sections.
10. Enter a key locator class. The key locator class is the implementation of the `KeyLocator` interface. When using default implementations, select a class from the menu.
11. Determine whether to click **Use key store**. Select this option when you use the default implementations as they use key stores. If you click **Use key store**, complete the following steps:
    a. Enter a value in the key store storepass field. The key store storepass is the password used to access the key store.
    b. Enter a path name in the key store path field. The key store path is the location on the file system where the key store resides. Make sure that the location can be found wherever you deploy the application.
    c. Enter a type value in the key store type field. The valid types to enter are JKS and JCEKS. JKS is used when you are not using the Java Cryptography Extensions (JCE) policy. JCEKS is used when you are using JCE. Although the JCEKS type is more secure, it might decrease performance.
    d. Click **Add** to create an entry for a key in the key store.
        1) Enter a value in the Alias field.

           The key alias is a reference to this particular key from the Signing Information section.
        2) Enter a value in the Key pass field.

           The key pass is the password associated with the certificate which is created using the Java SE Development Kit 6 `keytool.exe` file.
        3) Enter a value in the Key name field.

           The key name refers to the alias of the certificate as found in the key store.
12. Click **Add** to create a custom property. The property can be used by custom key locator implementations. For example, you can use properties with the `WSIdKeyStoreMapKeyLocator` default implementation. The key locator implementation has the following property names:
    - *id_*, which maps to a credential user ID.
    - *mappedName_* , which maps to the key alias to use for this user name.
    - *default*, which maps to a key alias to use when a credential does not have an associated *id_* entry.

    A typical set of properties for this key locator might be: `id_1=user1`, `mappedName_1=key1`, `id_2=user2`, `mappedName_2=key2`, `default=key3`. If `user1` or `user2` authenticates, then the associated `key1` or `key2` is used, respectively. However, if none of the user properties authenticate or the user is not `user1` or `user2`, then `key3` is used.

    a. Enter a name in the Name field. The name entered is the property name.
    b. Enter a value in the Value field. This value entered is the property value.

Securing web services for Version 5.x applications using XML digital signature:

XML digital signature is one of the methods WebSphere Application Server provides to secure your web services. It provides message integrity and authentication capabilities when used with SOAP messages.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services; XML digital signature is one of these methods. You can secure your web services by using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

About this task

XML digital signature provides both message integrity and authentication capabilities when it is used with SOAP messages. A message receiver can verify that attackers or accidents have not altered parts of the message after the message was signed by a key. If a message has a digital certificate issued by a certificate authority (CA) and a signature in the message is validated successfully by a public key in the certificate, it is proof that the signer has the corresponding private key. To use XML digital signature to secure web services, complete the following steps:

Procedure

1. Define the security constraints or extensions. To configure the security constraints, you must use an assembly tool. For more information, see the related information on Assembly Tools.
   a. Configure the client to digitally sign a message request. To configure the client, complete the following steps to specify which parts of the SOAP message to digitally sign and define the method used to digitally sign the message. The client in these steps is the request sender.
      1) Specify the message parts by following the steps found in "Configuring the client for request signing: digitally signing message parts" on page 609.
      2) Select the method used to digitally sign the request message. You can select the digital signature method by following the steps in "Configuring the client for request signing: choosing the digital signature method" on page 611.
   b. Configure the server to verify the digital signature that is used in the message request. To configure the server, you must specify which parts of the SOAP message, sent by the request sender, contain digitally signed information and which method was used to digitally sign the message. The settings chosen for the request receiver, or the server in this step, must match the settings chosen for the request sender in the previous step.
      1) Define the message parts by following the steps found in "Configuring the server for request digital signature verification: Verifying the message parts" on page 613.
      2) Select the same method used by the request sender to digitally sign the message. You can select the digital signature method by following the steps in "Configuring the server for request digital signature verification: choosing the verification method" on page 614
   c. Configure the server to digitally sign a message response. To configure the server, complete the following steps to specify which parts of the SOAP message to digitally sign and define the method used to digitally sign the message. The sender in these steps is the response sender.
      1) Specify which message parts to digitally sign by following the steps found in "Configuring the server for response signing: digitally signing message parts" on page 617.
      2) Select the method used to digitally sign the response message. You can select the digital signature method by following the steps in "Configuring the server for response signing: choosing the digital signature method" on page 619
   d. Configure the client to verify the digital signature that is used in the message response. To configure the client, you must specify which parts of the SOAP message sent by the response sender contain digitally signed information and which method was used to digitally sign the

message. The settings chosen for the response receiver, or client in this step, must match the settings chosen for the response sender in the previous step.

1) Define the message parts by following the steps found in "Configuring the client for response digital signature verification: verifying the message parts" on page 620
2) Select the same method used by the response sender to digitally sign the message. You can select the digital signature method by following the steps in "Configuring the client for response digital signature verification: choosing the verification method" on page 622

2. Define the client security bindings. To configure the client security bindings, complete the steps in either of the following topics:
   - "Configuring the client security bindings using an assembly tool" on page 624
   - Configuring the security bindings on a server acting as a client using the administrative console
3. Define the server security bindings. To configure the server security bindings, complete the steps in either of the following topics:
   - "Configuring the server security bindings using an assembly tool" on page 627
   - Configuring the server security bindings using the administrative console

Results

After completing these steps, you have secured your web services using XML digital signature.

Configuring the client for request signing: digitally signing message parts:

To configure the client for request signing, specify which message parts to digitally sign when configuring the client.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within an assembly tool.

- "Configuring the client security bindings using an assembly tool" on page 624
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which message parts to digitally sign when configuring the client for request signing:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Extension** tab. The Client Deployment Descriptor is displayed.

5. Expand **Request sender configuration** > **Integrity**. *Integrity* refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet.
6. Indicate which parts of the message to sign by clicking **Add** and selecting **body**, **timestamp**, or **SecurityToken**. The following list contains descriptions of the message parts

   body The body is the user data portion of the message.

   timestamp
   The time stamp determines if the message is valid based on the time that the message is sent and then received. If **timestamp** is selected, proceed to the next step and select **Add created time stamp** to add a time stamp to a message.

   SecurityToken
   The security token authenticates the client. If this option is selected, the message is signed.

   You can choose to digitally sign the message using a time stamp if **Add created time stamp** is selected and configured. You can digitally sign the message using a security token if a login configuration authentication method is selected.
7. Optional: Expand the **Add created time stamp** section and select this option if you want a time stamp added to the message. You can specify an expiration time for the time stamp, which helps defend against replay attacks. The lexical representation for duration is the [ISO 8601] extended format *PnYnMnDTnHnMnS*, where:
   - *nY* represents the number of years
   - *nM* represents the number of months
   - *nD* represents the number of days
   - *T* is the date and time separator
   - *nH* represents the number of hours
   - *nM* represents the number of minutes
   - *nS* represents the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

   For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, the format is: `P1Y2M3DT10H30M`. Typically, you configure a message time stamp for about 10 to 30 minutes, for example, 10 minutes is represented as: `P0Y0M0DT0H10M0S`. The *P* character precedes time and date values.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within an assembly tool:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the **Actor URI** field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the **Actor** fields on the client and server match, the request or response is acted upon instead

of being forwarded downstream. The **Actor** fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server **Actor** fields are synchronized.

What to do next

After you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message. See "Configuring the client for request signing: choosing the digital signature method" for more information.

Configuring the client for request signing: choosing the digital signature method:

To configure the client for request signing, specify which message parts to digitally sign when configuring the client.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Security extensions** tab and the **Port binding** tab in the web services client editor within an assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 624
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively. You must specify which parts of the message sent by the client must be digitally signed. See "Configuring the client for request signing: digitally signing message parts" on page 609 for more information.

About this task

Complete the following steps to specify which message parts to digitally sign when configuring the client for request signing:

Procedure

1. Launch an assembly tool. For more information, see the related information on assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Binding** tab. The Client Deployment Descriptor is displayed.
5. Expand **Security request sender binding configuration** > **Signing information**.
6. Select **Edit** to view the signing information and select a digital signature method from the **Signature method algorithm** field. The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following website http://www.w3.org/TR/xmldsig-core.

Table 133. Digital signature methods. The digital signature method information is stored in the client deployment descriptor.

Name	Purpose
Canonicalization method algorithm	Canonicalizes the `<SignedInfo>` element before the information is digested as part of the signature operation.
Digest method algorithm	Applies to the data after transforms are applied, if specified, to yield the `<DigestValue>` element. Signing the `<DigestValue>` element binds the resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Signature method algorithm	Converts the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Signing key name	Represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is found in the key store and is used to sign the request.
Signing key locator	Represents a reference to a key locator implementation class that locates the correct hey store where the alias and the certificate exist.

7. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the **Digest method algorithm** and **Signature method algorithm** drop-down lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the WebSphere administrative console.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the web services client editor within an assembly tool:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the **Actor URI** field.
- Click **Security extensions** > **Request sender configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the web services editor within an assembly tool:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **Actor** fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method is used to digitally sign a message when the client sends a message to a server.

What to do next

After you configure the client to digitally sign the message, you must configure the server to verify the digital signature. See "Configuring the server for request digital signature verification: Verifying the message parts" for more information.

Configuring the server for request digital signature verification: Verifying the message parts:

Configure the server for request digital signature verification by modifying the extensions to indicate which parts of the request to verify.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Extensions** tab and the **Binding Configurations** tab in the web services editor within the assembly tools:

- "Configuring the server security bindings using an assembly tool" on page 627
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and the Web Services Security bindings, respectively. Also, you must specify which parts of the message sent by the client must be digitally signed. See "Configuring the client for request signing: digitally signing message parts" on page 609 to determine which message parts are digitally signed. The message parts specified for the client request sender must match the message parts specified for the server request receiver.

About this task

Complete the following steps to configure the server for request digital signature verification. The steps describe how to modify the extensions to indicate which parts of the request to verify.

Procedure

1. Launch an assembly tool. For more information, see the related information on assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the Extensions tab in the web services editor.
6. Expand the **Request receiver service configuration details** > **Required integrity** section. Required integrity refers to the parts of the message that require digital signature verification. The purpose of digital signature verification is to make sure that the message parts have not been modified while transmitting across the Internet.
7. Indicate parts of the message to verify by clicking **Add**, and selecting one of the following three parts: **body**, **Timestamp**, or **SecurityToken**. You can determine which parts of the message to verify by looking at the web service request sender configuration in the client application. To view the web service request sender configuration information in the web services client editor, click the Security extensions tab and expand **Request sender configuration** > **Integrity**. The following includes a list and description of the message parts:

   Body This is the user data portion of the message.

Timestamp
The time stamp determines if the message is valid based on the time that the message is sent and then received. If **Timestamp** is selected, proceed to the next step to `Add Created Time Stamp` to the message.

SecurityToken
The security token authenticates the client. If **SecurityToken** is selected, the message is signed.

8. Optional: Expand the **Add received time stamp** section. The Add Received Time Stamp value indicates to validate the Add Created Time Stamp option configured by the client. You must select this option if you selected the Add Created Time Stamp on the client. The time stamp ensures message integrity by indicating the timeliness of the request. This option helps defend against replay attacks.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the **Actor URI** field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts are digitally signed and must be verified by the server when the client sends a message to a server.

What to do next

After you specify which message parts contain a digital signature that must be verified by the server, you must configure the server to recognize the digital signature method used to digitally sign the message. See "Configuring the server for request digital signature verification: choosing the verification method" for more information.

Configuring the server for request digital signature verification: choosing the verification method:

To configure the server for request digital signature verification, use an assembly tool to modify the extensions and indicate which digital signature method the server will use during verification.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the Extensions tab and the Binding Configurations tab in the Web Services Editor within the IBM assembly tools:

- "Configuring the server security bindings using an assembly tool" on page 627
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. You must specify which message parts contain digital signature information that must be verified by the server. See "Configuring the server for request digital signature verification: Verifying the message parts" on page 613. The message parts specified for the client request sender must match the message parts specified for the server request receiver. Likewise, the digital signature method chosen for the client must match the digital signature method used by the server.

About this task

Complete the following steps to configure the server for request digital signature verification. The steps describe how to modify the extensions to indicate which digital signature method the server will use during verification.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Binding Configurations** tab.
6. Expand the **Security request receiver binding configuration details** > **Signing information** section.
7. Click **Edit** to edit the signing information. The signing information dialog is displayed, select or enter the following information:
   - Canonicalization method algorithm
   - Digest method algorithm
   - Signature method algorithm
   - Use certificate path reference
   - Trust anchor reference
   - Certificate store reference
   - Trust any certificate

   For more conceptual information on digitally signing SOAP messages, see XML digital signature. The following table describes the purpose for each of these selections. Some of the following definitions are based on the XML-Signature specification, which is located at the following web address: http://www.w3.org/TR/xmldsig-core.

Table 134. Digital signature methods. The digital signature method is part of the binding configuration.

Name	Purpose
Canonicalization method algorithm	Canonicalizes the `<SignedInfo>` element before it is digested as part of the signature operation. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.

Table 134. Digital signature methods (continued). The digital signature method is part of the binding configuration.

Name	Purpose
Digest method algorithm	Applies to the data after transforms are applied, if specified, to yield the `<DigestValue>` element. The signing of the `<DigestValue>` element binds resource content to the signer key. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Signature method algorithm	Converts the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Use certificate path reference or **Trust any certificate**	Validates a certificate or signature sent with a message. When a message is signed, the public key used to sign it is sent with the message. This public key or certificate might not be validated at the receiving end. By selecting **User certificate path reference**, you must configure a trust anchor reference and a certificate store reference to validate the certificate sent with the message. By selecting **Trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference: Trust anchor reference	Refers to a key store that contains trusted, self-signed certificates and certificate authority (CA) certificates. These certificates are trusted certificates that you can use with any applications in your deployment.
Use certificate path reference: Certificate store reference	Contains a collection of X.509 certificates. These certificates are not trusted for all applications in your deployment, but might be used as an intermediary to validate certificates for an application.

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Signature method algorithm and Digest method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations on the client:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the Actor URI field.
- Click **Security extensions** > **Request sender configuration > Details** and indicate the actor information in the Actor field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the Actor field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified the method that the server uses to verify the digital signature in the message parts.

What to do next

After you configure the client for request signing and the server for request digital signature verification, you must configure the server and the client to handle the response. Next, specify the response signing for the server. See "Configuring the server for response signing: digitally signing message parts" for more information.

Configuring the server for response signing: digitally signing message parts:

Use an assembly tool to specify which message parts to digitally sign when configuring the server for response signing.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this topic supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Extensions** tab and the **Binding** configurations tab in the web services editor within the IBM assembly tools:

- "Configuring the server security bindings using an assembly tool" on page 627
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which message parts to digitally sign when configuring the server for response signing:

Procedure

1. Launch an assembly tool. For more information, see the related information on assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application _name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file and click **Open with** > **Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the Web Services Editor within the assembly tool.
6. Expand **Response sender service configuration details** > **Integrity**. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. For more information on digitally signing SOAP messages, see XML digital signature.
7. Indicate the parts of the message to sign by clicking **Add**, and selecting **Body**, **Timestamp**, or **SecurityToken**.

   The following list contains descriptions of the message parts:

   Body The body is the user data portion of the message.

   Timestamp
   The time stamp determines if the message is valid based on the time that the message is sent and then received. If this option if selected, proceed to the next step and click **Add Created Time Stamp**, which indicates that the time stamp is added to the message.

SecurityToken
The security token is used for authentication. If this option is selected, the authentication information is added to the message.

8. Optional: Expand the **Add created time stamp** section. Select this option if you want a time stamp added to the message. You can specify an expiration time for the time stamp, which helps defend against replay attacks. The lexical representation for duration is the ISO 8601 extended format, *PnYnMnDTnHnMnS*, where:
   - *nY* represents the number of years.
   - *nM* represents the number of months.
   - *nD* represents the number of days.
   - *T* is the date and time separator.
   - *nH* represents the number of hours.
   - *nM* represents the number of minutes.
   - *nS* represents the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

   For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, the format is: `P1Y2M3DT10H30M`. Typically, you configure a message time stamp for about 10 to 30 minutes. 10 minutes is represented as: `P0Y0M0DT0H10M0S`. The *P* character precedes time and date values.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the **Actor URI** field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts to digitally sign when the server sends a response to the client.

What to do next

After you specifying which message parts to digitally sign, you must specify which method is used to digitally sign the message. See “Configuring the server for response signing: choosing the digital signature method” on page 619 for more information.

Configuring the server for response signing: choosing the digital signature method:

Use an assembly tool to specify which digital signature method to use when configuring the server for response signing.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the Extensions tab and the **Binding configurations** tab in the web services editor within the IBM assembly tools:

- "Configuring the server security bindings using an assembly tool" on page 627
- "Configuring the server security bindings using the administrative console" on page 1004

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which digital signature method to use when configuring the server for response signing:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application _name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file and click **Open with** > **Web services editor**.
5. Click the **Binding Configurations** tab.
6. Expand **Response sender binding configuration details** > **Signing information**.
7. Click **Edit** to choose a signing method. The signing info dialog is displayed and either select or enter the following information:
   - Canonicalization method algorithm
   - Digest method algorithm
   - Signature method algorithm
   - Signing key name
   - Signing key locator

   The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following address: http://www.w3.org/TR/xmldsig-core.

Table 135. Digital signature methods. Use the methods to configure the server for response signing.

Name	Purpose
Canonicalization method algorithm	Canonicalizes the <SignedInfo> element before the information is digested as part of the signature operation. Use the same algorithm on the client response receiver. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

Table 135. Digital signature methods (continued). Use the methods to configure the server for response signing.

Name	Purpose
Digest method algorithm	Applies to the data after transforms are applied, if specified, to yield the <DigestValue> element. Signing the <DigestValue> element binds resource content to the signer key. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signature method algorithm	Converts the canonicalized <SignedInfo> element into the <SignatureValue> element. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signing key name	Represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is found in the key store and is used to sign the request.
Signing key locator	Represents a reference to a key locator implementation class that locates the correct key store where the alias and certificate exists. For more information on configuring key locators, see the following file: • "Configuring key locators using an assembly tool" on page 606

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Signature method algorithm and Digest method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

You have specified which method is used to digitally sign a message when the server sends a message to a client.

What to do next

After you configure the server to digitally sign the response message, you must configure the client to verify the digital signature contained in the response message. See "Configuring the client for response digital signature verification: verifying the message parts" for more information.

Configuring the client for response digital signature verification: verifying the message parts:

To configure the Web Services Security extensions and the Web Services Security bindings, use the **WS Extension** tab and the **WS Binding** tab in the Client Deployment Descriptor within an assembly tool.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to becomes familiar with the **WS Extension** tab and the **WS Binding** tab in the Client Deployment Descriptor within the assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 624
- Configuring the security bindings on a server acting as a client using the administrative console

You can use these two tabs to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to configure the client for response digital signature verification. The steps describe how to modify the extensions to indicate which parts of the response to verify.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file and click **Open With** > **Deployment descriptor editor**.
5. Click the **WS extension** tab.
6. Expand the **Response receiver configuration** > **Required integrity** section. Required integrity refers to parts that require digital signature verification. Digital signature verification decreases the risk that the message parts have been modified while the message is transmitted across the Internet.
7. Indicate the parts of the message that must be verified. You can determine which parts of the message to verify by looking at the web service response sender configuration. Click **Add** and select one of the following parts:

   Body The body is the user data portion of the message.

   Timestamp
   The time stamp determines if the message is valid based on the time that the message is sent and then received. If the timestamp option is selected, proceed to the next step to add a received time stamp to the message.

   Securitytoken
   The security token authenticates the client. If the Securitytoken option is selected, the message is signed.
8. Optional: Expand the **Add received time stamp** section. Select **Add received time stamp** to add the received time stamp to the message.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the web services client editor within an assembly tool:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the Actor URI field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the Actor field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the web services editor within an assembly tool:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the Actor field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure

that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts are digitally signed and must be verified by the client when the server sends a response message to the client.

What to do next

After you specify which message parts contain a digital signature that must be verified by the client, you must configure the client to recognize the digital signature method used to digitally sign the message. See “Configuring the client for response digital signature verification: choosing the verification method” for more information.

Configuring the client for response digital signature verification: choosing the verification method:

You can configure the Web Services Security extensions and Web Services Security bindings using the **WS extension** tab and the **WS binding** tab in the web services editor within an assembly tool.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS extension** tab and the **WS binding** tab in the web services editor within the IBM assembly tools:
- “Configuring the server security bindings using an assembly tool” on page 627
- “Configuring the server security bindings using the administrative console” on page 1004

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. Also, you must specify which message parts contain digital signature information that must be verified by the client. See “Configuring the client for response digital signature verification: verifying the message parts” on page 620 to specify which message parts are digitally signed by the server and must be verified by the client. The message parts specified for the server response sender must match the message parts specified for the client response receiver. Likewise, the digital signature method chosen for the server must match the digital signature method used by the client.

About this task

Complete the following steps to configure the client for response digital signature verification. The steps describe how to modify the extensions to indicate which digital signature method the client will use during verification.

Procedure
1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.

5. Click the **WS Binding** tab.
6. Expand the **Security response receiver binding configuration** > **Signing information** section.
7. Click **Edit** to select a digital signature method. The signing info dialog displays and either select or enter the following information:
   - **Canonicalization method algorithm**
   - **Digest method algorithm**
   - **Signature method algorithm**
   - **Signing key name**
   - **Signing key locator**

   For more conceptual information on digitally signing SOAP messages, see XML digital signature. The following table describes the purpose for each of these selections. Some of the following definitions are based on the XML-Signature specification, which can be found at: http://www.w3.org/TR/xmldsig-core.

Table 136. Digital signature methods. Use the methods to configure the client for response digital signature verification.

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the `<SignedInfo>` element before it is digested as part of the signature operation.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>`. The signing of the `<DigestValue>` binds resource content to the signer key. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.
Use certificate path reference or **Trust any certificate**	When a message is signed, the public key used to sign it is transmitted with the message. To validate this public key at the receiving end, configure a certificate path reference. By selecting **User certificate path reference,** you must configure a trust anchor reference and certificate store reference to validate the certificate sent with the message. By selecting **Trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference: Trust anchor reference	A trust anchor is a configuration that refers to a keystore that contains trusted, self-signed certificates and certificate authority (CA) certificates. These certificates are trusted certificates that you can use with any applications in your deployment.
Use certificate path reference: Certificate store reference	A certificate store is a configuration that has a collection of X.509 certificates. These certificates are not trusted for all applications in your deployment, but might be used as an intermediary to validate certificates for an application.

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Signature method algorithm and Digest method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the web services client editor within an assembly tool:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the Actor URI field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the Actor field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the web services editor within an assembly tool:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method the client uses to verify the digital signature in the message parts.

What to do next

After you configure the server for response signing and the client for request digital signature verification, verify that you have configured the client and the server to handle the message request.

Configuring the client security bindings using an assembly tool:

Use the web services client editor within an assembly tool to include the binding information, that describes how to run the security specifications found in the extensions, in the client enterprise archive (EAR) file.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

When configuring a client for Web Services Security, the bindings describe how to run the security specifications found in the extensions. Use the web services client editor within an assembly tool to include the binding information in the client enterprise archive (EAR) file.

You can configure the client-side bindings from a pure client accessing a web service or from a web service accessing a downstream web service. This document focuses on the pure client situation. However, the concepts, and in most cases the steps, also apply when a web service is configured to communicate downstream to another web service that has client bindings. Complete the following steps to edit the security bindings on a pure client (or server acting as a client) using an assembly tool:

Procedure

1. Import the web services client EAR file into an assembly tool. When you edit the client bindings on a server acting as a client, the same basic steps apply. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.

4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**. The Client Deployment Descriptor is displayed.
5. Click the **WS Extension** tab and select the port QName bindings that you want to configure. The Web Services Security extensions are configured for outbound requests and inbound responses. You need to configure the following information for Web Services Security extensions. These topics are discussed in more detail in other sections of the documentation.

   Request sender configuration details

   Details
   "Configuring the client for request signing: digitally signing message parts" on page 609

   Integrity
   "Configuring the client for request signing: digitally signing message parts" on page 609

   Confidentiality
   "Configuring the client for request encryption: Encrypting the message parts" on page 631

   Login Config

   BasicAuth
   "Configuring the client for basic authentication: specifying the method" on page 642

   IDAssertion
   "Configuring the client for identity assertion: specifying the method" on page 650

   Signature
   "Configuring the client for signature authentication: specifying the method" on page 657

   LTPA
   "Configuring the client for LTPA token authentication: specifying LTPA token authentication" on page 666

   ID assertion
   "Configuring the client for identity assertion: specifying the method" on page 650

   Add created time stamp
   "Configuring the client for request signing: digitally signing message parts" on page 609

   Response receiver configuration details

   Required integrity
   "Configuring the client for response digital signature verification: verifying the message parts" on page 620

   Required confidentiality
   "Configuring the client for response decryption: decrypting the message parts" on page 638

   Add received time stamp
   "Configuring the client for response digital signature verification: verifying the message parts" on page 620
6. Click the **WS binding** tab and select the port qualified name binding that you want to configure. The Web Services Security bindings are configured for outbound requests and inbound responses. You need to configure the following information for Web Services Security bindings. These topics are discussed in more details in other sections of the documentation.

   Security request sender binding configuration

   Signing information
   "Configuring the client for request signing: choosing the digital signature method" on page 611

   Encryption information
   "Configuring the client for request encryption: choosing the encryption method" on page 632

   Key locators
   "Configuring key locators using an assembly tool" on page 606

Login binding

BasicAuth
"Configuring the client for basic authentication: collecting the authentication information" on page 644

ID assertion
"Configuring the client for identity assertion: collecting the authentication method" on page 651

Signature
"Configuring the client for signature authentication: collecting the authentication information" on page 658

LTPA "Configuring the client for LTPA token authentication: collecting the authentication method information" on page 667

Security response receiver binding configuration

Signing information
"Configuring the client for response digital signature verification: choosing the verification method" on page 622

Encryption information
"Configuring the client for response decryption: choosing a decryption method" on page 639

Trust anchor
"Configuring trust anchors using an assembly tool" on page 602

Certificate store list
"Configuring the client-side collection certificate store using an assembly tool" on page 604

Key locators
"Configuring key locators using an assembly tool" on page 606

What to do next

Important: When configuring the security request sender binding configuration, you must synchronize the information used to perform the specified security with the security request receiver binding configuration, which is configured in the server EAR file. These two configurations must be synchronized in all respects because there is no negotiation during run time to determine the requirements of the server.

For example, when configuring the encryption information in the security request sender binding Configuration, you must use the public key from the server for encryption. Therefore, the key locator that you choose must contain the public key from the server configuration. The server must contain the private key to decrypt the message. This example illustrates the important relationship between the client and server configuration. Additionally, when configuring the security response receiver binding configuration, the server must send the response using security information known by this client security response receiver binding configuration.

The following table shows the related configurations between the client and the server. The client request sender and the server request receiver are relative configurations that must be synchronized with each other. The server response sender and the client response receiver are related configurations that must be synchronized with each other. Note that the related configurations are end points for any request or response. One end point must communicate its actions with the other end point because run time requirements are not negotiated.

Table 137. Related configurations. The configurations must be synchronized with each other.

Client configuration	Server configuration
Request sender	Request receiver

Table 137. Related configurations (continued). The configurations must be synchronized with each other.

Client configuration	Server configuration
Response receiver	Response sender

Configuring the server security bindings using an assembly tool:

Use an assembly tool to edit bindings for a web service after these bindings are deployed on a server.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to importing the web services enterprise archive (EAR) file into the assembly tool, make sure that you have already run the **WSDL2Java** command on your web service to enable your Java Platform, Enterprise Edition (Java EE) application. You must import the Web services EAR file into the assembly tool.

About this task

Create an Enterprise JavaBeans (EJB) file Java archive (JAR) file or a web application archive (WAR) file containing the security binding file (`ibm-webservices-bnd.xmi`) and the security extension file (`ibm-webservices-ext.xmi`). If this archive is acting as a client to a downstream service, you also need the client-side binding file (`ibm-webservicesclient-bnd.xmi`) and the client-side extension file (`ibm-webservicesclient-ext.xmi`). These files are generated using the **WSDL2Java** command. For more information, read about the **WSDL2Java** command for JAX-RPC applications. You can edit these files using the web services editor in the assembly tool.

When configuring server-side security for Web Services Security, the security extensions configuration specifies what security is performed, the security bindings configuration indicates how to perform what is specified in the security extensions configuration. You can use the defaults for some elements at the cell and server levels in the bindings configuration, including key locators, trust anchors, the collection certificate store, trusted ID evaluators, and login mappings and reference these elements from the WAR and JAR binding configurations.

Open the web services editor in an assembly tool to begin editing the server security extensions and bindings. The following steps can locate the server security extensions and bindings. Other tasks specify how to configure each section of the extensions and bindings in more detail.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java EE perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Configure the server for inbound requests and outbound responses security configuration. To configure the server for inbound requests and outbound responses, complete the following steps:
   a. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
   b. Right-click the `webservices.xml` file and click **Open with** > **Web services editor**. The `webservices.xml` file represents the server-side (inbound) web services configuration. The `webservicesclient.xml` file represents the client-side (outbound) web services configuration.
4. In the web services editor (for the `webservices.xml` file and inbound requests and outbound responses web services configuration), there are several tabs at the bottom of the editor including Web Services,

Port Components, Handlers, Security Extensions, Bindings, and Binding Configurations. The security extensions are edited using the **Security Extensions** tab. The security bindings are edited using the Security Bindings tab.

a. Click the **WS Extensions** tab and select the port component binding to edit. The Web Services Security extensions are configured for inbound requests and outbound responses. You need to configure the following information for Web Services Security extensions. These topics are discussed in more detail in other topics in the documentation.

   Request receiver service configuration details

   Required integrity
   "Configuring the server for request digital signature verification: Verifying the message parts" on page 613

   Required confidentiality
   "Configuring the server for request decryption: decrypting the message parts" on page 633

   Login config

   BasicAuth
   "Configuring the server to handle basic authentication information" on page 647

   ID assertion
   "Configuring the server to handle identity assertion authentication" on page 652

   Signature
   "Configuring the server to support signature authentication" on page 660

   LTPA "Configuring the server to handle LTPA token authentication information" on page 668

   Add received time stamp
   "Configuring the server for request digital signature verification: Verifying the message parts" on page 613

   Response sender service configuration details

   Details
   "Configuring the server for response signing: digitally signing message parts" on page 617

   Integrity
   "Configuring the server for response signing: digitally signing message parts" on page 617

   Confidentiality
   "Configuring the server for response encryption: encrypting the message parts" on page 636

   Add created time stamp
   "Configuring the server for response signing: digitally signing message parts" on page 617

b. Click the **Binding Configurations** tab and select the port component binding to edit. The Web Services Security bindings are configured for inbound requests and outbound responses. You need to configure the following information for Web Services Security bindings. These topics are discussed in more details in other topics in the documentation.

   Response receiver binding configuration details

   Signing Information
   "Configuring the server for request digital signature verification: choosing the verification method" on page 614

   Encryption Information
   "Configuring the server for request decryption: choosing the decryption method" on page 634

Trust Anchor
"Configuring trust anchors using an assembly tool" on page 602

Certificate Store List
"Configuring the server-side collection certificate store using an assembly tool" on page 605

Key Locators
"Configuring key locators using an assembly tool" on page 606

Login Mapping

Basic auth
"Configuring the server to validate basic authentication information" on page 648

ID assertion
"Configuring the server to validate identity assertion authentication information" on page 654

Signature
"Configuring the server to validate signature authentication information" on page 661

LTPA "Configuring the server to validate LTPA token authentication information" on page 669

Trusted ID evaluator

Trusted ID evaluator reference

Response sender binding configuration details

Signing information
"Configuring the server for response signing: choosing the digital signature method" on page 619

Encryption information
"Configuring the server for response encryption: choosing the encryption method" on page 637

Key locators
"Configuring key locators using an assembly tool" on page 606

What to do next

Configure the client for outbound requests and inbound responses security configuration by right-clicking the `webservicesclient.xml` file and clicking **Open With** > **Deployment descriptor editor**. For more information, see "Configuring the client security bindings using an assembly tool" on page 624.

Configuring XML encryption for Version 5.x web services with an assembly tool

XML encryption is one method that WebSphere® Application Server provides to secure your web services. It enables you to encrypt an XML element, the content of an XML element, or arbitrary data such as an XML document.

Securing web services for Version 5.x applications using XML encryption:

XML encryption is one method that WebSphere Application Server provides to secure your web services. It enables you to encrypt an XML element, the content of an XML element, or arbitrary data such as an XML document.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services. XML encryption is one of these methods. You can secure your web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

About this task

XML encryption enables you to encrypt an XML element, the content of an XML element, or arbitrary data such as an XML document. Like XML digital signature, a message is sent by the client as the request sender to the server as the request receiver. The response is sent by the server as the response sender to the client as the request receiver. Unlike XML digital signature, which verifies the authenticity of the sender, XML encryption scrambles the message content using a key, which can be unscrambled by a receiver that possesses the same key. You can use XML encryption in conjunction with XML digital signature to scramble the content while verifying the authenticity of the message sender.

To use XML encryption to secure web services, you must use an assembly tool. For more information, see the related information on Assembly Tools.

To securing web services for Version 5.*x* applications using XML encryption, complete the following steps:

Procedure

1. Specify the encryption settings for the request sender. The message parts and the encryption method settings chosen for the request sender on the client must match the message parts and the method settings chosen for the request receiver on the server. To specify the encryption settings for the request sender:
   a. "Configuring the client for request encryption: Encrypting the message parts" on page 631.
   b. "Configuring the client for request encryption: choosing the encryption method" on page 632.
2. Specify the encryption settings for the request receiver. The decryption settings chosen for the request receiver must match the encryption settings chosen for the request sender.

   To specify the decryption settings for the request receiver:
   a. "Configuring the server for request decryption: decrypting the message parts" on page 633.
   b. "Configuring the server for request decryption: choosing the decryption method" on page 634.
3. Specify the encryption settings for the response sender. The message parts and the encryption method settings chosen for the response sender on the server must match the message parts and the method settings chosen for the response receiver on the client. To specify the encryption settings for the response sender:
   a. "Configuring the server for response encryption: encrypting the message parts" on page 636.
   b. "Configuring the server for response encryption: choosing the encryption method" on page 637.
4. Specify the encryption settings for the response receiver.

Remember: The decryption settings chosen for the response receiver must match the encryption settings chosen for the response sender.

To specify the decryption settings for the response receiver, complete the following steps:

a. “Configuring the client for response decryption: decrypting the message parts” on page 638.
b. “Configuring the client for response decryption: choosing a decryption method” on page 639.

Results

After completing these steps, you have secured your web services using XML encryption.

Configuring the client for request encryption: Encrypting the message parts:

To configure the client for request encryption, specify which message parts to encrypt when configuring the client.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to familiarize yourself with the **WS Extensions** tab and the **WS Binding** tab in the Client Deployment Descriptor Editor within an assembly tool:

- “Configuring the client security bindings using an assembly tool” on page 624
- “Configuring the security bindings on a server acting as a client using the administrative console” on page 1002

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which message parts to encrypt when configuring the client for request encryption:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS extensions** tab, which is located at the bottom of Client Deployment Descriptor Editor within the assembly tool.
6. Expand **Request sender configuration** > **Confidentiality**. Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. For more information on encrypting , see XML encryption.
7. Select the parts of the message that you want to encrypt by clicking **Add**. You can select one of the following parts:

Bodycontent
User data portion of the message

Usernametoken
Basic authentication information, if selected

What to do next

After you specify which message parts to encrypt, you must specify which method to use to encrypt the request message. See "Configuring the client for request encryption: choosing the encryption method" for more information.

Configuring the client for request encryption: choosing the encryption method:

To configure the client for request encryption, specify which encryption method to use when configuring the client.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to familiarize yourself with the **WS Extensions** tab and the **WS Binding** tab in the Client Deployment Descriptor editor within an assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 624
- "Configuring the security bindings on a server acting as a client using the administrative console" on page 1002

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which encryption method to use when configuring the client for request encryption:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS binding** tab, which is located at the bottom of the Client Deployment Descriptor editor within the assembly tool.
6. Expand **Security request sender binding configuration** > **Encryption information**.
7. Select an encryption option and click **Edit** to view the encryption information or click **Add** to add another option. The following table describes the purpose of this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following web address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   Refers to the name of the encryption information entry.

Data encryption method algorithm
Encrypts and decrypts data in fixed size, multiple octet blocks.

Key encryption method algorithm
Represents public key encryption algorithms that are specified for encrypting and decrypting keys.

Encryption key name
Represents a Subject (**Owner** field of the certificate) from a public key certificate found by the encryption key locator, which is used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data.

The key chosen must be a public key of the target. Encryption must be done using the public key and decryption must be done by the target using the private key (the personal certificate of the target).

Encryption key locator
Represents a reference to a key locator implementation class that locates the correct key store where the alias and the certificate exist. For more information on configuring key locators, see "Configuring key locators using an assembly tool" on page 606 and Configuring key locators using the administrative console.

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the **Data Encryption method algorithm** and **Key Encryption method algorithm** dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the WebSphere administrative console.

Results

For more information, see "Configuring key locators using an assembly tool" on page 606 and Configuring key locators using the administrative console.

What to do next

You must specify which parts of the request message to encrypt. See "Configuring the client for request encryption: Encrypting the message parts" on page 631 if you have not previously specified this information.

Configuring the server for request decryption: decrypting the message parts:

Use the **WS Extensions** tab and the **WS Binding** configurations tab to specify which parts of the request message must be decrypted by the server.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.x and later applications.

Complete this task to specify which parts of the request message must be decrypted by the server. You must know which parts of the request message the client encrypts because the server must decrypt the same message parts.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Binding** configurations tab:

- "Configuring the server security bindings using an assembly tool" on page 627

- “Configuring the server security bindings using the administrative console” on page 1004

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to configure the request receiver extensions:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META_INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver service configuration details** > **Required confidentiality** section.
7. Select the parts of the message to decrypt. The message parts selected for the request decryption on the server must match the message parts selected for the message encryption on the client. Click **Add** and select either of the following message parts:

   bodycontent
      The user data section of the message.

   usernametoken
      This token is the basic authentication information.

What to do next

After you specify which parts of the request message to decrypt, you must specify the method to use decrypt the message. See “Configuring the server for request decryption: choosing the decryption method” for more information.

Configuring the server for request decryption: choosing the decryption method:

You can use an assembly tool and the administrative console to configure the Web Services Security extensions and Web Services Security bindings.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Bindings** tab:

- “Configuring the server security bindings using an assembly tool” on page 627
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete this task to specify which decryption method is used by the server to decrypt the request message. You must know which decryption method the client uses because the server must use the same method.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META_INF**.
4. Right-click the `webservices.xml` file, select **Open with** > **Web services editor**.
5. Click the **Binding Configurations** tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver binding configuration details** > **Encryption information** section.
7. Click **Edit** to view the encryption information. The following table describes the purpose for each of these selections. Some definitions are taken from the XML-Encryption specification , which is located at the following web address: http://www.w3.org/TR/xmlenc-core

   Encryption name
      Represents the name of this encryption information entry; an alias for the entry.

   Data encryption method algorithm
      Encrypts and decrypts data in fixed size, multiple octet blocks. This algorithm must be the same as the algorithm selected in the client request sender configuration.

   Key encryption method algorithm
      Represents algorithms specified for encrypting and decrypting keys. This algorithm must be the same as the algorithm selected in the client request sender configuration.

   Encryption key name
      Represents a Subject from a personal certificate, which is typically a distinguished name (DN) that is found by the encryption key locator. The subject is used by the key encryption method algorithm to decrypt the secret key, and the secret key is used to decrypt the data.

      The key chosen must be a private key in the key store configured by the key locator. The key requires the same Subject used by the client to encrypt the data. Encryption must be done using the public key and decryption by using the private key (personal certificate). To ensure that the client encrypts the data with the correct public or private key, you must extract the public key from the server key store and add it to the key store specified in the encryption configuration information for the client request sender.

      For example, the personal certificate of a server is `CN=Bob, O=IBM, C=US`. Therefore the server contains the public and private key pair. The client sending the request should encrypt the data using the public key for `CN=Bob, O=IBM, C=US`. The server decrypts the data using the private key for `CN=Bob, O=IBM, C=US`.

   Encryption key locator
      Represents a reference to a key locator implementation class that finds the correct keystore where the alias and the certificate exist. For more information on configuring key locators, go to the following sections: "Configuring key locators using an assembly tool" on page 606 and Configuring key locators using the administrative console.
8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Data Encryption method algorithm and Key Encryption method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application

Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

It is important to note that for decryption, the encryption key name chosen must refer to a personal certificate that can be located by the key locator of the server referenced in the encryption information. Enter the Subject of the personal certificate here, which is typically a Distinguished Name (DN). The Subject uses the default key locator to find the key. If a custom key locator is written, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that finds the correct key store where this alias and certificate exist. Refer to "Configuring key locators using an assembly tool" on page 606 and Configuring key locators using the administrative console for more information.

What to do next

You must specify which parts of the request message to decrypt. See "Configuring the server for request decryption: decrypting the message parts" on page 633 if you have not previously specified this information.

Configuring the server for response encryption: encrypting the message parts:

You can specify which parts of the response message to encrypt when configuring the server for response encryption.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Bindings** tab in the Web services editor within an assembly tool:

- "Configuring the server security bindings using an assembly tool" on page 627
- "Configuring the server security bindings using the administrative console" on page 1004

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which parts of the response message to encrypt when configuring the server for response encryption:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window > Open Perspective > J2EE**.
3. Click **EJB Projects >** ***application_name*** **> ejbModule > META_INF**.
4. Right-click the `webservices.xml` file, select **Open with > Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the Web Services Editor within the assembly tool.

6. Expand **Response sender service configuration details > Confidentiality**. Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the response is encrypted before it is sent and decrypted when it is received at the correct target.
7. Select the parts of the response that you want to encrypt by clicking **Add** and selecting **Bodytoken** or **Usernametoken**. The following information describes the message parts:

   Bodycontent
   User data portion of the message.

   Usernametoken
   Basic authentication information, if selected.

   A user name token does not appear in the response so you do not need to select this option for the response. If you select this option, make sure that you also select it for the client response receiver. If you do not select this option, make sure that you do not select it for the client response receiver.

What to do next

After you specify which message parts to encrypt, you must specify which method to use message encryption. See the task for choosing the encryption method when configuring the server for response encryption.

Configuring the server for response encryption: choosing the encryption method:

You can specify which method the server uses to encrypt the response message.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Extensions** tab and the **Binding configurations** tab in the web services editor within an assembly tool:
- "Configuring the server security bindings using an assembly tool" on page 627
- "Configuring the server security bindings using the administrative console" on page 1004

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which method the server uses to encrypt the response message:

Procedure
1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META_INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Binding Configuration**s tab, which is located at the bottom of the Web Services Editor within the assembly tool.

6. Expand **Response sender binding configuration details** > **Encryption information**.
7. Click **Edit** to view the encryption information. The following table describes the purpose of this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following web address: http://www.w3.org/TR/xmlenc-core

   Encryption name
      Refers to the name of the encryption information entry.

   Data encryption method algorithm
      Encrypts and decrypts data in fixed size, multiple octet blocks. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

   Key encryption method algorithm
      Represents public key encryption algorithms that are specified for encrypting and decrypting keys. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

   Encryption key name
      Represents a Subject from a public key certificate typically distinguished name (DN) that is found by the encryption key locator and used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data.

      The key name chosen in the server response sender encryption information must be the public key of the key configured in the client response receiver encryption information. Encryption by the response sender must be done using the public key and decryption must be done by the response receiver using the associated private key (the personal certificate of the response receiver).

   Encryption key locator
      The encryption key locator represents a reference to a key locator implementation class that finds the correct key store where the alias and the certificate exist. For more information, see the tasks for configuring key locators.
8. Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Data Encryption method algorithm and Key Encryption method algorithm drop-down lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

The encryption key name chosen must refer to a public key of the response receiver. For the encryption key name, use the Subject of the public key certificate, typically a Distinguished Name (DN). The name chosen is used by the default key locator to find the key. If you write a custom key locator , the encryption key name might be anything that is used by the key locator to find the correct encryption key (a public key). The encryption key locator references the implementation class that finds the correct key store where the alias and certificate exist.

What to do next

You must specify which parts of the response message to encrypt. See the task for configuring the server for response encryption if you have not previously specified this information.

Configuring the client for response decryption: decrypting the message parts:

To configure the client for response decryption, specify which response message parts to decrypt when configuring the client. The server response encryption and client response decryption configurations must match.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Binding** tab in the Client Deployment Descriptor Editor within an assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 624
- "Configuring the security bindings on a server acting as a client using the administrative console" on page 1002

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which response message parts to decrypt when configuring the client for response decryption. The server response encryption and client response decryption configurations must match.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Extensions** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Response receiver configuration** > **Required confidentiality** section.
7. Select the parts of the message that you must decrypt by clicking **Add** and selecting either **Bodycontent** or **Usernametoken**. The following information describes these message parts:

   Bodycontent
   The user data portion of the message.

   Usernametoken
   The basic authentication information, if selected.

   The information selected in this step is encrypted by the server in the response sender.

   Important: A Username Token is typically not sent in the response. Thus, you usually do not need to select username token.

What to do next

After you specify which message parts to decrypt, you must specify which method to use when decrypting the response message. See "Configuring the client for response decryption: choosing a decryption method" for more information.

Configuring the client for response decryption: choosing a decryption method:

To configure the client for response decryption, specify which decryption method to use when the client decrypts the response message. The server response encryption and client response decryption configurations must match.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Bindings** tab in the Client Deployment Descriptor Editor within an assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 624
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which decryption method to use when the client decrypts the response message. The server response encryption and client response decryption configurations must match.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Binding** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Security response receiver binding configuration** > **Encryption information** section. For more information on encrypting and decrypting SOAP messages, see XML encryption.
7. Click **Edit** to view the encryption information. The following table describes the purpose for this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following web address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   Refers to the alias that is used for the encryption information entry.

   Data encryption method algorithm
   Encrypts and decrypts data in fixed size, multiple octet blocks.

   Key encryption method algorithm
   Represents public key encryption algorithms specified for encrypting and decrypting keys.

   Encryption key name
   Represents a Subject from a personal certificate, which is typically a distinguished name (DN) that is found by the encryption key locator. The Subject is used by the key encryption method algorithm to decrypt the secret key. The secret key is used to decrypt the data.

   Important: The key chosen must be a private key of the client. Encryption must be done using the public key and decryption must be done by the private key (personal

certificate). For example, the personal certificate of the client is: `CN=Alice, O=IBM, C=US`. Therefore, the client contains the public and private key pair. The target server that sends the response encrypts the secret key by using the public key for `CN=Alice, O=IBM, C=US`. The client decrypts the secret key by using the private key for `CN=Alice, O=IBM, C=US`.

Encryption key locator
: Represents a reference to a key locator implementation class that finds the correct key store where the alias and the certificate exist. For more information on configuring key locators, see "Configuring key locators using an assembly tool" on page 606 and Configuring key locators using the administrative console.

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Data Encryption method algorithm and Key Encryption method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

For decryption, the encryption key name chosen must refer to a personal certificate that can be located by the client key locator. The Subject (**owner** field of the certificate) of the personal certificate should be entered in the Encryption key name, this is typically a Distinguished Name (DN). The default key locator uses the Encryption key name to find the key within the keystore. If you write a custom key locator, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that locates the correct key store where this alias and certificate exists. For more information, see "Configuring key locators using an assembly tool" on page 606 and Configuring key locators using the administrative console.

What to do next

You must specify which parts of the request message to decrypt. See the topic"Configuring the client for response decryption: decrypting the message parts" on page 638 if you have not previously specified this information.

Configuring XML basic authentication for Version 5.x web services with an assembly tool

With the basic authentication (BasicAuth) authentication method, the request sender generates a BasicAuth security token using a callback handler. The request receiver retrieves the BasicAuth security token from the SOAP message and validates it using a Java™ Authentication and Authorization Service (JAAS) login module. Trust is established by using user name and password validation.

Securing web services for Version 5.x applications using basic authentication:

With the basic authentication (BasicAuth) authentication method, the request sender generates a BasicAuth security token using a callback handler. The request receiver retrieves the BasicAuth security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. Trust is established by using user name and password validation.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services. BasicAuth authentication is one of these methods. You might also secure your web services using any of the following methods:

- XML digital signature
- XML encryption
- BasicAuth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

To use BasicAuth authentication to secure web services, complete the following tasks:

Procedure

1. Secure the client for BasicAuth authentication.
   a. Configure the client for basic authentication: specifying the method
   b. Configure the client for basic authentication: collecting the authentication information
2. Secure the server for BasicAuth authentication.
   a. Configure the server to handle basic authentication
   b. Configure the server to validate basic authentication information

Results

After completing these steps, you have secured your web services using BasicAuth authentication.

Configuring the client for basic authentication: specifying the method:

Basic authentication (BasicAuth) refers to the user ID and password of a valid user in the registry of the target server. BasicAuth information can be collected in many ways, including through an administrative console prompt, a standard in (Stdin) prompt, or specified in the bindings that prevents user interaction.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

For more information on BasicAuth authentication, see: "BasicAuth authentication method" on page 643.

About this task

Attention: WebSphere Application Server supports nonce (randomly generated token) with BasicAuth authentication. For more information, see Nonce.

Complete the following steps to specify BasicAuth as the authentication method:

Procedure

1. Launch an assembly tool. See more information on the assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.

5. Click the **WS Extensions** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Request sender configuration** > **Login configuration** section. The only valid login configuration choices for a pure client are **BasicAuth** and **Signature**.
7. Select **BasicAuth** to authenticate the client using a user ID and a password. This user ID and password must be specified in the target user registry. The other choice, Signature, attempts to authenticate the client using the certificate used to digitally sign the message.

What to do next

For more information on getting started with the web services client editor within the assembly tool, see either of the following topics:
- "Configuring the client security bindings using an assembly tool" on page 624
- "Configuring the security bindings on a server acting as a client using the administrative console" on page 1002

After you specify the BasicAuth authentication method, you must specify how to collect the authentication information. See "Configuring the client for basic authentication: collecting the authentication information" on page 644.

BasicAuth authentication method:

When you use the BasicAuth authentication method, the security token that is generated is a <wsse:UsernameToken> element with <wsse:Username> and <wsse:Password> elements.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

WebSphere Application Server supports text passwords but not password digest because passwords are not stored and cannot be retrieved from the server. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following sections.

BasicAuth token generation

The request sender generates a BasicAuth security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler that is used is specified in the <LoginBinding> element of the bindings file, `ibm-webservicesclient-bnd.xmi`. The following callback handler implementations are provided with WebSphere Application Server and can be used with the BasicAuth authentication method:
- com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler

You can add your own callback handlers that implement the `javax.security.auth.callback.CallbackHandler` method.

BasicAuth token validation

The request receiver retrieves the BasicAuth security token from the SOAP message and validates it using a JAAS login module. The <wsse:Username> and <wsse:Password> elements in the security token are used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. This Subject is set as the identity of the running thread. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the <LoginMapping> element of the bindings file. Default bindings are specified in the `ws-security.xml` file. However, you can override these bindings using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName value. The CallbackHandlerFactory option specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl CallbackHandlerFactory implementation. The ConfigName value specifies a JAAS configuration name entry. WebSphere Application Server searches the `security.xml` file for a matching configuration name entry. If a match is not found, it searches the `wsjaas.conf` file for a match. WebSphere Application Server provides the WSLogin default configuration entry, which is suitable for the BasicAuth authentication method.

Configuring the client for basic authentication: collecting the authentication information:

The basic authentication (BasicAuth) method refers to the user ID and the password of a valid user in the registry of the target server. Collection of BasicAuth information can occur in many ways including through a user interface prompt, a standard in (Stdin) prompt, or specified in the bindings, which prevents user interaction.

About this task

Note: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

For more information on BasicAuth authentication, see "BasicAuth authentication method" on page 643.

Complete this task to specify the authentication information needed for BasicAuth authentication:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Binding** tab, which is located at the bottom of deployment descriptor editor within the assembly tool.
6. Expand the **Security request sender binding configuration** > **Login binding** section.
7. Click **Edit** or **Enable** to view the login binding information. The login binding information displays and enter the following information:

   Authentication method
      Specifies the type of authentication. Select **BasicAuth** to use basic authentication.

   Token value type URI and Token value type local name
      When you select **BasicAuth**, you cannot edit the token value type URI and the local name values. Specifies values for custom authentication types. For BasicAuth authentication, leave these values blank.

   Callback handler
      Specifies the Java Authentication and Authorization Server (JAAS) callback handler implementation for collecting the BasicAuth information. You can use the following default implementations for the callback handler:

com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
This implementation is used for non-user interface console prompts.

Restriction: This implementation prompts for the user name and password and reads them into the configuration from standard in. If you have a multi-threaded client and multiple threads attempt to read from standard in at the same time, all the threads will not successfully obtain the user name and password information. Therefore, you cannot use the `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler` implementation with a multi-threaded client where multiple threads might attempt to obtain data from standard in concurrently.

com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
This implementation is used for user interface panel prompts.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This implementation is used when you plan to always enter the user ID and password in the BasicAuth user ID and password section that follows.

Basic Authentication user ID and Basic Authentication password
Specifies values for the BasicAuth user ID and password, regardless of the default callback handler indicated previously, these user ID and password values are used to authenticate to the server for the Web Services Security authentication. If you leave these values blank, use either the `GUIPromptCallbackHandler` or the `StdinPromptCallbackHandler` implementation, but only on a pure client. Always fill-in these values for any web service that acts as a client to another web service that you want to specify for BasicAuth for authentication downstream. If you want the client identity of the originator to flow downstream, configure the web service client to use either ID assertion or Lightweight Third Party Authentication (LTPA).

Property
Specifies properties with name and value pairs for custom callback handlers to use. For BasicAuth authentication, you do not need to enter any information. To enter a new property, click **Add** and enter the new property and value.

Results

Other basic authentication entries: There is a basic authentication entry in the Port Qualified Name Binding Details section. This entry is used for HTTP transport authentication, which might be required if the router servlet is protected.

Information specified in the Web Services Security basic authentication section overrides the basic authentication information specified in the Port Qualified Name Binding Details section for authorizing the web service.

For a server that acts as a client, do not specify a user interface or non-user interface prompt callback handler. To configure BasicAuth authentication from one web service to a downstream web service, select the `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHander` implementation and explicitly specify the BasicAuth user ID and password. If you want the client identity of the originator to flow downstream, configure the web service client to use ID assertion.

What to do next

To use the BasicAuth authentication method, you must specify the method in the Login configuration section of the assembly tool . See "Configuring the client for basic authentication: specifying the method" on page 642 if you have not previously specified this information.

Identity assertion authentication method:

When using the identity assertion (IDAssertion) authentication method, the security token generated is a <wsse:UsernameToken> element that contains a <wsse:Username> element.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* applications.

On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. These two operations, token generation and token validation operations, are described in the following sections.

Identity assertion token validation:

The request receiver retrieves the IDAssertion security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. With identity assertion, special processing is required to establish trust before asserting the identity as the established identity of the running thread. This special processing is defined by the <IDAssertion> element in the deployment descriptor file, `ibm-webservices-ext.xmi`. If all the validation checks are successful, the asserted identity is set as the identity of the running thread. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the<LoginMapping> element of the bindings file. Default bindings are specified in the `ws-security.xml` file. However, you can override these bindings using the application specific `ibm-webservices-bnd.xmi` file. The configuration information consists of CallbackHandlerFactory and a ConfigName. CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl` CallbackHandlerFactory implementation. ConfigName specifies a JAAS configuration name entry.

WebSphere Application Server searches the `security.xml` file for a matching configuration name entry. If a match is not found it searches the `wsjaas.conf` file. WebSphere Application Server provides the system.wssecurity.IDAssertion default configuration entry, which is suitable for the identity assertion authentication method.

The <IDAssertion> element in the `ibm-webservices-ext.xmi` deployment descriptor file specifies the special processing required when using the identity assertion authentication method. The <IDAssertion> element is composed of two sub-elements: <IDType> and <TrustMode>.

The <IDType> element specifies the method for asserting the identity. The supported values for asserting the identity are:

- Username
- Distinguished name (DN)
- X.509 certificate

When <IDType> is *username*, a username token (for example, Bob) is provided. This user name is mapped to a user in the user registry and is the asserted identity after successful trust validation. When the <IDType> value is *DN*, a user name token containing a distinguished name is provided (for example, `cn=Bob Smith, o=ibm, c=us`). This DN is mapped to a user in the user registry and this user is the asserted identity after successful trust validation. When the <IDType> is *X509Certificate*, a binary security token containing an X509 certificate is provided and the SubjectDN value from the certificate (for example, `cn=Bob Smith, o=ibm, c=us`) is extracted. This SubjectDN value is mapped to a user in the user registry and this user is the asserted identity after successful trust validation.

The `<TrustMode>` element specifies how the trust authority, or asserting authority, provides trust information. The supported values are:

- Signature
- BasicAuth
- No value specified

When the <TrustMode> value is `Signature`, the signature is validated. Then, the signer (for example, `cn=IBM Authority, o=ibm, c=us`) is mapped to an identity in the user registry (for example, IBMAuthority). To ensure that the asserting authority is trusted, the mapped identity (for example, IBMAuthority) is validated against a list of trusted identities. When the <TrustMode> element is `BasicAuth`, there is a user name token with a user name and password, which is the user name and password of the asserting authority.

The user name and password are validated. If they are successfully validated, that user name (for example, IBMAuthority) is validated against a list of trusted identities. If a value is not specified for `<TrustMode>`, trust is presumed and additional trust validation is not performed. This type of identity assertion is called *presumed trust mode*. Use the presumed trust mode only in an environment where the trust is established using some other mechanism.

If all the validations described previously succeed, the asserted identity (for example, Bob) is set as the identity of the running thread. If any of the validations fail, the request is rejected with a SOAP fault exception.

Configuring the server to handle basic authentication information:

Basic authentication (BasicAuth) refers to the user ID and the password of a valid user in the registry of the target server. After a request is received that contains basic authentication information, the server needs to log in to form a credential. The credential is used for authorization.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

About this task

Complete the following steps to configure the server to handle BasicAuth authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the web services editor within an assembly tool.
6. Expand the **Request receiver service configuration details** > **Login configuration** section. You can select the following options:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**

- **Lightweight Third Party Authentication (LTPA)**

7. Select **BasicAuth** to authenticate the client with a user ID and a password. The client must specify a valid user ID and password in the server user registry. If the user ID and the password supplied are not valid, an exception is provided, and the request ends without invoking the resource.

   You can select multiple login configurations, which means that different types of security information might be received at the server. The order in which the login configurations are added decides the order in which they are processed when a request is received. Problems can occur if you have multiple login configurations added that have security tokens in common. For example, ID assertion contains a BasicAuth token. For ID assertion to work properly, list ID assertion ahead of BasicAuth in the processing list or the BasicAuth processing overrides the IDAssertion processing.

What to do next

After you specify how the server handles BasicAuth authentication information, you must specify how the server validates the authentication information. See the task for configuring the server to validate BasicAuth authentication if you have not previously specified this information.

Configuring the server to validate basic authentication information:

Basic authentication (BasicAuth) refers to the user ID and the password of a valid user in the registry of the target server. You can specify how the server validates the BasicAuth information.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

After a request is received that contains basic authentication information, the server needs to log in to form a credential. The credential is used for authorization. If the user ID and the password supplied is invalid, an exception is thrown and the request ends without invoking the resource.

About this task

Complete the following steps to specify how the server validates the BasicAuth authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Binding Configurations** tab, which is located at the bottom of the web services editor within an assembly tool.
6. Expand the **Request receiver binding configuration details** > **Login mapping** section.
7. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog is displayed. Select or enter the following information:

   Authentication method

   Specifies the type of authentication that occurs. Select **BasicAuth** to use basic authentication.

Configuration name
: Specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the BasicAuth authentication method, enter `WSLogin` for the JAAS login Configuration name.

Use token valid type
: Determines if you want to specify a custom token type. For the default authentication method selections, you do not need to specify this option.

Token value type URI and Token value type URI local name
: When you select BasicAuth, you cannot edit the token value type URI and local name values. Specifies custom authentication types. For BasicAuth authentication leave these fields blank.

Callback handler factory class name
: Creates a JAAS CallbackHandler implementation that understands the following callbacks:

- javax.security.auth.callback.NameCallback
- javax.security.auth.callback.PasswordCallback
- com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
- com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback
- com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback

Callback handler factory property name and Callback handler factory property value
: Specifies callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For BasicAuth, you do not need to enter any property values.

Login mapping property name and Login mapping property value
: Specifies properties for a custom login mapping. For the default implementations including BasicAuth, leave these fields blank.

What to do next

You must specify how the server handles the BasicAuth authentication method. See the task for configuring the server to handle basic authentication if you have not previously specified this information.

Configuring identity assertion for Version 5.x web services with an assembly tool

With the identity assertion authentication method, the security token generates a <wsse:UsernameToken> element that contains a <wsse:Username> element. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. Trust is established through the use of a security token rather than through user name and password validation.

Securing web services for Version 5.x applications using identity assertion authentication:

With the identity assertion authentication method, the security token generates a <wsse:UsernameToken> element that contains a <wsse:Username> element. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. Unlike BasicAuth authentication, trust is established through the use of a security token rather than through user name and password validation.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services. Identity assertion authentication is one of these methods. You might also secure your web services using any of the following methods:

- XML digital signature
- XML encryption
- BasicAuth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

About this task

To use identity assertion authentication to secure web services, complete the following tasks:

Procedure

1. Secure the client for identity assertion authentication.
   a. "Configuring the client for identity assertion: specifying the method"
   b. "Configuring the client for identity assertion: collecting the authentication method" on page 651
2. Secure the server for identity assertion authentication.
   a. "Configuring the server to handle identity assertion authentication" on page 652
   b. "Configuring the server to validate identity assertion authentication information" on page 654

Results

After completing these steps, you have secured your web services by using identity assertion authentication.

Configuring the client for identity assertion: specifying the method:

You can configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service. Do not attempt to configure identity assertion from a pure client. Identity assertion works only when you configure on the client-side of a web service acting as a client to a downstream web service.

In order for the downstream web service to accept the identity of the originating client (just the user name), you must supply a special trusted BasicAuth credential that the downstream web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream web service configuration. See topics about trusted ID evaluators.

Complete the following steps to specify identity assertion as the authentication method:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.

2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Extension** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Request sender configuration** > **Login configuration** section.
7. Select **IDAssertion** as the authentication method. For more conceptual information on identity assertion authentication, see Identity assertion in a SOAP message.
8. Expand the **IDAssertion** section.
9. For the ID type, select **Username**. This value works with all registry types and originating authentication methods.
10. For the trust mode, select either **BasicAuth** or **Signature**.
    - By selecting **BasicAuth**, you must include basic authentication information (user ID and password), which the downstream web service has specified in the trusted ID evaluator as a trusted user ID. See “Configuring the client for signature authentication: collecting the authentication information” on page 658 to specify the user ID and password information.
    - By selecting **Signature** the certificate configured in the signature information section used to sign the data also is that is used as the trusted subject. The Signature is used to create a credential and user ID, which the certificate mapped to the downstream registry, is used in the trusted ID evaluator as a trusted user ID.

What to do next

See “Configuring the client security bindings using an assembly tool” on page 624 for more information on the web services client editor within the assembly tool.

After you specify identity assertion as the authentication method used by the client, you must specify how to collect the authentication information. See “Configuring the client for identity assertion: collecting the authentication method” for more information.

Configuring the client for identity assertion: collecting the authentication method:

You can configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service. Do not attempt to configure identity assertion from a pure client. Identity assertion works only when you configure on the client-side of a web service acting as a client to a downstream web service.

In order for the downstream web service to accept the identity of the originating client (just the user name), you must supply a special trusted BasicAuth credential that the downstream web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream web service configuration. See the information on trusted ID evaluators.

Complete the following steps to specify how the client collects the authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Binding** tab, which is located at the bottom of the Deployment Descriptor Editor within an assembly tool.
6. Expand the **Security request sender binding configuration** > **Login binding** section.
7. Click **Edit** to view the login binding information and select **IDAssertion**. The login binding dialog is displayed. Select or enter the following information:

   Authentication method
   : The authentication method specifies the type of authentication that occurs. Select **IDAssertion** to use identity assertion.

   Token value type URI and Token value type Local name
   : When you select IDAssertion, you cannot edit the token value type Universal Resource Identifier (URI) and the local name. Specifies custom authentication types. For IDAssertion authentication, leave these values blank.

   Callback handler
   : Specifies the Java Authentication and Authorization Service (JAAS) callback handler implementation for collecting the BasicAuth information. Specify the `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler` implementation for IDAssertion.

   Basic authentication User ID and Basic authentication Password
   : In this field, the trust mode entered in the extensions is BasicAuth. Specifies the trusted user ID and password in these fields. The user ID specified must be an ID that is trusted by the downstream web service. The web service trusts the user ID if it is entered as a trusted ID in a trusted ID evaluator in the downstream web service bindings. If the trust mode entered in the extensions is `Signature`, you do not need to specify any information in this field.

   Property name and Property value
   : Specifies properties with name and value pairs, for use by custom callback handlers. For IDAssertion, you do not need to specify any information in this field.

What to do next

To use the identity assertion authentication method, you must specify the method in the Security extensions section of an assembly tool. See "Configuring the client for identity assertion: specifying the method" on page 650 if you have not previously specified this information.

Configuring the server to handle identity assertion authentication:

The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service. You can configure identity assertion authentication for the server. Do not attempt to configure identity assertion from a pure client.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

For the downstream web service to accept the identity of the originating client (user name only), you must supply a special trusted BasicAuth credential that the downstream web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluator. The server side passes the special BasicAuth credential into the trusted ID evaluator, which returns `true` or `false` that this ID is trusted. After it is trusted, the user name of the client is mapped to the credential, which is used for authorization.

Complete the following steps to configure the server to handle identity assertion authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver service configuration details** > **Login configuration** section. The options you can select are:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**
   - **LTPA** (Lightweight Third Party Authentication)
7. Select **IDAssertion** to authenticate the client using the identity assertion data provided.

   The user ID of the client must be in the target user registry or repository, which is configured on the **Security** > **Global security** panel in the administrative console for WebSphere Application Server. You can select multiple login configurations, which means that different types of security information can be received at the server. The order in which the login configurations are added determines the processing order when a request is received. Problems can occur if you have multiple login configurations added that have common security tokens. For example, ID assertion contains a BasicAuth token, which is the trusted token. For ID assertion to work properly, you must list ID assertion ahead of BasicAuth in the list or BasicAuth processing overrides ID assertion processing.
8. Expand the **IDAssertion** section and select both the **ID Type** and the **Trust Mode**.
   a. For ID Type, the options are:
      - **Username**
      - **DN** (distinguished name)
      - **X509certificate**

      These choices are just preferences and are not guaranteed. Most of the time the Username option is used. You must choose the same ID Type as the client.

   b. For Trust Mode, the options are:
      - **BasicAuth**
      - **Signature**

      The Trust Mode refers to the information sent by the client as the trusted ID.

      1) If you select **BasicAuth**, the client sends basic authentication data (user ID and password). This BasicAuth data is authenticated to the configured user registry. When the authentication occurs successfully, the user ID must be part of the trusted ID evaluator trust list.
      2) If you select **Signature**, the client signing certificate is sent. This certificate must be mappable to the configured user registry. For **Local OS**, the common name (CN) of the distinguished

name (DN) is mapped to a user ID in the registry. For **Lightweight Directory Access Protocol (LDAP)**, the DN is mapped to the registry for the ExactDN mode. If it is in the CertificateFilter mode, attributes are mapped accordingly. In addition, the user name from the credential generated must be in the Trusted ID Evaluator trust list.

What to do next

For more information on getting started with the Web Services Editor within an assembly tool, see "Configuring the server security bindings using an assembly tool" on page 627.

After you specify how the server handles identity assertion authentication information, you must specify how the server validates the authentication information. See the task for configuring the server to validate identity assertion authentication information.

Configuring the server to validate identity assertion authentication information:

The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream Web service.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

Use this task to configure identity assertion authentication. Do not attempt to configure identity assertion from a pure client.

For the downstream web service to accept the identity of the originating client (user name only), you must supply a special trusted BasicAuth credential that the downstream web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream web service configuration. For more information on trusted ID evaluators, see the topic about the trusted ID evaluator. The server side passes the special BasicAuth credential into the trusted ID evaluator, which returns a `true` or `false` response that this ID is trusted. After it is trusted, the user name of the client is mapped to the credential, which is used for authorization.

Complete the following steps to validate the identity assertion authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective > J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the Binding Configurations tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver binding configuration details** > **Login mapping** section.
7. Click **Edit** to view the login mapping information. Click **Add** to add new login mapping information. The login mapping dialog is displayed. Select or enter the following information:

   Authentication method

   Specifies the type of authentication that occurs. Select **IDAssertion** to use basic authentication.

Configuration name
Specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the IDAssertion authentication method, enter `system.wssecurity.IDAssertion` for the Java Authentication and Authorization Service (JAAS) login configuration name.

Use token value type
Determines if you want to specify a custom token type. For the default authentication method selections, you do not need to specify this option.

Token value type URI and Token value type local name
When you select ID assertion, you cannot edit the token value type URI and local name values. Specifies custom authentication types. For the ID assertion authentication method, leave these values blank.

Callback handler factory class name
Ccreates a JAAS CallbackHandler implementation that understands the following callbacks:

- javax.security.auth.callback.NameCallback
- javax.security.auth.callback.PasswordCallback
- com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
- com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback
- com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback

For any of the default authentication methods (BasicAuth, IDAssertion, and Signature), use the callback handler factory default implementation. Enter the following class name for any of the default Authentication methods including IDAssertion:

```
com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl
```

This implementation creates the correct callback handler for the default implementations.

Callback handler factory property name and Callback handler factory property value
Specifies callback handler properties for custom callback handler factory implementations. The default callback handler factory implementation does not need any specified properties. For ID assertion, leave these values blank.

Login mapping property name and Login mapping property value
Specifies properties for a custom login mapping. For the default implementations including IDAssertion, leave these values blank.

8. Expand the **Trusted ID evaluator** section.
9. Click **Edit** to see a dialog that displays all the trusted ID evaluator information. The following table describes the purpose of this information.

Class name
Refers to the implementation of the trusted ID evaluator that you want to use. Enter the default implementation as

```
com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl
```

If you want to implement your own trusted ID evaluator, you must implement the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` interface.

Property name
Represents the name of this configuration. Enter `BasicIDEvaluator`.

Property value
Defines the name and value pairs that can be used by the trusted ID evaluator implementation. For the default implementation, the trusted list is defined here. When a request comes in and the trusted ID is verified, the user ID, as it appears in the user registry, must be listed in this property. Specify the property as a name and value pair where the name is *trustedId_n*. *n* is an integer starting from 0 and the value is the user ID associated with that name. An example list with the trusted names include two properties.

For example: `trustedId_0 = user1`, `trustedId_1 = user2`. The previous example means that both `user1` and `user2` are trusted. `user1` and `user2` must be listed in the configured user registry

10. Expand the **Trusted ID evaluator reference** section.
11. Click **Enable** to add a new entry. The text you enter for the **Trusted ID evaluator reference** must be the same as the name entered previously in the **Trusted ID evaluator**. Make sure that the name matches exactly because the information is case sensitive. If an entry is already specified, you can change it by clicking **Edit**.

What to do next

You must specify how the server handles the identity assertion authentication method. See "Configuring the server to handle identity assertion authentication" on page 652 if you have not previously specified this information.

Configuring signature authentication for Version 5.x web services with an assembly tool

With the signature authentication method, the request sender generates a signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The request receiver retrieves the Signature security token from the SOAP message and validates it using a Java™ Authentication and Authorization Service (JAAS) login module.

Securing web services for version 5.x applications using signature authentication:

WebSphere Application Server provides several different methods to secure your web services. XML digital signature is one of these methods.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

You can secure your web services by using any of the following methods:

- XML digital signature
- XML encryption
- BasicAuth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

About this task

With the signature authentication method, the request sender generates a signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The request receiver retrieves the Signature security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. To use signature authentication to secure Web services, complete the following tasks:

Procedure

1. Secure the client for signature authentication.
   a. "Configuring the client for signature authentication: specifying the method" on page 657.

   b. "Configuring the client for signature authentication: collecting the authentication information" on page 658.
2. Secure the server for signature authentication.
   a. "Configuring the server to support signature authentication" on page 660.
   b. "Configuring the server to validate signature authentication information" on page 661.

Results

After completing these steps, you have secured your web services using signature authentication.

Configuring the client for signature authentication: specifying the method:

Signature authentication, the use of an X.509 certificate to login on the target server, can be configured.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure signature authentication. A signature refers to the use of an X.509 certificate to login on the target server. For more information on signature authentication, see "Signature authentication method" on page 658.

Complete the following steps to specify signature as the authentication method:

Procedure
1. Launch an assembly tool. For more information, read about assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF** .
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Extension** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Request sender configuration** > **Login configuration** section. The following login configuration options are valid for a managed client and Web services acting as a client are:

   BasicAuth
      Use this option for a managed client.

   Signature
      Use this option for a managed client.

   IDAssertion
      Use this option for web services acting as a client.
7. Select **Signature** to authenticate the client using the certificate used to digitally sign the request.

Results

For more information on getting started with the web services client editor within the assembly tool, see "Configuring the client security bindings using an assembly tool" on page 624.

After you specify signature as the authentication method, you must specify how to collect the authentication information. See "Configuring the client for signature authentication: collecting the

authentication information” for more information.

Signature authentication method:

Signature authentication refers to an X.509 certificate that is sent by the client to the server. The certificate is used to authenticate to the user registry that is configured at the server. When using the signature authentication method, the security token is generated with a ds:Signature and a wsse:BinarySecurityToken element.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following sections.

Signature token generation

The request sender generates a Signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler is specified in the <LoginBinding> element of the bindings file, `ibm-webservicesclient-bnd.xmi`. WebSphere Application Server provides the following callback handler implementation that can be used with the Signature authentication method:
com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler

You can add your own callback handlers that implement the javax.security.auth.callback.CallbackHandler implementation.

Security token validation

The request receiver retrieves the Signature security token from the SOAP message and validates it using a JAAS login module. The <ds:Signature> and <wsse:BinarySecurityToken> elements in the security token are used to perform the validation. If the validation is successful, the login module returns a Java Authentication and Authorization Service (JAAS) Subject. This Subject then is set as the identity of the running thread. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the `<LoginMapping>` element of the bindings file. Default bindings are specified in the `ws-security.xml` file. However, you can override these bindings using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImp CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server searches in the `security.xml` file for a matching configuration name entry. If a match is not found, it searches the `wsjaas.conf` file. WebSphere Application Server provides the system.wssecurity.Signature default configuration entry, which is suitable for the signature authentication method.

Configuring the client for signature authentication: collecting the authentication information:

Signature authentication refers to an X.509 certificate that is sent by the client to the server. The certificate is used to authenticate to the user registry that is configured at the server. The client collects the authentication information for signature authentication.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

You can configure signature authentication. A signature refers to the use of an X.509 certificate to login on the target server.

Complete the following steps to specify how the client collects the authentication information for signature authentication:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the WS Binding tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Security request sender binding configuration** > **Signing information** and click **Edit** to modify the signing key name and signing key locator. To create new signing information, click **Enable**. The certificate that is sent to log in at the server is the one configured in the Signing Information section. Review the key locator information to understand how the signing key name maps to a key within the key locator entry.

   The following list describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following web address: http://www.w3.org/TR/xmldsig-core

   Canonicalization method algorithm
   Canonicalizes the <SignedInfo> element before it is digested as part of the signature operation.

   Digest mehod algorithm
   Represents the algorithm that is applied to the data after transforms are applied, if specified, to yield the <DigestValue> element. The signing of the <DigestValue> element binds the resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

   Signature method algorithm
   Represents the algorithm that is used to convert the canonicalized <SignedInfo> element value into the <SignatureValue> value. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

   Signing key name
   Represents the key entry that is associated with the signing key locator. The key entry refers to an alias of the key, which is used to sign the request.

   Signing key locator
   Represents a reference to a key locator implementation.
7. Expand the **Security request sender binding configuration** > **Login binding** section.
8. Click **Edit** to view the login binding information. Select or enter the following information:

   Authentication method
   Specifies the type of authentication that occurs. Select **Signature** to use signature authentication.

Token value type URI and Token value type URI local name
: When you select **Signature**, you cannot edit token value type Uniform Resource Identifier (URI) and local name values. Specifies custom authentication types. For signature authentication, leave these fields blank.

Callback handler
: Specifies the Java Authentication and Authorization Server (JAAS) callback handler implementation for collecting signature information. Enter the following callback handler for signature authentication:

```
com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
```

This callback handler is used because the signature method does not require user interaction.

Basic authentication user ID and Basic authentication password
: Leave the BasicAuth fields blank when signature authentication is used.

Property name and property value
: This field enables you to enter properties and name and value pairs for use by custom callback handlers. For signature authentication, do not enter any information.

What to do next

Other customization entries: There is a basic authentication entry in the Port Qualified Name Binding Details section. This entry is used for HTTP transport authentication, which might be required if the router servlet is protected.

Information specified in the Web Services Security signature authentication section overrides the basic authentication information specified in the Port Qualified Name Binding Details section for authorizing the Web service.

To use the signature authentication method, you must specify the authentication method in the Login configuration section of an assembly tool.

Configuring the server to support signature authentication:

Signature authentication refers to an X.509 certificate sent by the client to the server. The certificate is used to authenticate to the user registry configured at the server. After a request is received by the server that contains the certificate, the server needs to log in to form a credential. The credential is used for authorization. You can configure signature authentication at the server.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

If the certificate supplied cannot be mapped to an entry in the user registry, an exception is provided and the request ends without invoking the resource.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective by clicking **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.

5. Click the **Extensions** tab, which is located at the bottom of the Web Services Editor within the assembly tool.
6. Expand the **Request receiver service configuration details** > **Login configuration** section. You can select from the following options:
   - BasicAuth
   - Signature
   - ID assertion
   - Lightweight Third Party Authentication (LTPA)
7. Select **Signature** to authenticate the client using an X509 certificate. The certificate that is sent from the client is the certificate that issued for signing the message. You must be able to map this certificate to the configured user registry. For Local operating system (OS) registries, the common name (cn) of the distinguished name (DN) is mapped to a user ID in the registry. For Lightweight Directory Access Protocol (LDAP), you can configure multiple mapping modes:
   - `EXACT_DN` is the default mode that directly maps the DN of the certificate to an entry in the LDAP server.
   - `CERTIFICATE_FILTER` is the mode that provides the LDAP advanced configuration with a place to specify a filter that maps specific attributes of the certificate to specific attributes of the LDAP server.

What to do next

For more information on getting started with the web services editor within the assembly tool, see “Configuring the server security bindings using an assembly tool” on page 627.

After you specify how the server handles signature authentication information, you must specify how the server validates the authentication information. See the task for configuring the server to validate signature authentication.

Configuring the server to validate signature authentication information:

Signature authentication refers to an X.509 certificate sent by the client to the server. The certificate is used to authenticate to the user registry configured at the server. After a request is received by the server that contains the certificate, the server needs to log in to form a credential. The credential is used for authorization. You can validate signature authentication at the server.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

If the certificate supplied cannot be mapped to an entry in the user registry, an exception is thrown and the request ends without invoking the resource.

Complete the following steps to configure the server to validate signature authentication:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective by clicking **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.

5. Click the **Binding Configurations** tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver binding configuration details** > **Login mapping** section.
7. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog is displayed and you select (or enter) the following information:

Authentication method
Specifies the type of authentication. Select **Signature** to use signature authentication.

Configuration name
Specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the signature authentication method, enter `system.wssecurity.Signature` for the JAAS login configuration name. This specification logs in with the `com.ibm.wsspi.wssecurity.auth.module.SignatureLoginModule` JAAS login module.

Use token value type
Determines if you want to specify a custom token type. For the default authentication method selections, you can leave this field blank.

URI and local name
When you select Signature method, you cannot edit the token value type URI and local name values. Specifies custom authentication types. For signature authentication, you can leave this field blank.

Callback handler factory class name
Creates a JAAS CallbackHandler implementation that understands the following callback handlers:

- javax.security.auth.callback.NameCallback
- javax.security.auth.callback.PasswordCallback
- com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
- com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback
- com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback

For any of the default authentication methods (BasicAuth, IDAssertion, and Signature), use the callback handler factory default implementation. Enter the following class name for any of the default authentication methods including signature:

```
com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl
```

This implementation creates the correct callback handler for the default implementations.

Callback handler factory property name and callback handler factory property value
Specifies callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For signature, you can leave this field blank.

Login mapping property name and login mapping property value
Specifies properties for a custom login mapping to use. For the default implementations including signature, you can leave this field blank.

What to do next

Specify how the server handles the signature authentication method. See "Configuring the server to support signature authentication" on page 660 if you have not previously specified this information.

Configuring pluggable tokens for Version 5.x web services with an assembly tool

WebSphere Application Server provides several different methods to secure your web services, including the pluggable token method. To use pluggable tokens to secure your web services, you must configure both the client request sender and the server request receiver.

Securing web services for version 5.x applications using a pluggable token:

To use pluggable tokens to secure your web services, you must configure both the client request sender and the server request receiver. You can configure your pluggable tokens using the WebSphere Application Server administrative console.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services. A pluggable token is one of these methods. You might secure your Web services by using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

Complete the following steps to secure your web services using a pluggable token:

Procedure

1. Generate a security token using the Java Authentication and Authorization Service (JAAS) CallbackHandler interface. The Web Services Security runtime uses the JAAS CallbackHandler interface as a plug-in to generate security tokens on the client side or when web services are acting as a client.
2. Configure your pluggable token. For more information, see the following tasks:
   - "Configuring pluggable tokens using an assembly tool"
   - Configuring pluggable tokens using the administrative console

Configuring pluggable tokens using an assembly tool:

The following information describes how to configure a pluggable token using an assembly tool.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This document describes how to configure a pluggable token in the request sender (`ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file) and request receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` file).

The pluggable token is required for the request sender and request receiver because they are a pair. The request sender and the request receiver must match for the receiver to accept a request.

Prior to completing these steps, it is assumed that you have already created a web service that is based on the Java Platform, Enterprise Edition (Java EE) specification. See either of the following topics for an introduction of how to manage Web Services Security binding information for the server:

- “Configuring the server security bindings using an assembly tool” on page 627
- “Configuring the server security bindings using the administrative console” on page 1004

About this task

You must specify the security constraints in the `ibm-webservicesclient-ext.xmi` and the `ibm-webservices-ext.xmi` files for the required tokens using an IBM assembly tool.

Complete the following steps to configure the request sender using the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` files:

Procedure

1. Launch an assembly tool. For more information, read about assembly tools.
2. Switch to the Java EE perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Extension** tab. The web service client security extensions editor is displayed.
   a. Under Service References, select an existing service reference or click **Add** to create a new reference.
   b. Under Port Qname Bindings, select an existing port qualified name for the selected service reference or click **Add** to create a new port name binding.
   c. Under Request Sender Configuration: Login Configuration, select an exiting authentication method or type in a new one in the editable list box (Lightweight Third Party Authorization (LTPA) is a supported token generation when web services are acting as client).
   d. Click **File** > **Save** to save the changes.
6. Click the **Web services client binding** tab. The web services client binding editor is displayed.
   a. Under Port qualified name binding, select an existing entry or click **Add** to add a new port name binding. The web services client binding editor displays for the selected port.
   b. Under Login binding, click **Edit** or **Enable**. The Login Binding dialog box is displayed.
      1) In the Authentication Method field, enter the authentication method. The authentication method that you enter in this field must match the authentication method defined on the **Security Extension** tab for the same web service port. This field is mandatory.
      2) (Optional) Enter the token value type information in the URI and Local name fields. These fields are ignored for the BasicAuth, Signature, and IDAssertion authentication methods, but required for other authentication methods. The token value type information is inserted into the <wsse:BinarySecurityToken>@ValueType element for binary security token and is used as the namespace for the XML-based token.
      3) Enter an implementation of the Java Authentication and Authorization Service (JAAS) `javax.security.auth.callback.CallbackHandler` interface. This field is mandatory.
      4) Enter the basic authentication information in the **User ID** and **Password** fields. The basic authentication information is passed to the construct of the `CallbackHandler` implementation. The use of the basic authentication information depends on the implementation of `CallbackHandler`.
      5) In the Property field, add name and value pairs. These pairs are passed to the construct of the `CallbackHandler` implementation as `java.util.Map` values.
      6) Click **OK**.

      Click **Disable** under Login binding on the **Web services client port binding** tab to remove the authentication method login binding.

c. Click **File** > **Save** to save the changes.

7. In the Package Explorer window, right-click the `webservices.xml` file and click **Open with** > **Web services editor**. The Web Services window displays.

   a. Click the **Security extensions** tab. The Web Service Security extensions editor is displayed.

      1) Under Web Services Description Extension, select an existing service reference or click **Add** to create a new extension.

      2) Under Port Component Binding, select an existing port qualified name for the selected service reference or click **Add** to create a new one.

      3) Under Request Receiver Service Configuration Details: Login Configuration, select an exiting authentication method or click **Add** and enter a new method in the **Add AuthMethod** field that displays. You can select multiple authentication methods for the request receiver. The security token of the incoming message is authenticated against the authentication methods in the order that they are specified in the list. Click **Remove** to remove the selected authentication method or methods.

   b. Click **File** > **Save** to save the changes.

   c. Click the **Bindings** tab. The web services bindings editor is displayed.

      1) Under web service description bindings, select an existing entry or click **Add** to add a new web services descriptor.

      2) Click the **Binding configurations** tab. The web services binding configurations editor is displayed for the selected web services descriptor.

      3) Under Request receiver binding configuration details: login mapping, click **Add** to create a new login mapping or click **Edit** to edit the selected login mapping. The Login mapping dialog is displayed.

         a) In the Authentication method field, enter the authentication method. The information entered in this field must match the authentication method defined on the **Security Extensions** tab for the same web service port. This field is mandatory.

         b) In the **Configuration name** field, enter a JAAS login configuration name. This field is mandatory. You must define the JAAS login configuration name in the WebSphere Application Server administrative console under **Security** > **Global security**. Under Authentication, click **Java Authentication and Authorization Service** > **Application logins**. For more information, read about configuring programmatic logins for Java Authentication and Authorization Service.

         c) (Optional) Select **Use Token value type** and enter the token value type information in the URI and Local name fields. This information is optional for BasicAuth, Signature and IDAssertion authentication methods, but required for any other authentication method. The token value type is used to validate the <wsse:BinarySecurityToken>@ValueType element for binary security tokens and to validate the namespace of the XML-based token.

         d) Under Callback Handler Factory, enter an implementation of the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface in the Class name field. This field is mandatory.

         e) Under Callback Handler Factory property, click **Add** and enter the name and value pairs for the Callback Handler Factory Property. These name and value pairs are passed as `java.util.Map` to the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory.init()` method. The use of these name and value pairs is determined by the `CallbackHandlerFactory` implementation.

         f) Under Login Mapping Property, click **Add** and enter the name and value pairs for the Login mapping property. These name and value pairs are available to the JAAS Login Modules through the `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback` JAAS Callback interface. Click **Remove** to delete the selected login mapping.

         g) Click **OK**.

d. Click **File** > **Save** to save the changes.

Results

The previous steps define how to configure the request sender to create security tokens in the SOAP message and to configure the request receiver to validate the security tokens found in the incoming SOAP message. WebSphere Application Server supports pluggable security tokens.

You can use the authentication method defined in the login bindings and login mappings to generate security tokens in the request sender and validate security tokens in the request receiver.

What to do next

After you configure pluggable tokens, you must configure both the client and the server to support pluggable tokens. See the following topics to configure the client and the server:
- Configuring the client for LTPA token authentication: Specifying LTPA token authentication
- Configuring the client for LTPA token authentication: Collecting the authentication information
- Configuring the server to handle LTPA token authentication
- Configuring the server to validate LTPA token authentication information

Configuring the client for LTPA token authentication: specifying LTPA token authentication:

To configure Lightweight Third-Party Authentication (LTPA) token authentication, specify LTPA token authentication. Only configure the client for LTPA token authentication if the authentication mechanism configured in WebSphere Application Server is LTPA.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Use this task to configure Lightweight Third-Party Authentication (LTPA) token authentication. Only configure the client for LTPA token authentication if the authentication mechanism configured in WebSphere Application Server is LTPA. When a client authenticates to a WebSphere Application Server, the credential created contains an LTPA token. When a web service calls a downstream web service, you can configure the first web service to send the LTPA token from the originating client. Do not attempt to configure LTPA from a pure client. LTPA works only when you configure the client-side of a web service acting as a client to a downstream web service. For the downstream web service to validate the LTPA token, the LTPA keys on both servers must be the same.

Complete the following steps to specify LTPA token as the authentication method:

Procedure
1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **Extensions** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Request sender configuration** > **Login configuration** section.

7. Select **LTPA** as the authentication method. For more conceptual information on LTPA authentication, see “Lightweight Third Party Authentication” on page 670.

What to do next

After you specify LTPA token as the authentication method, you must specify how to collect the LTPA token information. See “Configuring the client for LTPA token authentication: collecting the authentication method information” for more information.

Configuring the client for LTPA token authentication: collecting the authentication method information:

To configure Lightweight Third-Party Authentication (LTPA) token authentication, collect the LTPA token authentication information. Do not configure the client for LTPA token authentication unless the authentication mechanism configured in WebSphere Application Server is LTPA.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Use this task to configure Lightweight Third-Party Authentication (LTPA) token authentication. Do not configure the client for LTPA token authentication unless the authentication mechanism configured in WebSphere Application Server is LTPA. When a client authenticates to a WebSphere Application Server, the credential created contains an LTPA token. When a web service calls a downstream web service, you can configure the first web service to send the LTPA token from the originating client. Do not attempt to configure LTPA from a pure client. LTPA works only when you configure the client-side of a web service acting as a client to a downstream web service. In order for the downstream web service to validate the LTPA token, the LTPA keys on both servers must be the same.

Complete the following steps to specify how to collect the LTPA token authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Bindings** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Security request sender binding configuration** > **Login binding** section.
7. Click **Edit** to view the login binding information and select **LTPA**. If LTPA is not already there, enter it as an option. The login binding dialog is displayed. Select or enter the following information:

   Authentication method
      Specifies the type of authentication that occurs. Select **LTPA** to use identity assertion.

   Token value type URI and token value type local name
      When you select **LTPA**, you must edit the token value type **URI** (Uniform Resource Identifier) and the **local name** fields. Specifies values for custom authentication types, which are authentication methods not mentioned in the specification. For the token value type **URI** field, enter the following string: `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`. For the **local name** field, enter the following string: `LTPA`.

Callback handler
Specifies the Java Authentication and Authorization Service (JAAS) callback handler implementation for collecting the LTPA information. Specify the `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler` implementation for LTPA.

Basic authentication user ID and basic authentication password
For LTPA, you can leave these fields empty. However, when you omit this information, the LTPA `CallbackHandler` implementation attempts to obtain the LTPA token from the invocation (RunAs) credential. If an invocation (RunAs) credential does not exist, then the LTPA token is not propagated.

Property name and property value
For LTPA, you can leave these fields blank.

What to do next

See "Configuring the client for LTPA token authentication: specifying LTPA token authentication" on page 666 if you have not previously specified this information.

Configuring the server to handle LTPA token authentication information:

Lightweight Third-Party Authentication (LTPA) is a type of authentication mechanism in WebSphere Application Server security that defines a particular token format. The purpose of the LTPA token authentication is to flow the LTPA token from the first web service, which authenticated the originating client, to the downstream web service. You can configure the server for LTPA token authentication.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure LTPA. Do not attempt to configure LTPA from a pure client. After the downstream web service receives the LTPA token, it validates the token to verify that the token has not been modified and has not expired. For validation to be successful, the LTPA keys that are used by both the sending and receiving servers must be the same.

Complete the following steps to specify that LTPA is the authentication method. The authentication method indicated in these steps must match the authentication method that is specified for the client.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the Extensions tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver service configuration details** > **Login configuration** section. You can select from the following options:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**

- **LTPA**

7. Select **LTPA** to authenticate the client using the LTPA token received from the request.

What to do next

After you specify the authentication method, you must specify the information that the server must validate. See "Configuring the server to validate LTPA token authentication information" for more information.

Configuring the server to validate LTPA token authentication information:

Lightweight Third-Party Authentication (LTPA) is a type of authentication mechanism in WebSphere Application Server security that defines a particular token format. The purpose of the LTPA token authentication is to flow the LTPA token from the first web service, which authenticated the originating client, to the downstream web service. You can configure the server to validate LTPA token authentication.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure LTPA. Do not attempt to configure LTPA from a pure client. After the downstream web service receives the LTPA token, it validates the token to verify that the token has not been modified and has not expired. For validation to be successful, the LTPA keys used by both the sending and receiving servers must be the same.

Complete the following steps to specify how the server must validate the LTPA token authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java 2 Platform, Enterprise Edition (J2EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the Binding Configurations tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver binding configuration details** > **Login mapping** section.
7. Click **Edit** to view the login mapping information. The login mapping information is displayed. Select or enter the following information:

   Authentication method
   Specifies the type of authentication that occurs. Select `LTPA` to use LTPA token authentication.

   Configuration name
   Specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the LTPA authentication method, enter `WSLogin` for the JAAS login configuration name. This configuration understands how to validate an LTPA token.

   Use token value type
   Determines if you want to specify a custom token type. For LTPA authentication, you must select this option because LTPA is considered a custom type. LTPA is not in the Web Services Security Specification.

Token value type URI and local name
Specifies custom authentication types. If you select **Use Token value type** you must enter data into the Token value Type URI (Uniform Resource Identifier) and local name fields. For the token value type URI field, enter the following string: `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`. For the local name, enter the following string: `LTPA`

Callback handler factory class name
Creates a JAAS CallbackHandler implementation that understands the following callback handlers:

- javax.security.auth.callback.NameCallback
- javax.security.auth.callback.PasswordCallback
- com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
- com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback
- com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback

For any of the default authentication methods (BasicAuth, IDAssertion, Signature, and LTPA), use the callback handler factory default implementation. Enter the following class name for any of the default authentication methods including LTPA:

```
com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl
```

This implementation creates the correct callback handler for the default implementations.

Callback handler factory property
Specifies callback handler properties for custom callback handler factory implementations. Default callback handler factory implementation does not any property specifications. For LTPA, leave this field blank.

Login mapping property
Specifies properties for a custom login mapping. For default implementations including LTPA, leave this field blank.

What to do next

See the task for configuring the server to handle LTPA token authentication information if you have not previously specified this information.

Lightweight Third Party Authentication:

When you use the lightweight third party authentication (LTPA) method, the <wsse:BinarySecurityToken> security token is generated. On the request sender side, the security token is generated by invoking a callback handler. On the request receiver side, the security token is validated by a Java Authentication and Authorization Service (JAAS) login module.

Important: The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

The following information describes token generation and token validation operations.

LTPA token generation
The request sender uses a callback handler to generate an LTPA security token. The callback handler returns a security token that is inserted in the SOAP message. Specify the appropriate callback handler in the <LoginBinding> element of the bindings file (`ibm-webservicesclient-bnd.xmi`). The following callback handler implementation can be used with the LTPA authentication method:

- `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler`

You can add your own callback handlers that implement the `javax.security.auth.callback.CallbackHandler` property.

When using the LTPA authentication method (or any authentication method other than BasicAuth, Signature or IDAssertion), the TokenValueType attribute of the <LoginBinding> element in the bindings file (`ibm-webservicesclient-bnd.xmi`) must be specified. The values to use for the LTPA TokenValueType attribute are:

- `uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2"`
- `localName="LTPA"`

LTPA token validation

The request receiver retrieves the LTPA security token from the SOAP message and validates the message using a JAAS login module. The <wsse:BinarySecurityToken> security token is used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. Subsequently, this Subject is set as the identity of the running thread. If the validation fails, the request is rejected with a SOAP fault.

The appropriate JAAS login configuration to use is specified in the bindings file <LoginMapping> element. Default bindings specified in the `ws-security.xml` file, but these can be overridden using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory, a ConfigName and a TokenValueType attribute. The CallbackHandlerFactory specifies the name of a class to use to create the JAAS CallbackHandler object. A CallbackHandlerFactory implementation is provided (com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl). The ConfigName attribute specifies a JAAS configuration name entry. The Web Services Security run time first searches the `security.xml` file for a matching entry and if a matching entry is not found, the run time searches the `wsjaas.conf` file. A default configuration entry suitable for the LTPA authentication method is provided (`WSLogin`). An appropriate TokenValueType element is located in the LTPA LoginMapping section of the default `ws-security.xml` file.

Administering Web Services Security

To secure web services, you must consider a broad set of security requirements, including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, delegation, and auditing across a spectrum of application and business topologies. You can choose to configure Web Services Security for the application level, the server level or the cell level, depending upon your environment and security needs.

Configuring HTTP outbound transport level security with the administrative console

You can configure HTTP outbound transport level security with the administrative console.

Before you begin

This task is one of several ways that you can configure the HTTP outbound transport level security for a web service acting as a client to another web service server. You can also configure the HTTP outbound transport level security with an assembly tool or by using the Java properties. If you do not configure the HTTP outbound transport level security, the web services runtime defers to the Web Services for Java Platform, Enterprise Edition (Java EE) security runtime in the WebSphere product for an effective Secure Sockets Layer (SSL) configuration. If there is no SSL configuration with the Java EE security runtime in the WebSphere product, the Java Secure Socket Extension (JSSE) system properties are used.

About this task

If you choose to configure the HTTP outbound transport level security with the administrative console or an assembly tool, the Web Services Security binding information is modified. You can use the administrative console to configure the web services client security bindings if you have deployed or installed the web services application into WebSphere Application Server. If you have not installed the web

services application, you can configure the HTTP SSL configuration with an assembly tool. This task assumes that you have deployed the web services application into the WebSphere product. See the deploying web services applications onto application servers information to learn more about deploying web services.

If you configure the HTTP outbound transport level security using the standard Java properties for JSSE, the properties are configured as system properties. The configuration specified in the binding takes precedence over the Java properties. However, the configurations that are specified by the Java EE security programming model , or that are associated the Dynamic selection , have higher precedence.

See the secure communications using Secure Sockets Layer information to learn more implementing transport level security.

Configure the HTTP outbound transport level security with the following steps provided in this task section.

Procedure

1. Open the administrative console.
2. Click **Applications** > **Enterprise Applications** > *application_instance* > **Manage Modules** > *module_instance* . Under Web Services Security Properties, click **Web Services: Client security bindings**.
3. Under the heading, HTTP SSL Configuration, click **Edit** to access the HTTP SSL configuration panel. Select the **Centrally-managed** radio button so that the system runtime chooses the SSL configuration that is based on the current context. Select the **Specific to this Web service port** radio button if you want to choose the SSL configuration in the HTTP SSL configuration drop down box.

Results

You have configured the HTTP outbound transport level security for a web service acting as a client to another web service with the administrative console.

HTTP SSL Configuration collection:

Use this page to configure transport-level Secure Sockets Layer (SSL) security. You can use this configuration when a web service is a client to another web service.

You can use transport-level security to enable HTTP SSL (or HTTPS). Transport-level security can be enabled or disabled independently from message-level security. Because transport-level security provides minimal security, use message-level security when security is essential to the web service application.

To view this administrative console page, complete the following steps:

1. Click **Applications > Application Types > WebSphere enterprise applications > *application_name*** .
2. Click **Manage modules > *URI_file_name* > Web Services: Client Security Bindings** .
3. Under HTTP SSL Configuration, click **Edit**.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

SSL configuration: Select the **Centrally-managed** radio button so that the system runtime chooses the SSL configuration that is based on the current context. Select the **Specific to this web service port** radio button if you want to choose the SSL configuration in the **HTTP SSL configuration** drop down box.

HTTP SSL configuration: The **HTTP SSL configuration** drop down box lists the SSL configurations used with the HTTP transport for a port. Use this drop down box if you want to select the SSL configuration rather than using the SSL configuration that the runtime automatically selects. To use the drop down box,

select the **Specific to the web service port** radio button that is located in the **SSL configuration** field. After you select the radio button, you can click the drop down box to view and select an SSL configuration.

Configuring HTTP outbound transport level security using Java properties

You can configure the HTTP outbound transport level security for a web service using Java properties.

Before you begin

This task is one of three ways that you can configure HTTP outbound transport-level security for a web service that is acting as a client to another web service. You can also configure the HTTP outbound transport level security with the administrative console or an assembly tool. However, you can also use this task to configure the HTTP outbound transport-level security for a web service client.

About this task

If you choose to configure the HTTP outbound transport-level security with the administrative console or an assembly tool, the Web Services Security binding information is modified.

If you configure the HTTP outbound transport-level security using Java properties, the properties are configured as system properties. However, the configuration specified in the binding takes precedence over the Java properties.

You can configure the HTTP outbound transport-level security using WebSphere SSL properties or JSSE SSL properties. However, the WebSphere SSL properties take precedence over the JSSE SSL properties.

Configure the HTTP outbound transport-level security with the following steps provided in this task section.

Procedure

1. Create a property file that includes the following properties:

```
com.ibm.ssl.protocol
com.ibm.ssl.keyStoreType
com.ibm.ssl.keyStore
com.ibm.ssl.keyStorePassword
com.ibm.ssl.trustStoreType
com.ibm.ssl.trustStore
com.ibm.ssl.trustStorePassword
```

2. Set the `com.ibm.webservices.sslConfigURL` Java system property to the absolute path of the created property file. If no WebSphere SSL properties are defined, the JSSE SSL properties are used. Set the JSSE SSL properties as JVM custom properties. See Secure transports with JSSE and JCE programming interfaces for more information about setting the JSSE SSL properties.

Results

You have configured the HTTP outbound transport-level security for a web service acting as a client to another web service.

Configuring HTTP basic authentication for JAX-RPC web services with the administrative console

You can configure HTTP basic authentication for Java API for XML-based RPC (JAX-RPC) web services with the administrative console.

Before you begin

This task is one of three ways that you can configure HTTP basic authentication. You can also configure HTTP basic authentication with an assembly tool or by modifying the HTTP properties programmatically.

If you choose to configure HTTP basic authentication with the administrative console or an assembly tool, the Web Services Security binding information is modified. You can use the administrative console to configure HTTP basic authentication if you have deployed or installed the web services application into WebSphere Application Server. If you have not installed the web services application, then you can configure the security bindings with an assembly tool. This task assumes that you have deployed the web services application into the WebSphere product. To learn more about deploying web services, see the deploying Web services applications onto application servers information.

If you configure HTTP basic authentication programmatically, the properties are configured in the Stub or Call instance. The values set programmatically take precedence over the values defined in the binding.

About this task

The HTTP basic authentication that is discussed in this topic is orthogonal to WS-Security and is distinct from basic authentication that WS-Security supports. WS-Security supports basic authentication token, not HTTP basic authentication.

Configure HTTP basic authentication with the following steps provided in this task section.

Procedure

Open the administrative console.

1. Click **Applications** > **Enterprise Applications** > *application_instance* > **Manage Modules** > *module_instance* > **Web services: Client security bindings**.
2. Click **HTTP Basic Authentication** to access the HTTP basic authentication panel. Enter the values in the HTTP Basic Authentication panel.

Results

You have configured the HTTP basic authentication.

HTTP basic authentication collection:

Use this page to specify a user name and password for transport-level basic authentication security for this port. You can use this configuration when a web service is a client to another web service.

You can use transport-level security to enable basic authentication. Transport-level security can be enabled or disabled independently from message-level security. Because transport-level security provides minimal security, use message-level security when security is essential to the web service application.

To view this administrative console page, complete the following steps:

1. Click **Applications > Application Types > WebSphere enterprise applications >** ***application_name*** .
2. Click **Manage modules >** ***URI_file_name*** **> Web Services: Client Security Bindings** .
3. Under HTTP basic authentication, click **Edit**.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

Basic authentication ID:

The user name for the HTTP basic authentication for this port is set in this field.

Use the Basic authentication ID field to specify the user name for the HTTP basic authentication for this port.

Basic authentication password:

The password for the HTTP basic authentication for this port is set in this field.

Use the Basic authentication password field to specify the password for the HTTP basic authentication for this port.

Configuring custom properties to secure web services

You can configure name-value pairs of data, where the name is a property key and the value is a string value that you can use to set internal system configuration properties. Defining a new property enables you to configure a setting beyond that which is available through options in the administrative console.

About this task

The Web Services Security custom properties topic provides the following information about each custom property:

- Provides a detailed description of the property
- States the type of data that is needed to set the property
- Provides a list of possible values
- Lists the default value

Important: Custom properties that you set for the default consumer or default generator bindings take precedence over general custom properties that you set as additional properties. However, custom bindings take precedence over default bindings.

The following steps explain how to set custom properties to secure Web services:

Procedure

- Set custom properties for Java API for XML-based RPC (JAX-RPC) applications. You can set custom properties to secure Web services for JAX-RPC applications in multiple locations within the administrative console. You can set these custom properties for the default consumer, default generator, or both bindings. Also, you can set custom properties as general additional properties. Collectively, the default consumer bindings, the default generator bindings, and the additional properties are referred to as the *default bindings*.
  - **Custom bindings**
    1. Expand **Applications > Application Types**.
    2. Click **WebSphere enterprise applications** > *application_name*.
    3. Under Modules, click **Manage Modules** > *module_name*.
    4. Under Web Services Security Properties, click **Web services: Server security bindings** or **Web services: Client security bindings** > **Edit custom**.
  - **Default consumer bindings**
    1. Expand **Servers > Server types**.
    2. Click **WebSphere applications servers >** *server_name*.
    3. Under Security, click **JAX-WS and JAX-RPC security runtime**.
    4. Under JAX-RPC Default Consumer Bindings, click **Properties**.
  - **Default generator bindings**
    1. Expand **Servers > Server types**.
    2. Click **WebSphere applications servers >** *server_name*.
    3. Under Security, click **JAX-WS and JAX-RPC security runtime**.
    4. Under JAX-RPC Default Generator Bindings, click **Properties**.
  - **Additional properties**

1. Expand **Servers > Server types**.
2. Click **WebSphere applications servers >** *server_name*.
3. Under Security, click **JAX-WS and JAX-RPC security runtime**.
4. Under Custom properties, click **Custom properties**.

Order of precedence for custom properties with JAX-RPC applications: Custom properties that you set in the WS-Security extension and custom bindings take precedence over custom properties that you set in the default bindings. Custom properties that you set in the WS-Security bindings take precedence over custom properties that you set in the WS-Security extension. Custom properties that you set in the generator or sender and consumer or receiver bindings take precedence over custom properties that you set in the additional properties.

- Set custom properties for Java API for XML-Based Web Services (JAX-WS) applications. You can set custom properties to secure web services for JAX-WS applications in multiple locations within the administrative console. You can set these custom properties in the custom bindings for an application, in the WS-Security default bindings, or for inbound and outbound messages.
  - **Custom bindings for an application**
    1. Expand **Services > Service clients** or **Services > Service providers**.
    2. Click *service_name* > *binding_name*.
    3. Click **WS-Security**.
    4. Under the Main message security policy bindings heading, click **Custom properties**.
  - **WS-Security default bindings**
    1. Expand **Services > Policy sets** .
    2. Click **General provider policy set bindings** or **General client policy set bindings** > *binding_name* > **WS-Security**.
    3. Under Main Message Security Policy Bindings, click **Custom properties**.
  - **Inbound and outbound custom properties**
    1. Expand **Services > Policy sets**.
    2. Click **Default policy set bindings**.
    3. Under the Policy heading, click **WS-Security**.
    4. Under the Main message security policy bindings heading, click **Custom properties**.

    For more information, see the Inbound and outbound custom properties topic.

  Alternatively, you can set these properties as parameters or inbound binding properties for your JAX-WS application using wsadmin scripting. The following WS-Security policy type property names are used in the setBinding function:
  - application.parameters
  - application.securityinboundbindingconfig.properties

– application.securityoutboundbindingconfig.properties

Note: Custom properties for policy set bindings can not be set using the Web Services Security API. The custom properties must be set using the administrative console.

Web services security custom properties:

You can configure name-value pairs of data, where the name is a property key and the value is a string value that you can use to set internal system configuration properties. Defining a new property enables you to configure a setting beyond that which is available through options in the administrative console.

Custom properties for web services security can be set in various levels of the application server and for JAX-RPC versus JAX-WS applications. The following list of custom properties provides information on where the custom property is set and how it is used.

The web services security generic security token login module custom properties and the Web services security SAML token custom properties are documented in other information topics. Links to these topics are provided in the Related reference section of this topic.

You can define the following web services security custom properties:

- "com.ibm.ws.wssecurity.createSTR"
- "com.ibm.ws.wssecurity.sc.FaultCode" on page 678
- "com.ibm.wsspi.wssecurity.Caller.assertionLoginConfig" on page 678
- "com.ibm.wsspi.wssecurity.config.disableWSSIfApplicationSecurityDisabled" on page 678
- "com.ibm.wsspi.wssecurity.config.gen.checkCacheUsernameTokens" on page 679
- "com.ibm.wsspi.wssecurity.config.request.setMustUnderstand and com.ibm.wsspi.wssecurity.config.response.forceMustUnderstandEqualsOne" on page 679
- "com.ibm.wsspi.wssecurity.config.token.inbound.retryOnceAfterTrustFailure" on page 681
- "com.ibm.wsspi.wssecurity.consumer.timestampRequired" on page 681
- "com.ibm.wsspi.wssecurity.dsig.inclusiveNamespaces" on page 682
- "com.ibm.wsspi.wssecurity.dsig.oldEnvelopedSignature" on page 682
- "com.ibm.wsspi.wssecurity.generator.usewssobject" on page 683
- "com.ibm.wsspi.wssecurity.login.useSoap12FaultCodes" on page 683
- "com.ibm.wsspi.wssecurity.token.forwardable" on page 684
- "com.ibm.wsspi.wssecurity.token.username.addNonce and com.ibm.wsspi.wssecurity.token.username.addTimestamp" on page 684
- "com.ibm.wsspi.wssecurity.token.username.password.forwardable" on page 684
- "com.ibm.wsspi.wssecurity.token.username.verifyNonce and com.ibm.wsspi.wssecurity.token.username.verifyTimestamp" on page 684
- "com.ibm.wsspi.wssecurity.token.UsernameToken.disableUserRegistryCheck" on page 684
- "com.ibm.wsspi.wssecurity.tokenGenerator.ltpav1.pre.v7" on page 685

com.ibm.ws.wssecurity.createSTR

The com.ibm.ws.wssecurity.createSTR property creates a security token reference to the security token in the SOAP security header when you specify a `True` value.

You can set this property to `True`, the com.ibm.ws.wssecurity.createSTR property creates a security token reference to the security token in the SOAP security header. Set this custom property to `True` when the following conditions exist:

- The referencing mechanism for the token signature is the STR Dereference Transform, `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform`
- The SignedParts element for the WS-Security policy contains an XPath value that represents the SecurityTokenReference.

Information	Value
Data type	String
Values	True, False
Default	False

The value for this property is case-insensitive.

com.ibm.ws.wssecurity.sc.FaultCode

Use this custom property in a JAAS login module to set the SOAP fault code in the event of an error. If this property is not specified, the SOAP fault code wsse:FailedAuthentication is always returned.

In the custom JAAS login module, set the com.ibm.ws.wssecurity.sc.FaultCode property on the wssecurity context to the QName of the fault code that you want to use. For example:

```
fcQname = new QName(
          "http://schemas.xmlsoap.org/ws/2003/06/secext",
          "FailedCheck");
this._context = propertyCallback.getProperties();
 _context.put("com.ibm.ws.wssecurity.sc.FaultCode", fcQname);
```

Information	Value
Data type	String
Default	none

com.ibm.wsspi.wssecurity.Caller.assertionLoginConfig

The com.ibm.wsspi.wssecurity.Caller.assertionLoginConfig property, which is configured on the caller part, specifies the name of the JAAS login configuration that is used by Web Services Security to obtain WebSphere Application Server authorization credentials. You must configure this property using an assembly tool such as the Rational Application Developer. For more information, see the "Configuring the caller in consumer security constraints" topic for Rational Application Developer. Within this topic, this custom property is set when you configure identity assertion.

Use this property with WS-Security V1.0 JAX-RPC applications only.

Information	Value
Data type	String
Default	system.DEFAULT

com.ibm.wsspi.wssecurity.config.disableWSSIfApplicationSecurityDisabled

When you set the com.ibm.wsspi.wssecurity.config.disableWSSIfApplicationSecurityDisabled custom property to `true`, Web Services Security does not enforce the configured WS-Security constraints if application security is disabled on the application server. You can use this custom property to debug services in a non-secure environment without needing to remove security constraints from web services applications.

Note: Use this custom property for diagnosis purposes only. Do not use it in a production environment.

Information	Value
Data type	String
Values	true, false
Defaultfalse	false

You can set this custom property as an inbound custom property or an inbound and outbound custom property for the policy set bindings. Complete the following steps in the administrative console to set the custom property:

1. Expand **Services** > **Policy sets**.
2. Click **General provider policy set bindings** or **General client policy set bindings**.
3. Click the *binding_name*.
4. Under the Policy heading, click **WS-Security** > **Custom properties**.

You can also set this custom property as a parameter or as an inbound binding property on your application using wsadmin tooling. The following WS-Security policy-type property names are used in setBinding:

- application.parameters
- application.securityinboundbinding config.properties

com.ibm.wsspi.wssecurity.config.gen.checkCacheUsernameTokens

The com.ibm.wsspi.wssecurity.config.gen.checkCacheUsernameTokens custom property specifies whether to cache UsernameTokens all of the time, which is the default behavior, or cache them as determined by a set of rules. You can configure this custom property for the token generator or as an additional property.

When the com.ibm.wsspi.wssecurity.config.gen.checkCacheUsernameTokens custom property is set to false, UsernameTokens are always cached on client threads. When you set this custom property to true, the web services security run time determines whether UsernameTokens are cached based on the following rules:

- Never cache UsernameTokens if the application is running on an application server.
- Cache UsernameTokens if the token generator for the UsernameToken has the following callback handler configured: com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler.

This custom property applies to the JAX-RPC run time only. Use an assembly tool, such as Rational Application Developer, to set the custom property within the encrypted message part bindings.

Information	Value
Data type	String
Values	true, false
Default	false

com.ibm.wsspi.wssecurity.config.request.setMustUnderstand and com.ibm.wsspi.wssecurity.config.response.forceMustUnderstandEqualsOne

In WebSphere Application Server prior to Version 6.1x, the mustUnderstand=1 attribute in the `<wsse:Security>` tag in the SOAP header on the request from the web services client was hardcoded. It was not possible to configure the mustUnderstand attribute in the SOAP Web Services Security header. In an update to the product, an administrator can configure the attribute using outbound generator custom properties.

You can configure the following outbound generator custom properties for Web Services Security:

com.ibm.wsspi.wssecurity.config.request.setMustUnderstand custom property

The `com.ibm.wsspi.wssecurity.config.request.setMustUnderstand` custom property specifies the mustUnderstand setting in outbound consumer requests. If the value of the property is set to zero (0), no, or false, then the mustUnderstand attribute is not set in the WS-Security header within outbound consumer requests.

Information	Value
Data type	String
Value	Zero (0), no, false
Default	true

In SOAP messages, the default value for the mustUnderstand attribute is zero (0). According to the SOAP specification, if the intended value for the attribute is zero, then the attribute must not be present in the message.

com.ibm.wsspi.wssecurity.config.response.forceMustUnderstandEqualsOne custom property

The `com.ibm.wsspi.wssecurity.config.response.forceMustUnderstandEqualsOne` custom property specifies that the provider should always respond with a `mustUnderstand="1"` attribute in the SOAP security header. If the value is set to one (1), yes, or true, the provider responds with the `mustUnderstand="1"` attribute in the WS-Security header. The default value of the attribute is false.

Information	Value
Data type	String
Value	One (1), yes, or true
Default	false

By default, the response contains the same mustUnderstand attribute as the request. For example, if the inbound request has mustUnderstand="1", the response also includes mustUnderstand="1". If the request does not have a mustUnderstand attribute, the response does not include a mustUnderstand attribute.

For JAX-RPC applications, you can specify both properties in the following locations within the administrative console:

- Click **Servers** > **Server Types** > **WebSphere application servers** > ***server name***. Under Security, click **JAX-WS and JAX-RPC security runtime**. Under JAX-RPC Default Generator Bindings, click **Properties**.
- Click **Servers** > **Server Types** > **WebSphere application servers** > ***server name*** . Under Security, click **JAX-WS and JAX-RPC security runtime**. Under Custom properties, click **Custom properties**.

If you are using an assembly tool with a JAX-RPC WS-Security version 1.0 application, you can set the `com.ibm.wsspi.wssecurity.config.request.setMustUnderstand` custom property on the security request generator extension or binding. You can set the `com.ibm.wsspi.wssecurity.config.response.forceMustUnderstandEqualsOne` custom property on the response generator extension or binding. A setting in the binding takes precedence over a setting in the extension.

If using an assembly tool with a JAX-RPC WS-Security specification draft 13–level application, you can set the `com.ibm.wsspi.wssecurity.config.request.setMustUnderstand` custom property as a parameter on the port qualified name binding. You can set the `com.ibm.wsspi.wssecurity.config.response.forceMustUnderstandEqualsOne` custom property as a parameter on the port component binding.

If using a JAX-WS application , you can set the custom properties as outbound binding properties or parameters on the application using wsadmin scripts. The following property names are used:

- `application.parameters`

- `application.securityoutboundbindingconfig.properties`

However, properties values in the `application.securityoutboundbindingconfig.properties` properties take precedence over properties in application parameters. The follow example shows how to use Jython wsadmin commands to obtain the ID of the policy set attachment for a consumer, then set the `com.ibm.wsspi.wssecurity.config.request.setMustUnderstand` property to false in the outbound binding configuration:

```
AdminTask.getPolicySetAttachments([-applicationName
HelloSvcClientEAR -attachmentType client])

AdminTask.setBinding([-policyType WSSecurity -bindingLocation "[
[application HelloSvcClientEAR] [attachmentId 1490] ]"
-attributes "[[application.securityoutboundbindingconfig.properties_999.name
com.ibm.wsspi.wssecurity.config.request.setMustUnderstand]
[application.securityoutboundbindingconfig.properties_999.value
false]]" -attachmentType client])
```

com.ibm.wsspi.wssecurity.config.token.inbound.retryOnceAfterTrustFailure

The com.ibm.wsspi.wssecurity.config.token.inbound.retryOnceAfterTrustFailure custom property specifies whether a trust store can be reloaded after an application server starts.

A trust store is a key store. By default, JAX-WS WS-Security does not acknowledge the refresh of any keystores while the application server is running. For performance reasons, keystores are cached in memory when each application is started. Because the cache is shared among applications, even if a single application is stopped, its keystores remain in the cache. Therefore, if a trusted certificate, that is used by an X.509 token consumer, is added to a trust store after the application server starts, the trust validation fails.

If you set the com.ibm.wsspi.wssecurity.config.token.inbound.retryOnceAfterTrustFailure property to `true`, when a trust validation occurs, the WS-Security runtime reloads its configured trust store and tries the trust validation one more time. The reloaded trust store is only used for this single re-validation attempt. The keystore object in the cache is not replaced because replacing the keystore object might cause currency issues.

If the second validation attempt fails, a trust validation failure is returned to the client.

The default value for this property is false.

gotcha: This property is set as a custom property on the Callback handler for an X.509, PKIPath, or PKCS#7 token consumer. To set this property in the administrative console, click ***binding_name*** > **WS-Security** > **Authentication and protection** > ***token_name*** > **Callback handler**. For an application using the WS-Security WSS API, this property can also be set on the Callback handler for the token consumers that are previously listed.

com.ibm.wsspi.wssecurity.consumer.timestampRequired

The com.ibm.wsspi.wssecurity.consumer.timestampRequired property specifies whether Timestamp is not expected in the security header for the response when the **Include timestamp in security header** setting is selected for the WS-Security policy.

The JAX-WS WS-Security runtime is updated to comply with the OASIS WS-SecurityPolicy 1.2 specification Timestamp Required requirement. If you want to configure an application to not require an inbound time stamp when an outbound time stamp is configured you can add the com.ibm.wsspi.wssecurity.consumer.timestampRequired custom property to your Web Services Security settings and set that property to `false`. When this property is set to `false`, even if the **Include timestamp in security header** is selected as a setting for the WS-Security policy, a Timestamp is not expected in the security header for a response.

The default value for this property is true.

gotcha: On the custom properties panel, you can set this property as either an inbound or an inbound/outbound custom property. It is not valid as an outbound custom property.

Information	Value
Data type	Boolean
Default	true

com.ibm.wsspi.wssecurity.dsig.inclusiveNamespaces

This custom property, which applies to both the JAX-RPC and JAX-WS applications, specifies whether to disable the inclusive namespace prefix list for XML digital signatures. WebSphere Application Server, by default, includes the prefix in the digital signature for Web Services Security. You can set this custom property to false if you do not want inclusive namespaces set as an element. Some implementations of Web Services Security cannot handle this prefix list. If you experience a signature validation failure when a signed SOAP message is sent and you are using another vendor in your environment, check with your service provider for a possible fix to their implementation before you disable this property.

For JAX-RPC applications, you can set the custom property in the administrative console in the signing information or as a web services security custom property in additional properties or in the default or custom generator bindings. For more information, see the additional properties and generator sections of the Configuring custom properties to secure web services topic. To add the custom property to the signing information, complete the following steps:
1. Click **Applications** > **Enterprise Applications** > ***application_name***.
2. Click **Manage Modules** > ***module_name***.
3. Under Web Services Security Properties, click **Web services: Client security bindings** or **Web services: Server security bindings**.
4. Under Request generator (sender) binding or Response generator (sender) binding, click **Edit custom**.
5. Under Required properties, click **Signing information** > ***signing_information_name*** > **Properties**.
6. Specify the custom property and its value.

For JAX-WS applications, you can configure this custom property in the outbound signing information. To configure the custom property, complete the following steps:
1. Click **Services** > **Service clients** or **Services** > **Service providers**.
2. Click the ***service_name*** > ***binding_name***.
3. Under Policy, click **WS-Security**
4. Under Message Security Policy Bindings, click **Authentication and protection**
5. Under either Request message signature and encryption protection or Response message signature and encryption protection, click the *signature_message_part_reference*. When you click the *signature_message_part_reference* name, you are accessing the configuration for the signed message part binding.
6. Specify the custom property and its value.

com.ibm.wsspi.wssecurity.dsig.oldEnvelopedSignature

Use this property in conjunction with thecom.ibm.wsspi.wssecurity.dsig.enableEnvelopedSignatureProperty JVM custom property to indicate to the WS-Security runtime that you want the WS-Security runtime to verify an XML Digital Signature in the same manner as it did in Versions 7.0.0.21 and earlier. See the topic *Java Virtual Machine (JVM) custom properties* for a description of when you might want to use this JVM custom property.

This property is specified as either an Inbound, Outbound, or Inbound and Outbound custom property for the WS-Security policy set bindings.

com.ibm.wsspi.wssecurity.generator.usewssobject

This custom property determines how the WS-Security run time builds the SOAP Security header that is sent in an outbound SOAP message. By default, the run time uses a fast path using internal web services security (WSS) object representations to build the Security header. Alternatively, the Axis2 run time and objects can be used to build the Security header.

This property is set in the WS-Security policy set bindings as an outbound custom property or an inbound and outbound custom property. This property can be set to `true` or `false`. When this property is set to `true`, WSS Objects are used to build the Security header. When this property is set to `false`, Axis2 objects are used to build the Security header.

When using both WS-Security and WS-Addressing policies for both inbound and outbound messages, a problem might occur where the Body element appears in the header element in the outbound SOAP message. If this error occurs, set the com.ibm.wsspi.wssecurity.generator.useWSSObject custom property to false.

The default value is true.

com.ibm.wsspi.wssecurity.login.useSoap12FaultCodes

The com.ibm.wsspi.wssecurity.login.useSoap12FaultCodes custom property specifies whether the WS-Security runtime is updated to emit the proper SOAP 1.2 fault code when a fault is returned in response to a SOAP 1.2 message.

When this property is set to `true`, the WS-Security runtime is returns a SOAP 1.2 fault code in response to a SOAP 1.2 message.

When this property is set to `false`, the WS-Security runtime returns a SOAP 1.1 fault code in response to a SOAP 1.2 message.

The default value for this property is true.

This property needs to be set as either a WS-Secrutiy **Inbound** or **Inbound and Outbound** custom properties for a specific binding.

Following is an example of a valid SOAP 1.2 fault that is returned when this property is set to true:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv=" http://www.w3.org/2003/05/soap-envelope">
  <soapenv:Body>
    <soapenv:Fault>
      <soapenv:Code>
        <soapenv:Value>soapenv:Sender</soapenv:Value>
        <soapenv:Subcode>
          <soapenv:Value xmlns:axis2ns1="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
            axis2ns1:FailedAuthentication</soapenv:Value>
        </soapenv:Subcode>
      </soapenv:Code>
      <soapenv:Reason>
        <soapenv:Text>CWWSS6521E: The Login failed because
          of an exception: javax.security.auth.login.LoginException:
          CWWSS7062E: Failed to check username [user1] and password in
          the UserRegsitry: WSSUserRegistryProcessor.checkRegistry()=false
        </soapenv:Text>
      </soapenv:Reason>
      <soapenv:Detail></soapenv:Detail>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

com.ibm.wsspi.wssecurity.token.forwardable

When configuring SecurityToken consumer bindings for the JAX-WS programming model, use this custom property to specify whether the receiving token is propagated to other servers. If you specify a value of `true` for this property, you enable this token for propagating to other servers. If you specify a value of `false` for this property, the token is not propagated to other servers. The default value is `true`, and the value is not case sensitive.

com.ibm.wsspi.wssecurity.token.username.addNonce and com.ibm.wsspi.wssecurity.token.username.addTimestamp

When configuring a username token for the JAX-WS programming model, to protect against replay attacks it is strongly recommended that you add custom properties, com.ibm.wsspi.wssecurity.token.username.addNonce and com.ibm.wsspi.wssecurity.token.username.addTimestamp, to the callback handler configuration for token generation. These custom properties enable and verify the nonce and timestamp for message authentication. The value of the properties must be set to `true`.

com.ibm.wsspi.wssecurity.token.username.password.forwardable

When configuring UsernameToken consumer bindings for the JAX-WS programming model, use this custom property to specify whether the password is propagated along with the UsernameToken to other servers during UsernameToken propagation. If you specify a value of `true` for this property, the password is preserved during propagation. If you specify a value of `false` for this property,the password must be removed prior to UsernameToken propagation. The default value is `true`, and the value is not case sensitive.

com.ibm.wsspi.wssecurity.token.username.verifyNonce and com.ibm.wsspi.wssecurity.token.username.verifyTimestamp

When configuring a username token for the JAX-WS programming model, to protect against replay attacks it is strongly recommended that you add custom properties, com.ibm.wsspi.wssecurity.token.username.verifyNonce and com.ibm.wsspi.wssecurity.token.username.verifyTimestamp, to the callback handler configuration for the token consumer. These custom properties enable and verify the nonce and timestamp for message authentication. The value of the properties must be set to `true`.

com.ibm.wsspi.wssecurity.token.UsernameToken.disableUserRegistryCheck

This propery allows the user registry check to be skipped for identity tokens. This means that the user name associated with the identity token in an identity assertion scenario can pass through the UNTConsumeLoginModule without generating a registry error. Typically an identity token must not contain a password, and there might, or might not be a trust token. For example there might be a blind trust.

This property does not affect any UsernameToken that contains a password. If you need to bypass the registry check for a UsernameToken that contains a password, the UNTConsumeLoginModule that is provided with the product cannot be used. The following WS-Security custom property is added to the UNTConsumeLoginModule to allow the user registry check to be skipped for identity tokens:

When the property is set to true, the UNTConsumeLoginModule does not validate the inbound UsernameToken if, and only if, the UsernameToken does not contain a password.

Valid values for this property are true and false. The default value is false.

To configure this property, in the administrative console:

1. Expand **Services** > **Policy sets**.

2. Click **General provider policy set bindings** or **General client policy set bindings**.
3. Click the binding name.
4. Under the Policy heading, click **WS-Security** > **Authentication and Protection** > ***tokenName*** > **Callback Handler**.
5. Add this property and its value in the Custom Properties **Name** and **Value** fields.

com.ibm.wsspi.wssecurity.tokenGenerator.ltpav1.pre.v7

Web services security supports both LTPA (Version 1) and LTPA Version 2 (LTPA2) tokens. The LTPA2 token, which is more secure than Version 1, is supported by the JAX-WS run time only. You can set the **Enforce token version** interoperability option on the token generator to determine whether an LTPA (Version 1) or an LTPA2 token is retrieved when a request message is received. However, if you want to force the run time to use LTPA (Version 1) tokens only, you can set the com.ibm.wsspi.wssecurity.tokenGenerator.ltpav1.pre.v7 custom property to true

To enable this custom property, complete the following steps in the administrative console:
1. Locate the binding that you want to configure.
2. Click the WS-Security policy in the Policies table.
3. Click the Authentication and protection link in the security policy bindings section.
4. Click the token generator that you want to configure.
5. Specify com.ibm.wsspi.wssecurity.tokenGenerator.ltpav1.pre.v7 to true in the Custom properties section.

The following table explains how combinations of this custom property and the **Enforce token version** interoperability option affect the runtime.

Table 138. LTPA interoperability. Table of the com.ibm.wsspi.wssecurity.tokenGenerator.ltpav1.pre.v7 custom property and **Enforce token version** *interoperability option values.*

com.ibm.wsspi.wssecurity.tokenGenerator.ltpav1.pre.v7 custom property value	Enforce token version value	Result
false	Disabled	The run time can use both LTPA (Version 1) and LTPA2 tokens.
not specified, which implies a false value	Disabled	The run time can use both LTPA (Version 1) and LTPA2 tokens.
true	Disabled	The run time can use LTPA (Version 1) tokens only.
true	Enabled	The run time can use LTPA (Version 1) tokens only.

For more information, see the documentation about enabling or disabling single sign-on interoperability mode for the LTPA token.

Web services security generic security token login module custom properties:

When you configure a generic security token login module, you can configure name-value pairs of data, where the name is a property key and the value is a string value that you can use to set internal system configuration properties. You can use these configuration properties, along with the options provided in the administrative console, to control how the token is generated or consumed.

To configure these custom properties for the callback handler in the administrative console, complete the following steps:
1. Expand **Services**.
2. Select **Service provider** or **Service client**
3. Click on the appropriate application in the **Name** column.

4. Click on the appropriate binding in the **Binding** column.

   You must have previously attached a policy set and assigned a binding.

or

1. Expand **Applications** > **Application Types** and click **WebSphere enterprise applications**.
2. Select an application that contains Web services. The application must contain a service provider or a service client.
3. Under the **Web Services Properties** heading, click **Service provider policy sets and bindings** or **Service client policy sets and bindings**.
4. Select a binding. You must have previously attached a policy set and assigned an application-specific binding.

Then complete the following steps:

1. Click **WS-Security** in the Policies table.
2. Under the **Main Message Security Policy Bindings** heading, click **Authentication and protection**.
3. Under the **Authentication tokens** heading, click the name of the authentication token.

   Note: You can use the token, which is processed by the generic security token login module, for authentication only. You cannot use the token as a protection token.
4. Under the **Additional Bindings** heading, click **Callback handler**.
5. Under the **Custom Properties** heading, enter the name and value pairs.

- "Callback handler custom properties for both token generator and token consumer bindings"
- "Callback handler custom properties for token generator bindings" on page 688
- "Callback handler custom properties for token consumer bindings" on page 689

Callback handler custom properties for both token generator and token consumer bindings

The following table lists the callback handler custom properties that can be used to configure both token generator and token consumer bindings.

Table 139. Callback handler custom properties for both token generator and token consumer bindings.. This table contains the custom property name, its values, and a short description.

Name	Values	Description
`clockSkew`	This custom property does not have a default value.	Use this custom property to specify, in minutes, an adjustment to the times in the self-issued SAML token that the SAMLGenerateLoginModule creates. The clockSkew custom property is set on the Callback handler of the SAML token generator that uses the SAMLGenerateLoginModule class. The value specified for this custom property must be numeric and is specified in minutes. When a value is specified for this custom property, the following time adjustments are made in the self-issued SAML token that the SAMLGenerateLoginModule creates: • The new NotBefore time setting equals the initial NotBefore time setting, minus the amount of time specified for the clockSkew custom property. • The new NotAfter time setting equals the initial NotAfter time setting, plus the amount of time specified for the clockSkew custom property.

Table 139. Callback handler custom properties for both token generator and token consumer bindings. (continued). This table contains the custom property name, its values, and a short description.

Name	Values	Description
`stsURI`	This custom property does not have a default value.	Use this custom property to specify the Security Token Service (STS) address. This custom property is required for the token consumer. However, this custom property is optional for the token generator if the requested token exists in the RunAs Subject and its verification is not required.
`wstrustClientBinding`	This custom property does not have a default value.	Use this custom property to specify the binding name for the WS-Trust client.
`wstrustClientBindingScope`	You can specify an `application` or `domain` value.	Use this custom property to specify the type of bindings that are used for the WS-Trust client. The following conditions apply: • If you specify the `domain` value, general bindings are used. • If you specify the `application` value, custom bindings are used. • If you do not specify a value and application bindings exist, those application bindings are used. • If you do not specify a value and general bindings exist, those general bindings are used. • If neither application or general bindings exist, the default bindings are used. This custom property is optional.
`wstrustClientPolicy`	This custom property does not have a default value.	Use this custom property to specify the policy set name for the WS-Trust client.
`wstrustClientSoapVersion`	You can specify a `1.1` or `1.2` value.	Use this custom property to specify the SOAP message version that the trust client uses to generate the SOAP message. The SOAP message is sent to the Security Token Service (STS). If you do not define this custom property, the generic security token login module uses the SOAP version of the application when it generates the SOAP message for the trust client request. The default value corresponds to the SOAP version that is used by the application client. This custom property is optional.
`wstrustClientWSTNamespace`	Specify one of the following values: **Trust Version 1.3 (Default)** Specify `1.3` to use Trust Version 1.3 (Default).`http://docs.oasis-open.org/ws-sx/ws-trust/200512` **Trust Version 1.2** Specify `1.2` to use Trust Version 1.2.`http://schemas.xmlsoap.org/ws/2005/02/trust`	Use this custom property to specify which trust client namespace the generic security token login modules uses when it makes the WS-Trust request.
`wstrustValidateClientBinding`	By default, the value for this custom property is the same value that is specified for the `wstrustClientBinding` custom property.	Use this custom property to specify the bindings that are used by the WS-Trust Validate request. If you do not specify this custom property, the WS-Trust Validate request uses the same bindings that are used by WS-Trust Issue, which are defined by the `wstrustClientBinding` custom property.

Table 139. Callback handler custom properties for both token generator and token consumer bindings. (continued). This table contains the custom property name, its values, and a short description.

Name	Values	Description
`wstrustValidateClientPolicy`	By default, the value for this custom property is the same value that is specified for the `wstrustClientPolicy` custom property.	Use this custom property to specify the policy sets to use with the WS-Trust Validate request. If you do not specify a value for this custom property, WS-Trust Validate uses the same policy set as WS-Trust Issue, which is defined by the required `wstrustClientPolicy` custom property.
`wstrustIssuer`	You can use any string value.	Use this custom property to specify the issuer for the request token. This custom property is optional
`wstrustValidateTargetOption`	The default value is the WS-Trust `Base` element extension. You can specify a `token` value or a `base` value, which is also the default value.	Use this custom property to specify whether the WS-Trust client passes the validation token to the WS-Trust Security Token Service using the ValidateTarget or the Base element extension. The following conditions apply: • If you do not specify a value for this custom property, the token is wrapped in the Base element extension within the RequestedSecurityToken element. • If you specify the `token` value, the token is wrapped in the ValidateTarget element within the RequestedSecurityToken element.

Callback handler custom properties for token generator bindings

The following table lists the callback handler custom properties that can only be used to configure token generator bindings.

Table 140. Callback handler custom properties for token generator bindings only.. This table contains the custom property name, its values, and a short description.

Name	Value	Description
`useRunAsSubject`	You can use a `True` or `False` value. By default, a `True` value is used. This value for this custom property is not case sensitive.	Use this custom property to specify whether the generic security token login modules use the token from the RunAs Subject for the outgoing request. By default, the login module uses the validated tokens in the RunAs Subject first. The following conditions apply: • If you set this custom property to a `false` value, the generic security token login module does not use WS-Trust Validate to exchange the token for the outbound request. Instead, it uses WS-Trust Issue to request a token. • If you do not specify this custom property, the generic security token login module attempts to use a token from the RunAs Subject and WS-Trust Validate to exchange the token. • If a token does not exist in the RunAs Subject, the generic security token login module uses WS-Trust Issue and is protected by the trust client policy sets.
`useRunAsSubjectOnly`	You can use a `True` or `False` value. By default, a `False` value is used. This value for this custom property is not case sensitive.	Use this custom property to disable or enable WS-Trust Issue in the generic security token login module. If you set this custom property to a `true` value, the generic security token login module uses the token from the RunAs Subject and WS-Trust Validate to exchange the tokens. The generic security token login module does not use WS-Trust Issue to request a token even if WS-Trust Validate fails or it does not find a matching token in the RunAs Subject.

Table 140. Callback handler custom properties for token generator bindings only. (continued). This table contains the custom property name, its values, and a short description.

Name	Value	Description
`useToken`	You can use any string value of the ValueType value for the security token.	When you use a security token in a RunAs Subject to validate and exchange tokens for an outbound request, you can use this custom property to specify which token ValueType value in the RunAs Subject to validate and exchange for the requested token. For example, you might have a token with a ValueType value of *Token_1* in the RunAs Subject. However, the ValueType value of *Token_2* is the required token. You can set this custom property to *Token_1* . If you do not define this custom property, the validation token is the token from the RunAs Subject that has the same ValueType value as the required token. This custom property is optional.
`validateUseToken`	You can use a `True` or `False` value. By default, a `True` value is used. This value for this custom property is not case sensitive.	Use this custom property to specify whether the token generator uses WS-Trust Validate to validate the token from the RunAs Subject. By default, the generic security token login module validates a token from the RunAs Subject against the Security Token Service (STS) before sending the token in the SOAP message to the service provider. If you set this custom property value to `false` and the generic security token login module finds a matching token from the RunAs Subject, the login module does not invoke WS-Trust Validate to validate the matching token. Instead, it sends the matching token to the downstream service provider without validation.
`wstrustIncludeTokenType`	You can use a `True` or `False` value. By default, a `True` value is used. This value for this custom property is not case sensitive.	Use this custom property to specify whether the WS-Trust RequestedSecurityToken token includes the requested token ValueType value. If you do not specify this custom property, the generic security token login modules includes the requested token type in the WS-Trust RequestedSecurityToken token. This custom property is optional.

Callback handler custom properties for token consumer bindings

The following table lists the callback handler custom properties that can only be used to configure token consumer bindings.

Table 141. Callback handler custom properties for token consumer bindings only.. This table contains the custom property name, its values, and a short description.

Name	Value	Description
`exchangedTokenType`	The valid value for this custom property is the string ValueType value for the token that is supported by the system default login modules.	Use this custom property to specify the new token with the defined ValueType value, which the trust service must return after successful validation. If you do not specify a value for the custom property, the generic security token login module accepts whichever token the trust service returns. This custom property is optional.

Web services security SAML token custom properties:

When you configure a web services security SAML token, you can configure name-value pairs of data, where the name is a property key and the value is a string value that you can use to set internal system configuration properties. You can use these configuration properties, along with the options provided in the administrative console, to control how the SAML token is generated or consumed.

To configure these SAML custom properties, in the administrative console, either:

1. Expand **Services**.
2. Select **Service provider** or **Service client**
3. Click on the appropriate application in the **Name** column.
4. Click on the appropriate binding in the **Binding** column.

   You must have previously attached a policy set and assigned a binding.

or

1. Expand **Applications** > **Application Types** and click **WebSphere enterprise applications**.
2. Select an application that contains Web services. The application must contain a service provider or a service client.
3. Under the **Web Services Properties** heading, click **Service provider policy sets and bindings** or **Service client policy sets and bindings**.
4. Select a binding. You must have previously attached a policy set and assigned an application-specific binding.

Then complete the following steps:

1. Click **WS-Security** in the Policies table.
2. Under the **Main Message Security Policy Bindings** heading, click **Authentication and protection**.
3. Under the **Authentication tokens** heading, click the name of the authentication token.

   Note: You can use the token, which is processed by the generic security token login module, for authentication only. You cannot use the token as a protection token.
4. Under the **Additional Bindings** heading, click **Callback handler**.
5. Under the **Custom Properties** heading, enter the name and value pairs.

The following sections list the custom properties and indicate how each custom property is used.

- "SAML token generator custom properties"
- "SAML token consumer custom properties" on page 692
- "SAML token custom properties for both token generator and token consumer" on page 693
- "Trust client custom properties" on page 694

SAML token generator custom properties

The following table lists the callback handler custom properties that can only be used to configure SAML token generator bindings.

Table 142. SAML token callback handler custom properties for token generator bindings only.. This table contains the custom property name, its values, and a short description.

Name	Values	Description
`WSSGenerationContext`	This custom property does not have a default value.	Use this custom property to specify the `WSSGenerationContext` object that the WS-Trust client uses to request a SAML token.

Table 142. SAML token callback handler custom properties for token generator bindings only. (continued). This table contains the custom property name, its values, and a short description.

Name	Values	Description
WSSConsumingContext	This custom property does not have a default value.	Use this custom property to specify the WSSConsumingContext object that the WS-Trust client uses to request a SAML token.
com.ibm.wsspi.wssecurity.saml.put.SamlToken	This custom property does not have a default value.	Use this custom property to set the SAML token to RequestContext.
com.ibm.wsspi.wssecurity.saml.get.SamlToken	This custom property does not have a default value.	Use this custom property to get the SAML token to RequestContext.
stsURI	This custom property does not have a default value.	Use this custom property to specify the SecurityTokenService address.
keySize	This custom property does not have a default value.	Use this custom property to specify the KeySize when requesting a SecretKey from STS.
tokenRequest	Valid values include issue, propagation, and issueByWSPrincipal. The default value is issue.	Use this custom property to specify the SAMLToken request method.
confirmationMethod	Valid values include bearer, holder-of-key, and sender-vouches. This custom property does not have a default value.	Use this custom property to specify a SAML token subject ConfirmationMethod.
signToken	This custom property does not have a default value.	Use this custom property to specify whether a SAML token should be signed with an application message.
usekeyType	This custom property is optional. The valid values are KeyValue, X509Certificate, and X509IssuerSerial.	Use this custom property to specify the Usekey type, which tells the client to generate a specific type of key Information.
cacheCushion	The default value is 5 minutes.	Use this custom property to specify, in minutes, the amount of time during which a cached token should not be reused, and a new token should be issued.
cacheToken	Valid values are true and false. The default behavior is true, which allows SAML token caching for reuse.	Use this custom property to specify whether a SAML token can be cached for reuse.
com.ibm.wsspi.wssecurity.saml.client.SamlTokenCacheTimeout	The default value is 60 minutes.	Use this JVM custom property to specify, in minutes, the length of time a SAML token could be maintained in a client cache.
com.ibm.wsspi.wssecurity.saml.client.SamlTokenCacheEntries	The default value is 250.	Use this JVM custom property to specify the maximum number of cache entries that can be maintained.
recipientAlias	This custom property does not have a default value.	Use this custom property to specify a target service alias for a certificate.
failOverToTokenRequest	Valid values are true or false. The default value is true, which means that the web services security runtime always issues a new SAML token if the input token is invalid.	Use this custom property to specify whether the web services security runtime should use the attached policy set to issue a new SAML token if the input SAML token in the RequestContext is invalid.
tokenType	This custom property does not have a default value.	Use this custom property to set the required token type to SAMLGenerateCallback
appliesTo	This custom property does not have a default value.	Use this custom property to specify the AppliesTo for the requested SAML token when a WSS API is used.
com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath	This custom property does not have a default value.	Use this custom property to specify the required configuration data when generating a self-issued SAML token.
sslConfigAlias	If a value is not specified for this property, the default SSL alias defined in your system's SSL configuration is used. This property is optional.	Use this custom property to specify the alias to an SSL configuration that a WS-Trust client uses to request a SAML token.
com.ibm.wsspi.wssecurity.saml.config.issuer.IssuerURI	This custom property does not have a default value.	Use this custom property to specify the issuer URL in the custom properties.
com.ibm.wsspi.wssecurity.saml.config.issuer.TimeToLiveMilliseconds	The default value is 3600000 (60 minutes).	Use this custom property to specify, in milliseconds, the amount of time that can elapse before a token expires in the custom properties.
com.ibm.wsspi.wssecurity.saml.config.issuer.KeyStoreRef	This custom property does not have a default value.	Use this custom property to specify a reference to a centrally managed keystore in the custom properties.

SAML token consumer custom properties

The following table lists the callback handler custom properties that can only be used to configure SAML token consumer bindings.

Table 143. SAML token callback handler custom properties for token consumer bindings only.. This table contains the custom property name, its values, and a short description.

Name	Values	Description
trustStoreRef	This custom property does not have a default value.	Use this custom property to specify the truststore reference for a SAML consumer token.
trustStorePath	This custom property does not have a default value.	Use this custom property to specify the truststore file path for a SAML consumer token.
trustStoreType	This custom property does not have a default value.	Use this custom property to specify the truststore type name for a SAML consumer token
trustStorePassword	This custom property does not have a default value.	Use this custom property to specify the truststore password for a SAML consumer token.
trustedAlias	This custom property does not have a default value.	Use this custom property to specify the trusted STS certificate's alias for a SAML consumer token.
TtrustAnySigner	The default value is `false`.	Use this custom property to specify whether a recipient can trust any certificate that signs a SAML assertion.
signatureRequired	The default value is `true`.	Use this custom property to specify whether a signature is required on a SAML assertion.
keyStoreRef	This custom property does not have a default value.	Use this custom property to specify the keystore reference for a SAML consumer token.
keyStorePath	This custom property does not have a default value.	Use this custom property to specify the keystore file path for a SAML consumer token.
keyStoreType	This custom property does not have a default value.	Use this custom property to specify the keystore type for a SAML consumer token.
keyStorePassword	This custom property does not have a default value.	Use this custom property to specify the keystore password for a SAML consumer token
keyAlias	This custom property does not have a default value.	Use this custom property to specify the key alias for a SAML consumer token.
keyName	This custom property does not have a default value.	Use this custom property to specify the key name for a SAML consumer token.
keyPassword	This custom property does not have a default value.	Use this custom property to specify the key password for a SAML consumer token.
validateOneTimeUse	Valid values are `true` or `false`. The default value is `true`, which means that OneTimeUse assertion validation is required.	Use this custom property to specify whether a OneTimeUse assertion in SAML 2.0, or a DoNotCacheCondition in SAML 1.1 must be validated.
validateAudienceRestriction	Valid values are `true` or `false`. The default value is `false` which means that an AudienceRestriction assertion validation is not required.	Use this custom property specify whether an an AudienceRestriction assertion must be validated.
trustedIssuer_	The name is specified as `trustedIssuer_n` where *n* is an integer. This custom property does not have a default value.	Use this custom property to specify the name of a trusted issuer.
trustedSubjectDN_	The value specified must be in the format *trustedSubjectDN_n*, where *n* is an integer. This custom property does not have a default value.	Use this custom property to specify the X509Certificate's SubjectDN name for the trusted issuer.
X509PATH	This custom property does not have a default value.	Use this custom property to specify the intermediate X509Certificate file path for a SAML consumer token.
CRLPATH	This custom property does not have a default value.	Use this custom property to specify the file path to the list of revoked certificates for a SAML consumer token.
X509PATH_	The value specified must be in the format *X509_path_n*, where *n* is an integer. This custom property does not have a default value.	Use this custom property to specify the file path for the intermediate X509 certificate for a SAML consumer token.

Table 143. SAML token callback handler custom properties for token consumer bindings only. (continued). This table contains the custom property name, its values, and a short description.

Name	Values	Description
`CRLPATH_`	The value specified must be in the format *trustedSubjectDN_n*, where *n* is an integer. This custom property does not have a default value.	Use this custom property to specify the file path to the list of revoked X509 certificates for a SAML consumer token.
`com.ibm.wsspi.wssecurity.saml.signature.SignatureCacheTimeout`	An integer. The default value is `60 minutes`.	Use this custom property to specify how many minutes a SAML token is to be cached. A signature validation does not need to be repeated while the SAML token is cached.
`com.ibm.wsspi.wssecurity.saml.signature.SignatureCacheEntries`	An integer. The default value is `1000`.	Use this custom property to specify how many signature cache entries can be maintained. for a SAML consumer token.

SAML token custom properties for both token generator and token consumer

The following table lists the callback handler custom properties that can be used to configure both SAML token generator and SAML token consumer bindings.

Table 144. SAML token callback handler custom properties for token generator and token consumer bindings.. This table contains the custom property name, its values, and a short description.

Name	Values	Description
`clockSkew`	The default value is `5 minutes`.	Use this custom property to specify, in minutes, an adjustment to the times in the self-issued SAML token that the SAMLGenerateLoginModule creates. The clockSkew custom property is set on the Callback handler of the SAML token generator that uses the SAMLGenerateLoginModule class. The value specified for this custom property must be numeric and is specified in minutes. When a value is specified for this custom property, the following time adjustments are made in the self-issued SAML token that the SAMLGenerateLoginModule creates: • The new NotBefore time setting equals the initial NotBefore time setting, minus the amount of time specified for the clockSkew custom property. • The new NotAfter time setting equals the initial NotAfter time setting, plus the amount of time specified for the clockSkew custom property.
`requireDKT`	The default value is `false`.	Use this custom property to specify an option for the derived keys whenever a WSS API is used with the requested SAML token.
`useImpliedDKT`	The default value is `false`.	Use this custom property to specify an option that is used with Implied derived keys whenever a WSS API is used with the requested SAML token.
`nonceLength`	The default value is `128`.	Use this custom property to specify, in bytes, the derived nonce length to use for the derived keys whenever a WSS API is used with the requested SAML token.
`keylength`	This custom property does not have a default value.	Use this custom property to specify, in bytes, the derived key length to use for the derived keys whenever a WSS API is used with the requested SAML token.
`clientLabel`	This custom property does not have a default value.	Use this custom property to specify, in bytes, the client label to use for the derived keys whenever a WSS API is used with the requested SAML token.

Table 144. SAML token callback handler custom properties for token generator and token consumer bindings. (continued). This table contains the custom property name, its values, and a short description.

Name	Values	Description
serviceLabel	This custom property does not have a default value.	Use this custom property to specify, in bytes, the service label to use for the derived keys whenever a WSS API is used with the requested SAML token.

Trust client custom properties

The following table lists the callback handler custom properties that can be used to configure trust client generator bindings.

Table 145. Callback handler custom properties for the trust client generator bindings.. This table contains the custom property name, its values, and a short description.

Name	Values	Description
wstrustClientPolicy	This custom property does not have a default value.	Use this custom property to specify the policy set name for a WS-Trust client.
keyType	The following keyTypes can be specified for WS-Trust 1.2: • com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.WST12#KEYTYPE_PUBLICKEY • com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.WST12#KEYTYPE_SYMMETRICKEY The following keyTypes can be specified for WS-Trust 1.3: • com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.WST13#KEYTYPE_PUBLICKEY • com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.WST13#KEYTYPE_SYMMETRICKEY • com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.WST13#KEYTYPE_BEARER	Use this custom property to specify the keyType when making a WS-Trust request to STS.
wstrustClientBinding	This custom property does not have a default value.	Use this custom property to specify a binding name for the WS-trust client.
wstrustClientSoapVersion	Valid values are 1.1 and 1.2. If no value is specified, the SOAP version defaults to SOAP version that the application client is using.	Use this custom property to specify the SOAP version in a WS-Trust request.
TwstrustClientBindingScope	This custom property does not have a default value.	Use this custom property to specify the binding scope for the policy set that is attached to the WS-Trust client.
wstrustClientWSTNamespace	The default value is trust 13. Valid values are trust12 and trust 13.	Use this custom property to specify the WS-Trust namespace for a WS-Trust request.
com.ibm.wsspi.wssecurity.trust.client.TrustSer[illegible]	The default value is 60 minutes.	Use this custom property to specify, in minutes, the length of time an STS service instance can be kept in a client side cache.
com.ibm.wsspi.wssecurity.trust.client.TrustSer[illegible]	The default value is 1000.	Use this custom property to specify the maximum number of STS service instance cache entries that can be maintained.
wstrustClientCollectionRequest	Valid values are true or false. The default value is false which means that a RequestSecurityToken is used instead of a RequestSecurityTokenCollection.	Use this custom property to specify whether a RequestSecurityTokenCollection is required in a WS-Trust request.

Administering message-level security for JAX-WS web services

Web Services Security standards and profiles describe how to provide security and protection for SOAP messages that are exchanged in a web services environment. Using JAX-WS, development of web services and clients is simplified with greater platform independence for Java applications through the use of dynamic proxies and Java annotations.

Auditing the Web Services Security runtime:

Security auditing provides tracking and archiving of auditable events for the web services runtime operations. When security auditing is enabled for web services, the event generator utility collects and logs signing, encryption, security, authentication, and delegation events in audit event records. You can analyze the audit event records to identify possible security breaches or potential weaknesses in the security configuration of your environment.

The audit security subsystem must be enabled for WebSphere Application Server before the event generator can collect auditing records for the Web Services Security runtime. The recording of auditable security events is achieved by enabled the security auditing subsystem. For more information about enabling security auditing, read the topic "Enabling the security auditing subsystem."

Web Services Security auditing is enabled for the JAX-WS runtime only. Several auditing events can occur when a SOAP message is received. Auditing data is collected during various Web Services Security runtime operations, such as validating the digital signature of a SOAP message, decrypting the message, or checking the message security header. The auditing data is stored and managed by the event generator, and stored in audit logs for later analysis.

Auditable events for web services include:

Signing

As the digital signature in each SOAP message part is validated, a SECURITY_SIGNING event is sent to the event generator, along with an outcome, which can be SUCCESS or ERROR. Reason codes, either VALID_SIGNATURE or INVALID_SIGNATURE, are also sent. The integrity of the message is audited, also with a SECURITY_SIGNING event, with the outcome of SUCCESS or DENIED. Reason codes are INTEGRITY or INTEGRITY_BAD. The following table summarizes the signing events:

Table 146. Signing audit events. Use the signing audit event records to identify possible security breaches or weaknesses in the security configuration.

Event type	Possible outcomes	Reason codes
SECURITY_SIGNING (digital signature)	SUCCESS ERROR	VALID_SIGNATURE INVALID_SIGNATURE
SECURITY_SIGNING (integrity)	SUCCESS DENIED	INTEGRITY INTEGRITY_BAD

If the SECURITY_SIGNING event outcome is DENIED, and the reason code is INTEGRITY_BAD, this means that at least one message part, which must be signed by the security policy, failed signature validation. If the SECURITY_SIGNING event outcome is ERROR, and the reason code is INVALID_SIGNATURE, this means that the digital signature validation failed. This could lead to an INTEGRITY_BAD reason code in the audit record, depending on whether the digital signature is required by the security policy.

Encryption

Encryption auditing is performed in two parts: first, the encrypted parts of the SOAP message are processed, then the confidentiality of the message is audited. To audit each encrypted part of the message, a SECURITY_ENCRYPTION event is sent. An event record is created only if the outcome of the event is ERROR, meaning that an exception is encountered. The reason code is DECRYPTION_ERROR. Next, the confidentiality of the message is checked with another SECURITY_ENCRYPTION event, which has possible outcomes of SUCCESS or DENIED. Reason codes for this event are CONFIDENTIALITY or CONFIDENTIALITY_BAD. The following table summarizes the encryption events:

Table 147. Encryption audit events. Use the encryption audit event records to identify possible security breaches or weaknesses in the security configuration.

Event type	Possible outcomes	Reason codes
SECURITY_ENCRYPTION (encrypted message parts)	ERROR	DECRYPTION_ERROR
SECURITY_ENCRYPTION (confidentiality)	SUCCESS DENIED	CONFIDENTIALITY CONFIDENTIALITY_BAD

If the SECURITY_ENCRYPTION event outcome is DENIED and the reason code is CONFIDENTIALITY_BAD, this means that at least one message part, which is required to be encrypted, is not correctly encrypted. A DECRYPTION_ERROR in the audit record could lead to a CONFIDENTIALITY_BAD reason code, depending on whether message encryption is required.

Time stamp

For each SOAP message, the time stamp is audited when a SECURITY_RESOURCE_ACCESS event is sent to the event generator. The possible outcomes of this event are SUCCESS or DENIED. Reason codes are TIMESTAMP or TIMESTAMP_BAD. The following table summarizes the time stamp events:

Table 148. Time stamp audit event. Use the time stamp audit event record to identify possible security breaches or weaknesses in the security configuration.

Event type	Possible outcome	Reason code
SECURITY_RESOURCE_ACCESS	SUCCESS DENIED	TIMESTAMP TIMESTAMP_BAD

Security header

The security header is audited to make sure it is not missing from the SOAP message. A SECURITY_RESOURCE_ACCESS event is sent, and if the header is missing, the outcome is DENIED, with the reason code SECURITY_HEADER_MISSING. The following table summarizes the security header event:

Table 149. Security header audit event. Use the security audit event record to identify possible security breaches or weaknesses in the security configuration.

Event type	Possible outcome	Reason code
SECURITY_RESOURCE_ACCESS	DENIED	SECURITY_HEADER_MISSING

Authentication

The security token used to authenticate a message is audited to determine if authentication is successful, and information about the authentication type, provider name and provider status are saved in the audit event record. The token ID is also recorded, along with information that is specific to the token type, such as username or keystore. Sensitive information such as the token itself, or the token password, is not recorded in the audit event record. Information recorded for each token type is described in the following table:

Table 150. Token information recorded in audit event record. Use the authentication audit event records to identify possible security breaches or weaknesses in the security configuration.

Token type	Recorded information
Username token	Username Password (null or not-null) Token ID
LTPA	Principal Expiration Token ID
LTPAPropagate	Principal Expiration Token ID
SecureConversation	UUID Instance UUID Token ID
DerivedKey	Reference ID Token ID
Kerberos	Principal SPN pSHA1 Token ID

Table 150. Token information recorded in audit event record (continued). Use the authentication audit event records to identify possible security breaches or weaknesses in the security configuration.

Token type	Recorded information
X509	Certificate Subject Issuer Keystore Token ID TrustAny
PKP Path (public key infrastructure)	Certificate Subject Issuer Keystore Token ID TrustAny
PKCS7 (Public Key Cryptography Standards)	Certificate Subject Issuer Keystore Token ID TrustAny
SAML token	Principal SAML Token Issuer Name Confirmation Method

Message authentication is audited using a SECURITY_AUTHN event. Possible outcomes of the event are SUCCESS, DENIED or FAILURE. Reason codes are AUTHN_SUCCESS, AUTHN_LOGIN_EXCEPTION or AUTHN_PRIVILEDGE_ACTION_EXCEPTION. The following table summarizes the authentication events:

Table 151. Authentication audit event. Use the authentication audit event records to identify possible security breaches or weaknesses in the security configuration.

Event type	Possible outcomes	Reason codes
SECURITY_AUTHN	SUCCESS DENIED FAILURE	AUTHN_SUCCESS AUTHN_LOGIN_EXCEPTION AUTHN_PRIVILEDGE_ACTION_EXCEPTION

Delegation

Authentication of a SOAP message can be delegated, and the delegation function is audited using a SECURITY_AUTHN_DELEGATION event. This event is sent when the client identity is propagated or when delegation involves the use of a special identity. In the Web Services Security runtime, auditing events are recorded only when the delegation type is set to Identity Assertion. Possible outcomes are SUCCESS or DENIED, with reason codes AUTHN_SUCCESS or AUTHN_DENIED. Additional information, such as delegation type, role name and identity name, is collected and stored in the audit event record. The following table summarizes the delegation events:

Table 152. Delegation audit event. Use the delegation audit event record to identify possible security breaches or weaknesses in the security configuration.

Event type	Possible outcomes	Reason codes
SECURITY_AUTHN_DELEGATION	SUCCESS DENIED	AUTHN_SUCCESS AUTHN_DENIED

Validation

As part of the generic security token login module support, a Security Token Service can validate a token using a WS-Trust Validate request. This validation operation can return a token in exchange for a token that is being validated. The information that is exchanged is located in the documentation on auditing for a generic security token. The following table shows the token exchange information that is recorded.

Table 153. Token exchange. Use the token exchange information to trace the token exchange process.

Event	Description
TokenSentForExchangeId	This information identifies the token that is sent for validation and is exchanged.
TokenSentForExchangeType	This information identifies the type of the token that is sent for validation and is exchanged.
ExchangedTokenId	This information identifies the token that is received in exchange during the validation request.
ExchangedTokenType	This information identifies the type of the token that is received in exchange during the validation request.

If the token is validated without a token exchange, only the Token ID is recorded.

Securing web services using policy sets:

Policy sets are assertions about how services are defined. They are used to simplify the quality of service configuration for web services.

About this task

Policy sets combine configuration settings, including those for transport and message level configuration, such as WS-Addressing, WS-ReliableMessaging, and WS-Security. There are two main types of policy sets; application policy sets and system policy sets. Application policy sets are used for business-related assertions. These assertions are related to the business operations that are defined in the Web Services Description Language (WSDL) file. System policy sets, on the other hand, are used for non-business-related system messages. These messages are not related to the business operations that are defined in the WSDL, but instead refer to messages that are defined in other specifications which apply qualities of service (QoS). Such QoS are the request security token (RST) messages that are defined in WS-Trust, or create sequence messages that are defined in WS-Reliable Messaging metadata exchange messages of the WS-MetadataExchange.

Note: You can use policy sets only with Java™ API for XML-Based Web Services (JAX-WS) or Service Component Architecture (SCA) applications. You cannot use policy sets with Java API for XML-based RPC (JAX-RPC) applications.

Policies are defined based on a quality of service. Policy definition is typically based on WS-Policy standard language, for example, the WS-Security policy is based on the current WS-SecurityPolicy from the Organization for the Advancement of Structured Information Standards (OASIS) standards.

Policy sets do not include environment or platform-specific information, such as keys for signing, keystore information, or persistent store information. This type of information is defined in the binding. A policy set attachment defines how a policy set is attached to service resources and bindings. The attachment definition is outside the policy set definition and is defined as meta-data associated with application data.

To secure JAX-WS web services with message-level security using policy sets, follow these steps:

Procedure

1. Select, create, or copy and modify a policy set to specify the message-level protection required. The policy specifies what protection will be applied, for example, what message parts to sign or encrypt and the token types and algorithms to use.
   - Select one of the web services policy sets.
   - Create, copy, modify, import, export or delete a policy set. For more information, read about managing policy sets using the administrative console
2. Attach the policy set to the application.

3. Create or select the policy set bindings to be used. The bindings are then attached to the application along with the policy set. The bindings used can either be general bindings that can be shared among applications or application specific bindings. For more information, read about defining and managing policy set bindings.
4. If WS-SecureConversation is being used, specify the trust service system policy sets and bindings on the application server.

Configuring a policy set and bindings for Asymmetric XML Digital Signature and/or XML Encryption:

This procedure describes how to configure the message-level WS-Security policy set and bindings to sign and encrypt a SOAP message using asymmetric XML Digital Signature and Encryption. As part of this procedure you must specify whether you will sign and/or encrypt both the request and response messages.

Before you begin

This task assumes that the service provider and client that you are configuring are in the JaxWSServicesSamples application. Refer to the topic *Accessing Samples* for more information on how to obtain and install this application.

You should use the following trace specification on your server. These specifications enable you to debug any future configuration problems that might occur.

```
*=info:com.ibm.wsspi.wssecurity.*=all:com.ibm.ws.webservices.wssecurity.*=all:
com.ibm.ws.wssecurity.*=all: com.ibm.xml.soapsec.*=all: com.ibm.ws.webservices.trace.*=all:
com.ibm.ws.websvcs.trace.*=all:com.ibm.ws.wssecurity.platform.audit.*=off:
```

About this task

This procedure explains the actions you need to complete to configure a WS-Security policy set to use the asymmetric XML-Digital Signature and Encryption WS-Security constraints. This procedure also explains the actions you need to complete to configure asymmetric XML Digital Signature and Encryption application specific custom bindings for a client and provider.

The keystores that are used in this procedure are provided with WebSphere Application Server and are installed in every profile that is created. You can use the ${USER_INSTALL_ROOT} variable directly in the configuration to conveniently point to the keystore locations without using a fully qualified path. ${USER_INSTALL_ROOT} resolves to a path such as c:/WebSphere/AppServer/profiles/AppSrv01.

```
${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks
${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks
${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks
```

Because of the nature of JaxWSServicesSamples, to apply the policy set and bindings to this application, in the administrative console click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples**. When using your own applications, you can use the following paths as an alternative way to access the provider and client for attachment of the policy set and bindings:

```
* Services > Service Providers > (AppName)
* Services > Service clients > (AppName)
```

gotcha: Pay close attention to the names of the token consumers and generators in the administrative console. The Initiator and recipient might not be what you think they should be for the tokens. The usage column in the table specifies whether a token is a consumer token or a generator token.

Procedure

1. Create the custom policy set.
   a. In the administrative console, click **Services > Policy sets > Application Policy sets**.

b. Click **New**.
c. Specify `Name=`*`AsignEncPolicy`*.
d. Click **Apply**.
e. Under Policies, click **Add > WS-Security**.

2. Edit the custom policy set.
   a. In the administrative console, click **WS-Security > Main Policy**.

      By default, the policy will now have the following configuration:
      - Timestamp sent in outbound messages
      - Timestamp required in inbound messages
      - Sign the request and the response (Body, WS-Addressing header, and Timestamp)
      - Encrypt the request and the response (Body and Signature element in SOAP Security header)

      If this is the configuration that you want, click **Apply**, then **Save**, and continue to the next step.

      If you want to change this configuration, complete one or more of the following substeps.
   b. Optional: Remove Timestamp from both request and response. You cannot do one-way Timestamp.

      To remove Timestamp from both request and response, unselect the **Include timestamp in security header** setting, and then click **Apply**.
   c. Optional: Remove request message parts.
      1) Under Message level protection, click **Request message part protection**.
      2) To remove the request encrypted part, click *app_encparts*, and then click **Delete**.
      3) To remove the request signed part, click *app_signparts*, and then click **Delete**.
      4) Click **Done**.
   d. Optional: Remove response message parts.
      1) Under Message level protection, click **Response message part protection**.
      2) To remove the response encrypted part, click *app_encparts*, and then click **Delete**.
      3) To remove the response signed part, click *app_signparts*, and then click **Delete**.
      4) Click **Done**.
   e. Optional: View or change parts that are being signed or encrypted in the request.
      1) Under Message level protection, click **Request message part protection**.
      2) To view or change the request encrypted part, click *app_encparts*, and then click **Edit**.

         The **Elements in Part** page displays with the parts that will be encrypted in the request message. You can update the settings on this page to add, change, or remove elements to encrypt. By default, the **Body** and an **XPath expression to the Signature** are configured.

         If you would like to add encryption of a UsernameToken, add the following XPath expression:

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-name()='Envelope']
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-name()='Header']
/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd' and loca
/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd' and loca
```

         When you finish making your changes, click **OK**.
      3) To view or change the request signed part, click *app_signparts*, and then click **Edit**.

         The **Elements in Part** page displays with the parts that will be signed in the request message. You can update the settings on this page to add, change, or remove elements to sign. By default, the **Body**, the **QNames for the WS-Addressing header**, and **XPath expressions to the Timestamp** are configured.

         If you will be using the STR Dereference Transform (STR-Transform) to sign a security token, add the following XPath expression:

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-name()='Envelope']
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and local-name()='Header']
/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd' and l
/*[namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and local-name()='Signature']
/*[namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and local-name()='KeyInfo']
/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd' and l
```

When you finish making your changes, click **OK**.

4) Click **Done**.

f. Optional: View or change parts that are being signed or encrypted in the response.

1) Under Message level protection, click **Response message part protection**.

2) To view or change the response encrypted part, click *app_encparts*, and then click **Edit**.

The **Elements in Part** page displays with the parts that will be encrypted in the response message. You can update the settings on this page to add, change, or remove elements to encrypt. By default, the **Body** and an **XPath expression to the Signature** are configured.

When you finish making your changes, click **OK**.

3) To view or change the response signed part, click *app_signparts*, and then click **Edit**.

The **Elements in Part** page displays with the parts that will be signed in the response message. You can update the settings on this page to add, change, or remove elements to sign. By default, the **Body**, the **QNames for the WS-Addressing header**, and **XPath expressions to the Timestamp** are configured.

When you finish making your changes, click **OK**.

4) Click **Done**.

g. Click **Apply**.

h. Save the configuration.

3. Configure the client to use the AsignEncPolicy policy set.

a. In the administrative console, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings**.

b. Select the web services client resource (JaxWSServicesSamples).

c. Click **Attach Policy Set**.

d. Select **AsignEncPolicy**.

4. Create a custom binding for the client.

a. Select the web services resource again.

b. Click **Assign Binding**.

c. Click **New Application Specific Binding** to create an application-specific binding.

d. Specify the bindings configuration name.

`name:` *`signEncClientBinding`*

e. Click **Add > WS-Security**.

f. If the **Main Message Security Policy Bindings** panel does not display, select **WS-Security**.

5. Configure the client's custom bindings.

a. Configure a Certificate Store.

1) Click **Keys and Certificates**.

2) Under Certificate store, click **New Inbound...** .

3) Specify `name=`*`clientCertStore`*.

4) Specify `Intermediate X.509 certificate=${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`

5) Click **OK**.

b. Configure a Trust Anchor.

1) Under Trust anchor, click **New...**

2) Specify name=*clientTrustAnchor*
3) Click **External Keystore** .
4) Specify Full path=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks.
5) Specify Password=*client*.
6) Click **OK**.
7) Click **WS-Security** in the navigation for this page.

c. Optional: If Signing the request message, complete the following actions.
   1) Configure the Signature Generator.
      a) Click **Authentication and protection** > **AsymmetricBindingInitiatorSignatureToken0** (signature generator), and then click **Apply**.
      b) Click **Callback handler**
      c) Specify Keystore=custom.
      d) Click **Custom keystore configuration**, and then specify
         Full path==${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks
         Keystore password=*client*
         Name=*client*
         Alias=*soaprequester*
         Password=*client*
      e) Click **OK**, **OK**, and **OK**.
   2) Configure the request Signing Information.
      a) Click **request:app_signparts**, and specify Name=*clientReqSignInfo*.
      b) Under Signing Key Information, click **New** , and then specify:
         Name=*clientReqSignKeyInfo*
         Type=Security Token reference
         Token generator or consumer name=AsymmetricBindingInitiatorSignatureToken0
      c) Click **OK**, and then click **Apply**.
      d) Under Message part reference, select **request:app_signparts** .
      e) Click **Edit**.
      f) Under Transform algorithms, click **New**
      g) Specify URL=http://www.w3.org/2001/10/xml-exc-c14n#.
      h) Click **OK**, **OK**, and **OK**.

d. Optional: If Signing the response message, complete the following actions.
   1) Configure the Signature Consumer.
      a) Click **AsymmetricBindingRecipientSignatureToken0** (signature consumer), and then click **Apply**.
      b) Click **Callback handler**
      c) Under Certificates, click the Certificate store radial button, and specify:
         - Certificate store=clientCertStore
         - Trusted anchor store=clientTrustAnchor
      d) Click **OK** and **OK**.
   2) Configure the response Signing Information.
      a) Click **response:app_signparts**, and specify Name=*clientRspSignInfo*.
      b) Click **Apply**.
      c) Under Signing Key Information, click **New**, and then specify:
         Name=*clientReqSignKeyInfo*
         Token generator or consumer name=AsymmetricBindingInitiatorSignatureToken0
      d) Click **OK**.

e) Under Signing Key Information, click **clientRspSignKeyinfo**, and then click **Add**.
f) Under Message part reference, select **response:app_signparts** .
g) Click **Edit**.
h) Under Transform algorithms, click **New**
i) Specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
j) Click **OK**, **OK**, and **OK**.

e. Optional: If Encrypting the request message, complete the following actions.
1) Configure the Encryption Generator.
a) Click **AsymmetricBindingRecipientEncryptionToken0** (encryption generator), and then click **Apply**.
b) Click **Callback handler**, and specify `Keystore=custom`.
c) Click **Custom keystore configuration**, and then specify
```
Full path==${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks
Type=JCEKS
Keystore password=storepass
Key Name=bob
Key Alias=bob
```
d) Click **OK**, **OK**, and **OK**.
2) Configure the request Encryption Information.

gotcha: The setting for **Usage of key information references** must be set to `Key encryption`, which is the default value. Data encryption is used for Symmetric encryption.

a) Click **request:app_encparts**, and specify `Name=clientReqEncInfo`.
b) Click **Apply**.
c) Under Key Information, click **New**, and then specify
```
Name=clientReqEncKeyInfo
Type=Key_identifier
Token generator or consumer name=AsymmetricBindingRecipientEncryptionToken0
```
d) Click **OK**.
e) Under Key Information, select **clientReqEncKeyInfo**, and then click **OK**.

f. Optional: If Encrypting the response message, complete the following actions.
1) Configure the Encryption Consumer.
a) Click **AsymmetricBindingInitiatorEncryptionToken0** (encryption consumer), and then click **Apply**.
b) Click **Callback handler**, and specify `Keystore=custom`.
c) Click **Custom keystore configuration**, and then specify
```
Full path==${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks
Type=JCEKS
Keystore password=storepass
Key Name=alice
Key Alias=alice
Key password=keypass
```
d) Click **OK** and **OK**.
2) Configure the response Encryption Information.

gotcha: The setting for **Usage of key Information references** must be set to `Key encryption`, which is the default value. Data encryption is used for Symmetric encryption.

a) Click **response:app_encparts**, and specify `Name=clientRspEncInfo`.

b) Click **Apply**.

c) Under Key Information, click **New**, and then specify

```
Name=clientRspEncKeyInfo
Token generator or consumer name=AsymmetricBindingRecipientEncryptionToken0
```

d) Click **OK**.

e) Under Key Information, select **clientRspEncKeyInfo**.

f) Click **Add**, and then click **OK**.

6. Configure the provider to use the AsignEncPolicy policy set.
   a. In the administrative console, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service provider policy sets and bindings**.
   b. Select the web services provider resource (JaxWSServicesSamples).
   c. Click **Attach Policy Set**.
   d. Select **AsignEncPolicy**.
7. Create a custom binding for the provider.
   a. Select the web services provider resource again.
   b. Click **Assign Binding**.
   c. Click **New Application Specific Binding** to create an application-specific binding.
   d. Specify `Bindings configuration name: signEncProviderBinding`.
   e. Click **Add** > **WS-Security**.
   f. If the **Main Message Security Policy Bindings** panel does not display, select **WS-Security**.
8. Configure the custom bindings for the provider.
   a. Configure a Certificate Store.
      1) Click **Keys and Certificates**.
      2) Under Certificate store, click **New Inbound...**.
      3) Specify:
      ```
      Name=providerCertStore
      Intermediate X.509 certificate=${USER_INSTALL_ROOT}/etc/ws-security/samples/
      intca2.cer
      ```
      4) Click **OK**.
   b. Configure a Trust Anchor.
      1) Under Trust anchor, click **New...**
      2) Specify, `Name=`*`providerTrustAnchor`*.
      3) Click **External Keystore**, and specify:
      ```
      Full path=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
      Password=server
      ```
      4) Click **OK**, and then click **WS-Security** in the navigation for this page, and then click **Authentication and protection**.
   c. Optional: If Signing the request message, complete the following actions.
      1) Configure the Signature consumer.
         a) Click **AsymmetricBindingInitiatorSignatureToken0** (signature consumer), and then click **Apply**.
         b) Click **Callback handler**.
         c) Under Certificates, click the **Certificate store** radial button, and specify:
         ```
         Certificate store=providerCertStore
         Trusted anchor store=providerTrustAnchor
         ```
         d) Click **OK**.
         e) Click **Authentication and protection** in the navigation for this page.

2) Configure the request Signing Information.
   a) Click **request:app_signparts**, and specify `Name=reqSignInfo`.
   b) Click **Apply**.
   c) Under Signing Key Information, click **New**, and specify:
```
Name=reqSignKeyInfo
Token generator or consumer
name=AsymmetricBindingInitiatorSignatureToken0
```
   d) Click **OK**.
   e) Under Signing Key Information, click **reqSignKeyinfo**, and then click **Add**.
   f) Under Message part reference, select **request:app_signparts**.
   g) Click **Edit**.
   h) Under Transform algorithms, click **New**, and then specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
   i) Click **OK**, **OK**, and **OK**.

d. Optional: If Signing the response message, complete the following actions.
   1) Configure the Signature Generator.
      a) Click **AsymmetricBindingRecipientSignatureToken0** (signature generator), and then click **Apply**.
      b) Click **Callback handler > Custom keystore configuration**, and specify:
```
Full path=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
Keystore password=server
Name=server
Alias=soapprovider
Password=server
```
      c) Click **OK**, **OK**, and **OK**.
   2) Configure the response Signing Information.
      a) Click **response:app_signparts**, and specify `Name=rspSignInfo`.
      b) Under Signing Key Information, click **New**, and specify:
```
Name=rspSignKeyInfo
Type=Security Token reference
Token generator or consumer
name=AsymmetricBindingRecipientSignatureToken0
```
      c) Click **OK**, and then click **Apply**.
      d) Under Message part reference, select **response:app_signparts**.
      e) Click **Edit**.
      f) Under Transform algorithms, click **New**, and then specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
      g) Click **OK**, **OK**, and **OK**.

e. Optional: If Encrypting the request message, complete the following actions.
   1) Configure the Encryption Consumer.
      a) Click **AsymmetricBindingRecipientEncryptionToken0** (encryption consumer), and then click **Apply**.
      b) Click **Callback handler**, and specify `Keystore=custom`
      c) Click **Custom keystore configuration**, and specify:
```
Full path==${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks
Type=JCEKS
Keystore password=storepass
Key Name=bob
Key Alias=bob
```

`Key password=`*keypass*

d) Click **OK**, **OK**, and **OK**.

2) Configure the request Encryption Information.

gotcha: The setting for **Usage of key information references** must be set to `Key encryption`, which is the default value. Data encryption is used for Symmetric encryption.

a) Click **request:app_encparts**, and specify `Name=reqEncInfo`.

b) Click **APPLY**

c) Under Key Information, click **New**, and specify:

```
Name=reqEncKeyInfo
Type=Key identifier
Token generator or consumer
name=AsymmetricBindingRecipientEncryptionToken0
```

d) Click **OK**.

e) Under Key Information, select **reqEncKeyInfo**.

f) Click **Add**, and then click **OK**.

f. Optional: If Encrypting the response message, complete the following actions.

1) Configure the Encryption Generator.

a) Click **AsymmetricBindingInitiatorEncryptionToken0** (encryption generator), and then click **Apply**.

b) Click **Callback handler**, and specify `Keystore=custom`

c) Click **Custom keystore configuration**, and specify:

```
Full path==${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks
Type=JCEKS
Keystore password=storepass
Key Name=alice
Key Alias=alicee
```

d) Click **OK**, **OK**, and **OK**.

2) Configure the request Encryption Information.

gotcha: The setting for **Usage of key information references** must be set to `Key encryption`, which is the default value. Data encryption is used for Symmetric encryption.

a) Click **response:app_encparts**, and specify `Name=rspEncInfo`.

b) Click **APPLY**

c) Under Key Information, click **New**, and specify:

```
 Name=rspEncKeyInfo
Token generator or consumer
name=AsymmetricBindingInitiatorEncryptionToken0
```

d) Click **OK**.

e) Under Key Information, select **rspEncKeyInfo**.

f) Click **OK**.

9. Click **Save** to save your configuration changes.

10. Restart the client and provider.

a. Stop the client and the provider.

b. Restart the client and the provider.

11. Test the Service.

a. Point your web browser at the JaxWSServicesSamples:

```
http://localhost:9080/wssamplesei/demo
```

gotcha: Make sure you provide the correct hostname and port if your profile is not on the same machine, or the port is not 9080.

b. Select **Message Type Synchronous Echo**.
c. Make sure **Use SOAP 1.2** is not selected.
d. Enter a message and click **Send Message**.

The sample application should reply with `JAXWS==>Message`.

Results

The JaxWSServicesSamples web services application is configured to use asymmetrical XML Digital Signature and Encryption to protect your SOAP requests and responses.

Configuring a policy set and bindings for XML Digital Signature with client and provider application specific bindings:

You can create a custom policy set and application specific bindings for using XML Digital Signature to sign the body of the request and response SOAP messages.

Before you begin

This task assumes that the service provider and client that you are configuring are in the JaxWSServicesSamples application. Refer to the topic *Accessing Samples* for more information on how to obtain and install this application.

Use the following trace specification on your server. These specifications enable you to debug any future configuration problems that might occur.

```
*=info:com.ibm.wsspi.wssecurity.*=all:com.ibm.ws.webservices.wssecurity.*=all:
com.ibm.ws.wssecurity.*=all: com.ibm.xml.soapsec.*=all: com.ibm.ws.webservices.trace.*=all:
com.ibm.ws.websvcs.trace.*=all:com.ibm.ws.wssecurity.platform.audit.*=off:
```

About this task

This procedure explains the actions you need to complete to configure WS-Security policy set to use only the XML-Digital Signature WS-Security constraint. This procedure also explains the actions you need to complete to configure XML Digital Signature application specific custom bindings for a client and provider.

The keystores that are used in this procedure are provided with WebSphere Application Server and are installed in every profile that is created. You can use the ${USER_INSTALL_ROOT} variable directly in the configuration to conveniently point to the keystore locations without using a fully-qualified path. ${USER_INSTALL_ROOT} resolves to a path such as c:/WebSphere/AppServer/profiles/AppSrv01.

```
${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks
${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
```

Because of the nature of JaxWSServicesSamples, to apply the policy set and bindings to this application, in the administrative console click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples**. When using your own applications, you can use the following paths as an alternative way to access the provider and client for attachment of the policy set and bindings:

```
* Services > Service Providers > (AppName)
* Services > Service clients > (AppName)
```

gotcha: Pay close attention to the names of the token consumers and generators in the administrative console. The Initiator and recipient might not be what you think they should be for the tokens. The usage column in the table specifies whether a token is a consumer token or a generator token.

Procedure

1. Create the custom policy set.
   a. In the administrative console, click **Services > Policy sets > Application Policy sets**.
   b. Click **New**.
   c. Specify `Name=`*`AsignPolicy`*.
   d. Click **Apply**.
   e. Under Policies, click **Add > WS-Security**.
2. Edit the custom policy set to remove encryption and timestamp.
   a. In the administrative console, click **WS-Security > Main Policy**.
   b. Under Message level protection, click **Request message part protection**.
   c. Click **app_encparts**.
   d. Click **Delete**.
   e. Click **Done**.
   f. Click **Response message part protection**.
   g. Click **app_encparts**.
   h. Click **Delete**.
   i. Click **Done**.
   j. Unselect **Include timestamp in security header**.
   k. Click **Apply**.
   l. Save the configuration.
3. Configure the client to use the AsignPolicy policy set.
   a. In the administrative console, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings**.
   b. Select the web services client resource (JaxWSServicesSamples).
   c. Click **Attach Policy Set**.
   d. Select **AsignPolicy**.
4. Create a custom binding for the client.
   a. Select the web services resource again.
   b. Click **Assign Binding**.
   c. Click **New Application Specific Binding** to create an application-specific binding.
   d. Specify the bindings configuration name.
      `name:` *`clientBinding`*
   e. Click **Add > WS-Security**.
   f. If the **Main Message Security Policy Bindings**' panel does not display, select **WS-Security**.
5. Configure the client's custom bindings.
   a. Configure a Certificate Store.
      1) Click **Keys and Certificates**.
      2) Under Certificate store, click **New Inbound...** .
      3) Specify `name=`*`clientCertStore`*.
      4) Specify `Intermediate X.509 certificate=${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`.
      5) Click **OK**.
   b. Configure a Trust Anchor.
      1) Under Trust anchor, click **New...**
      2) Specify `name=`*`clientTrustAnchor`*.

3) Click **External Keystore** .
4) Specify `Full path=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks`.
5) Specify `Password=`*`client`*.
6) Click **OK**.
7) Click **WS-Security** in the navigation for this page.

c. Configure the Signature Generator.
1) Click **Authentication and protection** > **AsymmetricBindingInitiatorSignatureToken0** (signature generator), and then click **Apply**.
2) Click **Callback handler**
3) Specify `Keystore=custom`.
4) Click **Custom keystore configuration**, and then specify
```
Full path==${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks
Keystore password=client
Name=client
Alias=soaprequester
Password=client
```
5) Click **OK**, **OK**, and **OK**.

d. Configure the Signature Consumer.
1) Click **AsymmetricBindingRecipientSignatureToken0** (signature consumer), and then click **Apply**.
2) Click **Callback handler**.
3) Under Certificates, click the **Certificate store** radial button, and specify:
```
Certificate store=clientCertStore
Trusted anchor store=clientTrustAnchor
```
4) Click **OK**, and **OK**.

e. Configure the request Signing Information.
1) Click **request:app_signparts**, and specify `Name=`*`clientReqSignInfo`*.
2) Under Signing key information, click **New** , and then specify:
```
Name=clientReqSignKeyInfo
Type=Security Token reference
Token generator or consumer name=AsymmetricBindingInitiatorSignatureToken0
```
3) Click **Ok**, and then click **Apply**.
4) Under Message part reference, select **request:app_signparts** .
5) Click **Edit**.
6) Under Transform algorithms, click **New**
7) Specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
8) Click **OK**, **OK**, and **OK**.

f. Configure the response Signing Information.
1) Click **response:app_signparts**, and specify `Name=`*`clientRespSignInfo`*.
2) Click **Apply**.
3) Under Signing key information, click **New** , and then specify:
```
Name=clientRspSignKeyInfo
Token generator or consumer name=AsymmetricBindingRecipientSignatureToken0
```
4) Click **Ok**.
5) Under Signing key information, click **clientRspSignKeyinfo** , and then click **Add**.
6) Under Message part reference, select **response:app_signparts** .
7) Click **Edit**.
8) Under Transform algorithms, click **New**

9) Specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.

10) Click **OK**, **OK**, and **OK**.

6. Configure the provider to use the AsignPolicy policy set.
   a. In the administrative console, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service provider policy sets and bindings**.
   b. Select the web services provider resource (JaxWSServicesSamples).
   c. Click **Attach Policy Set**.
   d. Select **AsignPolicy**.
7. Create a custom binding for the provider.
   a. Select the web services provider resource again.
   b. Click **Assign Binding**.
   c. Click **New Application Specific Binding** to create an application-specific binding.
   d. Specify `Bindings configuration name:`*`providerBinding`*.
   e. Click **Add > WS-Security**.
   f. If the **Main Message Security Policy Bindings**' panel does not display, select **WS-Security**.
8. Configure the custom bindings for the provider.
   a. Configure a Certificate Store.
      1) Click **Keys and Certificates**.
      2) Under Certificate store, click **New Inbound...**.
      3) Specify:

```
Name=providerCertStore
Intermediate X.509 certificate=${USER_INSTALL_ROOT}/etc/ws-security/samples/
intca2.cer
```

      4) Click **OK**.
   b. Configure a Trust Anchor.
      1) Under Trust anchor, click **New...**
      2) Specify, `Name=`*`providerTrustAnchor`*.
      3) Click **External Keystore**, and specify:

```
Full path=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
Password=server
```

      4) Click **OK**, and then click **WS-Security** in the navigation for this page.
   c. Configure the Signature Generator.
      1) Click **Authentication and protection > AsymmetricBindingRecipientSignatureToken0** (signature generator), and then click**Apply**.
      2) Click **Callback handler**
      3) Specify `Keystore=custom`.
      4) Click **Custom keystore configuration**, and then specify

```
Full path=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
Keystore password=server
Name=server
Alias=soapprovider
Password=server
```

      5) Click **OK**, **OK**, and **OK**.
   d. Configure the Signature Consumer.
      1) Click **AsymmetricBindingInitiatorSignatureToken0** (signature consumer), and then click **Apply**.
      2) Click **Callback handler**.

3) Under Certificates, click the Certificate store radial button, and specify:
   ```
   Certificate store=providerCertStore
   Trusted anchor store=providerTrustAnchor
   ```
4) Click **OK**.
5) Click **Authentication and protection** in the navigation for this page.

e. Configure the request Signing Information.
1) Click **request:app_signparts**, and specify `Name=reqSignInf`.
2) Click **Apply**.
3) Under Signing key information, click **New** , and then specify:
   ```
   Name=reqSignKeyInfo
   Token generator or consumer name=AsymmetricBindingInitiatorSignatureToken0
   ```
4) Click **Ok**.
5) Under Signing key information, click **reqSignKeyinfo**, and then click **Add**.
6) Under Message part reference, click **request:app_signparts**.
7) Click **Edit**.
8) Under Transform algorithms, click **New**, and then specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
9) Click **OK**, **OK**, and **OK**.

f. Configure the response Signing Information.
1) Click **response:app_signparts**, and specify `Name=rspSignInfo`.
2) Click **Apply**.
3) Under Signing key information, click **New** , and then specify:
   ```
   Name=rspSignKeyInfo
   Type=Security Token reference
   Token generator or consumer name=AsymmetricBindingRecipientSignatureToken0
   ```
4) Click **Ok**, and then click **Apply**.
5) Under Message part reference, select **response:app_signparts** .
6) Click **Edit**.
7) Under Transform algorithms, click **New**.
8) Specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
9) Click **OK**, **OK**, and **OK**.

9. Click **Save** to save your configuration changes.
10. Restart the client and provider.
    a. Stop the client and the provider.
    b. Restart the client and the provider.
11. Test the Service.
    a. Point your web browser at the JaxWSServicesSamples:
       `http://localhost:9080/wssamplesei/demo`

       gotcha: Make sure you provide the correct hostname and port if your profile is not on the same machine, or the port is not 9080.
    b. Select **Message Type Synchronous Echo**.
    c. Make sure **Use SOAP 1.2** is not selected.
    d. Enter a message and click **Send Message**.

    The sample application should reply with `JAXWS==>Message`.

Results

The JaxWSServicesSamples web services application is configured to use XML Digital Signature to sign the body for both the SOAP request and response.

Configuring the username and password for WS-Security Username or LTPA token authentication:

When using the Username WSSecurity default policy set, you must configure the username and password for username token authentication separately from the security settings defined in the bindings.

About this task

When you install a JAX-WS application and attach the default Username WSSecurity default policy set, the next step is to configure the general provider sample binding for the JAX-WS provider, and the general client sample binding for the JAX-WS client. However, the binding file for the default client sample binding does not include a username or password for token authentication. Since the username and password is not available from the target deployed system, you must specify a valid username and password in your environment using the administrative console.

Procedure

1. Log in to the administrative console, then click **Services** > **Policy sets** > **General client policy set bindings**.
2. Click **Client sample** to edit the binding.
3. Click **WS-Security**.

   Add basic authentication information, such as username and password, to the general client sample bindings for any policy set that uses a Username token or LTPA token, including:
   - Username SecureConversation
   - Username WS-I RSP
   - LTPA SecureConversation
   - LTPA WS-I RSP
   - LTPA WSSecurity default
4. Click **Authentication and protection**.
5. In the Authentication tokens table, click **gen_signunametoken** to edit the username token settings.
6. Click **Callback handler** in the Additional Bindings section.
7. Enter the appropriate username and password information for your environment in the **User name** and **Password** fields.
8. Enter the password a second time in the **Confirm Password** field, then click **Apply**.
9. Repeat steps 5 through 8 for the gen_signltpatoken LTPA token generator.

Results

Note: This administrative console panel applies only to Java™ API for XML Web Services (JAX-WS) web services.

Securing requests to the trust service using system policy sets:

WebSphere Application Server provides message-level protection for its security token service, known as the WebSphere Application Server trust service. For the trust service, you must use a special class of policy sets known as system policy sets.

Before you begin

You can secure requests to the trust service by using two different configuration methods:

- Use the administrative console to define and attach a system policy set and binding to a trust service operation that is associated with an endpoint.
- Use the wsadmin tool, which supports the Jython and Jacl scripting languages, to configure system policy sets for the trust service. You can manage the policies for the Quality of Service (QoS) by creating policy sets and managing associated policies.

About this task

For WebSphere Application Server trust service security, you must configure the system policy sets, the bindings, the trust service attachments, and the security cache.

Perform the following high-level steps. The order of the tasks is not important but all high-level required steps must be performed to complete the trust configuration.

Procedure

1. Define a new system policy set or manage existing system policy sets. To manage system policy sets, you can perform the following tasks:
   a. Define the system policy set and binding. The system policy set can be a new or existing policy set. If you create a new system policy set, you must specify and configure the policy types. A default binding configuration is associated with each policy type.
   b. Modify the system policy set, as needed.

      Other optional policy set-related tasks that you can perform include:
      - Add, edit, or remove policy set attachments.
      - Edit, enable, disable or remove policy types
      - Create a system policy set by selecting and copying an existing system policy set. When copying an existing system policy set, you also specify whether to move the existing attachments to this new system policy set.
      - Delete system policy sets. You cannot delete pre-configured system policy sets that are provided by WebSphere Application Server by default.
      - Archive a system policy set by selecting and exporting an existing system policy set. When exporting an existing system policy set, you create a .zip archive file. The .zip file for exporting the policy set is provided for downloading. For example, if you have a policy set named ABC_ps and you want to export and move the archive file from ServerA to ServerB, first use the export function to create the .zip file. Then, manually transfer the archive file to ServerB.
2. Create and manage explicit attachments. You can perform the following trust service attachment tasks:
   a. Attach the system policy set and assign a binding to an endpoint. For an endpoint, you can create explicit attachments for each of the four trust service operations to the respective Trust Service Defaults policy sets and bindings. After you have created these initial attachments, you can view and further modify existing policy set and binding configurations.
   b. Modify existing policy set attachment and binding configurations, as needed.. The system policy set can be a new or existing policy set. If you create a new system policy set, you must specify and configure the policy types. A default binding configuration is associated with each policy type.

      The system policy set that is attached to issue and renew must correspond to the client and endpoint's bootstrap policy set and the system policy set attached to validate and cancel must correspond to the client and endpoint's application policy set. The bootstrap policy set for the endpoint service is only required if the endpoint service makes issue and renew requests to the trust service.

      Other optional attachment-related tasks that you can perform include:
      - Change the system policy set and binding configurations.

- Create custom system policy sets and bindings.
- Attach each of the four default trust service operations to a system policy set and binding.
- Attach each of the four trust service operations associated with a specific endpoint to a system policy set and binding.
- Specify that the selected trust service operations for an endpoint inherit the respective default trust service policy set and binding.
- Assign the Default binding or a custom binding configuration to the selected policy set attachment.
- Update the trust service runtime configuration.

3. Manage the security context token provider that the trust service provides. You can perform the following trust service token provider tasks:
   a. Modify the configuration of the Security Context Token provider, as needed..

      Other optional token provider-related tasks that you can perform include:
      - Update the trust service runtime configuration for any token provider configuration changes.
4. Manage the trust service default token provider and any endpoints that have an explicitly assigned token (rather than inheriting from the default). Targets are endpoints that are assigned a specific token provider. You can perform the following trust service target tasks:
   a. Create a new trust service target by explicitly assigning a service endpoint URL to the default token provider.. Performing this task creates an explicit assignment to the default trust service token provider, the Security Context Token. All other endpoints inherit the trust service default token provider.
   b. Configure a target. WebSphere Application Server defines one default supported token provider, the Security Context Token. Other tasks that you can perform for existing targets include:
      - Modifying one or more endpoints that have a security context token provider explicitly assigned.
      - Changing the token provider for an endpoint from inherited to explicitly assigned. Therefore, the token provider for the endpoint does not change as the default trust service token provider changes.
      - Changing the token provider for an endpoint from explicitly assigned to inherited. Therefore, the token provider for the endpoint is the default trust service token provider and changes as the default changes.
      - Updating the trust service runtime configuration.
5. Configure the security cache. You can change the behavior of client-side security caching.
6. Update the trust service runtime configuration. You must update the runtime configuration whenever one or all of the following trust-related items are created or changed:
   - Trust service attachments
   - Token providers
   - Targets

Results

After the configurations are completed and the trust service runtime configuration has been updated, you have used the administrative console to secure requests to the trust service by using system policy sets.

Enabling secure conversation:

Use secure conversation to secure web services application messages.

Before you begin

Applications that contain web services must have been deployed.

About this task

The Organization for the Advancement of Structured Information Standards (OASIS) Web Services Secure Conversation (WS-SecureConversation) draft specification describes ways to establish a secure session between the initiator and recipient of SOAP messages. The WS-SecureConversation draft specification also defines how to use the OASIS Web Services Trust (WS-Trust) protocol to establish a security context token (SCT). For complete information, see the OASIS Web Services Secure Conversation specification.

WebSphere Application Server supports the ability of an endpoint to issue a security context token for WS-SecureConversation, and thereby provides a secure session between the initiator and recipient of SOAP messages.

The following figure describes the flow that is required to establish a secured context and to use session-based security.

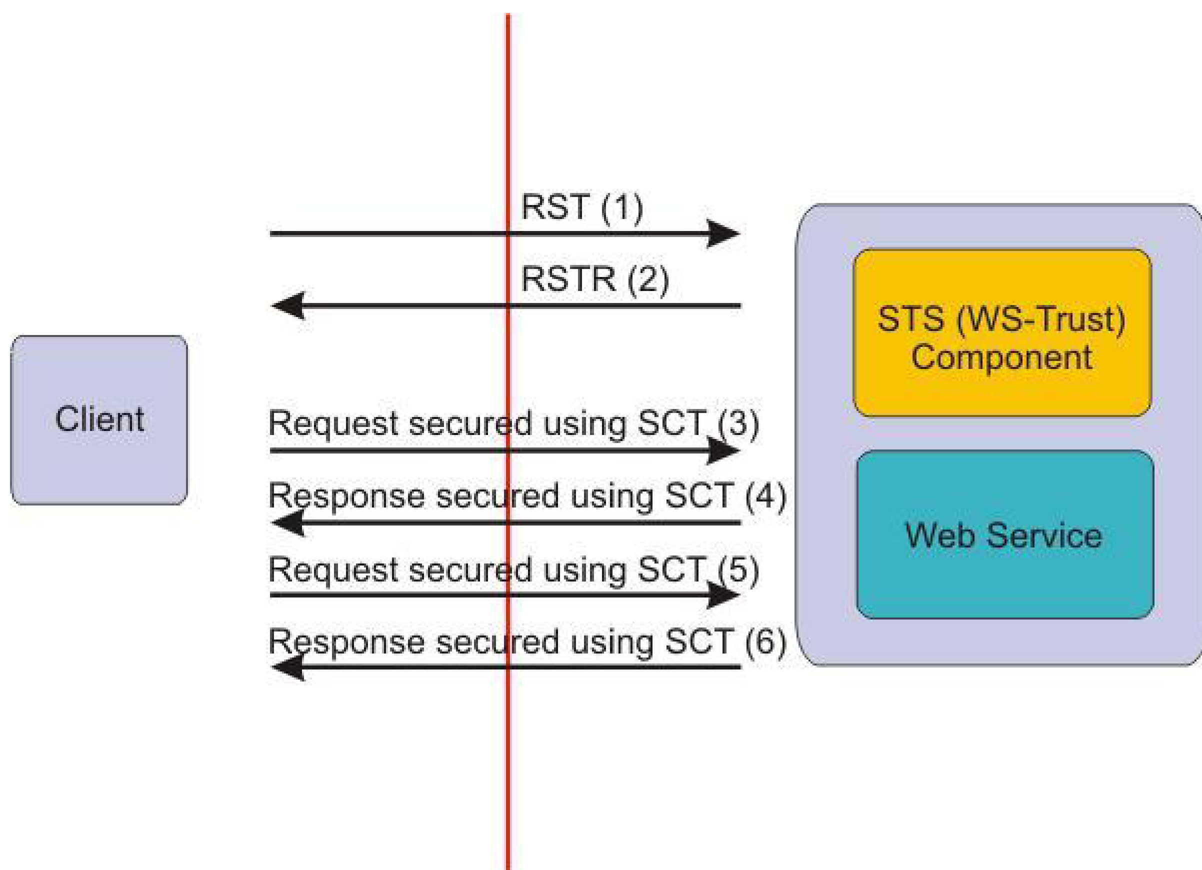


Figure 23. Displaying the flow between the client and the web service and security token service

In the WS-SecureConversation specification, a security context is represented by the <wsc:SecurityContextToken> security token. The following example represents the assertion syntax for a <wsc:SecurityContextToken> element.

```
<wsc:SecurityContextToken wsu:Id="..." ...>
    <wsc:Identifier>...</wsc:Identifier>
    <wsc:Instance>...</wsc:Instance>
    ...
</wsc:SecurityContextToken>
```

The security context token does not support references to it by using key identifiers or key names. All references must either use an ID (to a wsu:Id attribute) or a <wsse:Reference> to the <wsc:Identifier> element.

WebSphere Application Server provides these pre-configured secure conversation-related polices:

- The **SecureConveration** policy set follows the WS-SecureConversation and WS-Security specifications and provides a policy set with secure conversation enabled and using keys derived from security context token for signing and encrypting the application messages.
- The **Username SecureConversation** policy set follows the WS-SecureConversation and WS-Security specifications and adds authentication using the Username token.
- The **LTPA SecureConversation** policy follows the WS-SecureConversation and WS-Security specifications and provides authentication using the Lightweight Third Party Authentication (LTPA) tokens.

In this example, the default SecureConversation policy set, and the default WS-Security binding and TrustServiceSecurityDefault binding are used to achieve the task of enabling secure conversation. The default SecureConversation policy set has both the application policy (symmetricBinding) and the bootstrap policy (asymmetricBinding). The application policy is used to secure application messages and the bootstrap policy is used to secure the RequestSecurityToken (RST) messages.

A trust service that issues a security context token is configured with the TrustServiceSecurityDefault system policy and the TrustServiceSecurityDefault binding. The trust policy is responsible for securing RequestSecurityTokenResponse (RSTR) messages. If the bootstrap policy is modified, the trust policy has to be modified to match both of the configurations.

Note: The following steps are to be used only in development and test environments.

The Web Services Security (WS-Security) default bindings that are used here contain sample key files and must be customized before use in a production. For the production environment, use of custom bindings is advised. Also note that, if the profile is created by using the choice of **Create the server using the development template**, you can skip steps 2 and 3.

To configure secure conversation, configure the policy set, and add a policy assertion to the policy, complete the following steps:

Procedure

1. Make a copy of a default secure conversation policy so you can customize the policy set for your own environment.
   a. Launch the administrative console, and click **Services** > **Policy sets** > **Application policy sets**.
   b. Select the check box next to an existing policy set that follows the WS-SecureConversation specifications. For example, you might click the check box next to **SecureConversation**. This policy set is one of the pre-configured secure conversation-related application policy sets that is listed in the table. The SecureConversation policy set has a bootstrap policy to match the default policy set for the trust service to issue and renew tokens.
   c. Click **Copy**.
   d. Enter a unique name for the new copy of the SecureConversation application policy set. For example: CopyOfSCPolicySet
   e. Optional: Change the description, as needed, for your customized version of this policy set.
2. Attach the policy set and binding to the application.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application name***.
   b. Click either **Service provider policy sets and bindings** or **Service client policy sets and bindings** to attach resources to the CopyOfSCPolicySet policy set. The general binding is assigned automatically as the default.
   c. You can use the **Attach Policy Set** and **Assign Binding** menu lists to select a different policy set or binding.

Results

After completing these steps, you have configured secure conversation.

What to do next

Next, review the example scenario about how to establish a security context token to secure a secure conversation.

Web Services Secure Conversation:

Web Services Secure Conversation (WS-SecureConversation) provides a secured session for long running message exchanges and leveraging of the symmetric cryptographic algorithm.

WS-SecureConversation provides session-based security. Session-based security optimizes long message exchanges, as symmetric cryptography can be used to sign and encrypt the message. Typically, symmetric cryptographic algorithm is less CPU intensive than the asymmetric cryptography. Symmetric cryptographic algorithms should provide better performance and throughput when compared to the asymmetric cryptographic algorithms.

The symmetric cryptographic algorithm also provides a means to secure other session-based protocol and exchange patterns, such as Web Services Reliable Messaging (WS-ReliableMessaging).

Security context token for secure conversation

The Web Services Security specification defines the basic mechanisms for providing secure messaging. The Web Services Trust (WS-Trust) specification defines extensions to Web Services Security that provide ways to establish and broker trust relationships between two parties. The WS-Trust protocol defines the syntax of the request that can be sent to a security token service and the corresponding or subsequent response of the security token service. The security token service provided with WebSphere Application Server is called the *trust service*.

Using the WS-Trust protocol, a party can request the trust service issue a security context token (SCT). Then, this token can be used to establish a secure conversation (WS-SecureConversation). The request for a security token is sent to an application endpoint. The request is intercepted by the WebSphere Application Server and routed to the trust service.

A policy can be defined as the default for all trust issue operations, renew operations, validate operations, or cancel operations. Additionally, a policy can be attached to a specific URL and operation pair.

WS-SecureConversation defines extensions to allow security context establishment and sharing, and session key derivation, which allows contexts to be established and, potentially, more efficient keys, or new key material, to be exchanged. The WebSphere Application Server support for WS-Trust and WS-SecureConversation focuses on the issuing, renewing, validating, and cancelling of the security context token for secure conversation.

Policy set and bootstrap policy

In addition to describing these functions, the OASIS WS-SecureConversation draft submission describes multiple methods of establishing a secure session between the initiator and the recipient of the SOAP messages.

The bootstrap security policy is the security policy for the initiating party to acquire the security token for secure conversation from the trust service by using a token-issuing WS-Trust or WS-SecureConversation protocol message. The policy set configuration consists of the security policy for communication with the application service, and the bootstrap policy for communication with the trust service.

If sharing of a policy configuration (using WS-Policy) containing the secure conversation bootstrap policy fails, it may be because the bootstrap request and response policies differ. The message part protection for the bootstrap policy must be the same for both request and response bootstrap messages, because a single policy is published for both request and response.

What is supported for Web Services Secure Conversation

The following list highlights some of the key functions that are supported in WebSphere Application Server. The list is not exhaustive.

- A security context token (SCT) established between the initiating party and the recipient party.
- The WS-SecureConversation operations that are supported on the security context token (SCT), such as Issue token, Renew token, and Cancel token. Validate token is supported using WS-Trust protocol.
- A derived key (explicit and implied)

What is not supported for Web Services Secure Conversation

The following list highlights some of the key functions that are not supported in WebSphere Application Server. The list is not exhaustive.

- WS-SecureConversation does not support establishing a security context through the security context token that is created by an external security token service (trust component). However, WebSphere Application Server supports an internal security token service.
- WebSphere Application Server does not support establishing a security context through the security context token that is created by one of the communicating parties and propagated with a message.
- WebSphere Application Server does not support amending a security context token.
- WebSphere Application Server does not support a client creating the security context token.
- WebSphere Application Server provides no support for exchange and negotiation.

Secure conversation scenarios

The following scenarios describe the WS-SecureConversation functions that WebSphere Application Server supports:

- WS-SecureConversation

  This scenario is based on establishing a security context token with the recipient and using the derived key to sign and encrypt the message. It describes how to establish a security context by using session-based security. Session-based security is where the flow of the initiator establishes the security context token by using the WS-SecureConversation protocol with the recipient.
- WS-SecureConversation with WS-ReliableMessaging

  This scenario is a composite scenario that includes functions that are required for the composition scenario of Web Services Reliable Messaging (WS-ReliableMessaging), WS-SecureConversation, and WS-Trust. This scenario describes how to use WS-SecureConversation with WS-ReliableMessaging where the flow is similar to the previous scenario, but which is from the secure conversation prospective. However, the main difference is that the WS-ReliableMessaging sequence is secured with the security context token and scopes the WS-ReliableMessaging sequence to the security context token. This description focuses on the message exchanges that are using the security context token in the overall flow.

Scoping of Web Services Secure Conversation:

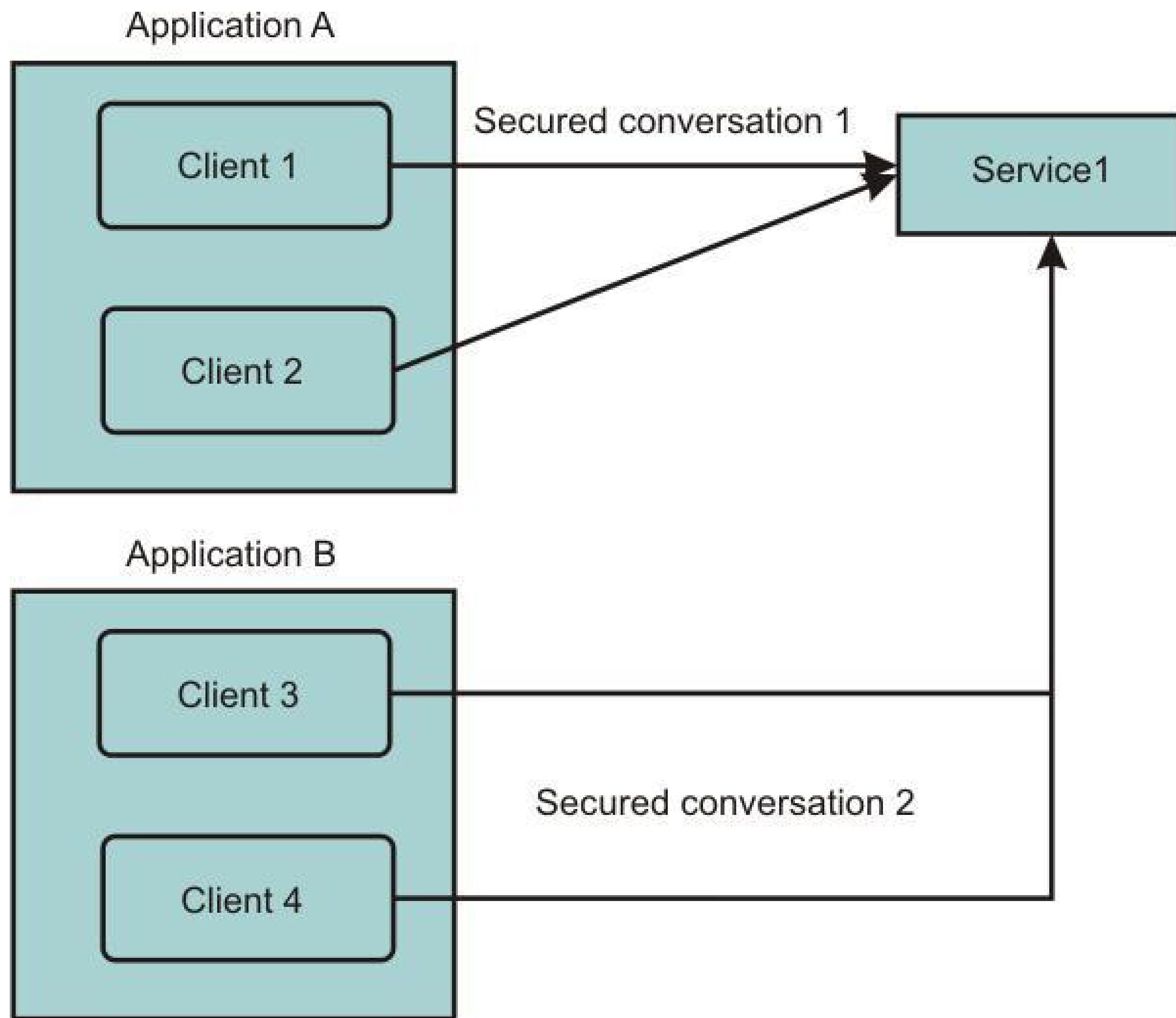
Web Services Secure Conversation supports two scoping mechanisms: the default and the Java API for XML Web Services (JAX-WS) client service level.

Review the following information about the two scoping mechanisms to ensure the proper scoping of secure conversation and policy set for WebSphere Application Server.

Default

The default scope is based on a cluster, an application, a module, and a target service endpoint. For a client running in a thin client environment, it is considered to be a single application, cluster, and module.

In this scoping mode, all the instances of the JAX-WS client within a particular application, cluster, and module to the same target service endpoint share the same secure conversation. For example, in the following figure, the two client instances (Client 1 and Client 2) are in the same module. Client 1 and Client 2 share the same secured conversation with Service 1. The other two client instances (Client 3 and Client 4), which are in a different module than Clients 1 and 2 and which share a secured conversation with each other but not with Clients 1 and 2.



JAX-WS client service level

Scope at the JAX-WS client service level is enabled by specifying a property in the token generator binding configuration of the Secure Conversation Token (SCT)in the client application request (application outbound). The binding is located in the META-INF of the deployed application.

For example, if the application is `WSSampleClientBeta.ear` and the binding directory is `SecureConversation123binding`, the binding file would be located at:

```
$PROFILE_DIR/config/cells/<cellname>/WSSampleClientBeta.ear/deployments/WSSampleClientBeta
/META-INF/SecureConversation123binding/PolicyTypes/WSSecurity/bindings.xml.
```

An example of the configuration follows:

```
<tokenGenerator name="gen_enctgen"
      classname="com.ibm.ws.wssecurity.wssapi.token.impl.CommonTokenGenerator">
   <valueType localName="http://schemas.xmlsoap.org/ws/2005/02/sc/sct" uri="" />
      <callbackHandler classname="com.ibm.ws.wssecurity.impl.auth.callback.WSTrustCallbackHandler">
         <properties name="com.ibm.ws.wssecurity.sc.SCTScope" value="SERVICE_SCOPE"/>
      </callbackHandler>
         <properties name="com.ibm.ws.wssecurity.sc.dkt.ServiceLabel" value="WSC"/>
         <properties name="com.ibm.ws.wssecurity.sc.dkt.ClientLabel" value="WSC"/>
            <jAASConfig configName="system.wss.generate.sct"/>
</tokenGenerator>
```

The following code example demonstrate the behavior after the property in the token generator binding configuration of the SCT in the client application request (application outbound) is enabled. In this mode, Web Services Secure Conversation is scoped at the JAX-WS client service instance.

```
QName serviceQname = new QName("http://ws.apache.org/axis2", "EchoService");
QName portQname = new QName("http://ws.apache.org/axis2", "EchoServicePort");
String endpointUrl = "http://myhost/......";
Service svc1 = Service.create(serviceQname);
svc1.addPort(portQname, null, endpointUrl);
Dispatch<Source> dispatch = svc1.createDispatch(portQname, Source.class, null);
......
......
Service svc2 = Service.create(serviceQname);
svc2.addPort(portQname, null, endpointUrl);
Dispatch<Source> dispatch = svc2.createDispatch(portQname, Source.class, null);
```

where `svc1` and `svc2` are in two different secure conversations with the target service endpoint.

You can change the scope by using either the administrative console or by using scripting to add a property.

Secure conversation client cache and trust service configuration:

For both distributed and local clients, the WebSphere Application Server secure conversation client cache stores tokens on the client.

WebSphere Application Server supports caching of the security context token for both the distributed client and local client. If the security context token is distributed, a client in the same replication domain uses the same security context token. Distributed caching also supports disk offload to save the security context token to disk for recovery. When the client runs applications using secure conversation, and is part of a cluster setup, then the client can use the distributed cache mechanism to replicate the token data among the cluster members.

To use the administrative console to modify the cache settings, click **Services** > **Security Cache**.

You can configure the cache settings, such as the following.

- Set the time that the token remains in the cache after timeout. The default value is 10 minutes. This value is a time window to renew an expired token.
- Set the renewal interval before the token expires. The default value is 10 minutes, and the minimum value is 3 minutes. Entering a number less than 3 minutes causes an error.

  Important: This setting is critical. This setting represents the maximum roundtrip time for a client to make a request, the transport request to go to the server, the server to process the request, and the transport response (if applicable) back to the client. If the time specified is too small and there is not enough time specified, then the token might expire during the roundtrip, and the client receives a failure response. If the time specified is too large, then performance diminishes.

  If the security context token is renewed too often, it might cause Web Services Secure Conversation (WS-SecureConversation) to fail or even cause an out-of-memory error to occur. It is required that you set the renewal interval before the token expires value for the Secure conversation client cache to a

value less than the token timeout value for the security context token. It is also suggested that the token timeout value be at least two times the renewal interval before the token expires value.

- Select the **Enable distributed caching** check box to support distributed clients. You must ensure that the WebSphere Application Server dynamic cache service, and cache replication, are enabled. For more information on enabling the dynamic cache service, refer to the topic Enabling the distributed cache using synchronous update and token recovery.
- Define a custom property, edit, or remove existing custom properties.

The WS-SecureConversation client rejects a security context token that is issued at a future time. If you cannot synchronize the clock between the client machine and service machine, the clock skew could be configured to prevent the rejection of a valid token. The default clock skew is 3 minutes. To modify the default clock skew setting, add the following custom property to the desired minutes:

```
clockSkewToleranceInMinutes
```

Alternatively, use the wsadmin commands to manage secure conversation client cache configurations.

Thin client

For a web service application client running outside WebSphere Application Server, the security context token is cached only in the local Java process. The following system properties can be used to override the default cache setting on the thin client:

com.ibm.wsspi.wssecurity.SC.cache.cushion
: Specifies the time in minutes to renew a security context token to be used with WS-SecureConversation on the client side so that the security context token has enough time to complete the downstream call. The default value is 10 minutes, and the minimum value is 3 minutes.

com.ibm.wsspi.wssecurity.SC.token.clockSkewTolerance
: Specifies the tolerant clock skew time for a token between two machines. The default value is 3 minutes.

WS-Reliable Messaging settings

When WebSphere Application Server applications use policies such as WS-I RSP with managed persistent WS-Reliable Messaging, modify the cache and trust configuration values.

Set the cache configuration time value to 120 minutes.

1. In the WebSphere Application Server administrative console, click **Services** > **Security Cache**.
2. Modify the value of the **Time token is in cache after timeout** field from 10 to 120.
3. Click **Apply**, and then click **Save**.

Increasing the cache time value means that the token remains in the cache for a longer period after token expiration, so that the token is available for renewal. The WS-Reliable Messaging runtime scopes the CreateSequence message to the security context token. Therefore, it is important to maintain the same security context for the life time of the Reliable Messaging sequence.

Enable distributed caching using the default option, Synchronous update of cluster members, to support distributed clients. For more information, refer to the topic Enabling the distributed cache using synchronous update and token recovery.

Additional recommended changes

Other important configuration changes are also recommended.

- Modify the life time of the Security Context Token by changing the value from the default of 120 minutes, to 600 minutes.

- Modify the Renew after expiration value by changing the value from `false` to `true`.
- Modify settings for the token providers, as follows:
  1. In the administrative console, click on **Services** > **Trust service** > **Token providers**.
  2. Click **Security Context Token**.
  3. Change the value in the **Token timeout** field from 120 to 600.
  4. Click the check box to select **Allow renewal after timeout**.
  5. Click **Apply**, and then click **Save**.

Derived key token:

After establishing the security context and after the secret have been established (authenticated), derived keys can be used to sign and encrypt the SOAP message to provide message level protection. You can then use derived keys for each key that is used in the security context.

You can enable Web Services Secure Conversation (WS-SecureConversation) by using symmetric keys that are derived from the security token for signing and encrypting the application messages.

Using WS-SecureConversation, the initiator can establish a security context token using the Web Services Trust (WS-Trust) protocol with the recipient. A security context token implies or contains a shared secret. Using a common secret, different key derivations can be defined. Then, using the security context token, the <wsc:DerivedKeyToken> token can be used to derive keys from any security token that has a shared secret, key, or key material. This secret can be used for signing or encrypting messages, but it is recommended that derived keys be used for signing and encrypting messages that are associated only with the security context.

Syntax for the <wsc:DerivedKeyToken> element

The <wsc:DerivedKeyToken> element is used to indicate that the key for a specific reference is generated from the function so that explicit security tokens, secrets, or key material need not be exchanged as often. The derived key token does not support references to it using key identifiers or key names. All references must use an ID to a *wsu:Id* attribute or use a URI reference, <wsse:Reference>, to the <wsc:Identifier> element in the security context token.

The syntax for <wsc:DerivedKeyToken> element is as follows:

```
<wsc:DerivedKeyToken wsu:Id="...">
    <wsse:SecurityTokenReference>...</wsse:SecurityTokenReference>
    <wsc:Label>...</wsc:Label>
    <wsc:Nonce>...</wsc:Nonce>
  </wsc:DerivedKeyToken>
```

Derived keys are expressed as security tokens and use different algorithms for deriving keys. The following URI is used to represent the derived key token type:

```
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/dk
```

The nonce is processed as a binary octet sequence (the value prior to base64 encoding). The nonce seed is required, and must be generated by one or more of the communicating parties. Use separate nonces and have independently generated keys for signing and encrypting for request and response. New keys should be derived for each message, meaning that a previous nonce should not be reused.

Implied derived key generation

Implied derived keys define a shortcut mechanism for referencing certain types of derived keys. Specifically, an @wsc:Nonce attribute can be added to the security token reference (STR) that is defined in the WS-Security specification. When present, an implied derived key indicates that the key is not in the referenced token but, instead, is a key that is derived from the key or secret of the referenced token. It is recommended that you do not use implied derived Keys in the <wsc:DerivedKeyToken> element.

The following example illustrates a message that is sent using two derived keys, one for signing and one for encrypting:

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..."
    xmlns:xenc="..." xmlns:wsc="..." xmlns:ds="...">
  <S11:Header>
    <wsse:Security>
      <wsc:SecurityContextToken wsu:Id="ctx2">
        <wsc:Identifier>uuid:...UUID2...</wsc:Identifier>
      </wsc:SecurityContextToken>
      <wsc:DerivedKeyToken wsu:Id="dk2">
        <wsse:SecurityTokenReference>
          <wsse:Reference URI="#ctx2"/>
        </wsse:SecurityTokenReference>
        <wsc:Nonce>KJHFRE...</wsc:Nonce>
      </wsc:DerivedKeyToken>
      <xenc:ReferenceList>
        ...
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#dk2"/>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
          ...
      </xenc:ReferenceList>
      <wsc:SecurityContextToken wsu:Id="ctx1">
        <wsc:Identifier>uuid:...UUID1...</wsc:Identifier>
      </wsc:SecurityContextToken>
      <wsc:DerivedKeyToken wsu:Id="dk1">
        <wsse:SecurityTokenReference>
          <wsse:Reference URI="#ctx1"/>
        </wsse:SecurityTokenReference>
        <wsc:Nonce>KJHFRE...</wsc:Nonce>
      </wsc:DerivedKeyToken>
      <xenc:ReferenceList>
        ...
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#dk1"/>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
        ...
      </xenc:ReferenceList>
    </wsse:Security>
  ...
  </S11:Header>
  <S11:Body>
    ...
  </S11:Body>
</S11:Envelope>
```

Enabling secure conversation in a mixed cluster environment:

When Web Services Security applications using secure conversation run in a mixed cluster environment, an interoperability property must be set for the WebSphere Application Server Version 8.5 nodes.

About this task

A mixed cluster environment consists of nodes running WebSphere Application Server Version 6.1 Feature Pack for Web Services, and nodes running WebSphere Application Server V7 and later. To run Web Services Security applications utilizing secure conversation in this environment, enable the following property for the V7.0 and later nodes in the cluster: com.ibm.ws.wssecurity.distributedcache.PreV70InteropMode.

This property ensures that the method for adding entries in the cache is consistent between the different nodes, and also allows the applications to interoperate. To enable the property, follow these steps:

Procedure

1. In the WebSphere Application Server administrative console, click **Servers** > **Server Types** > **WebSphere application servers**.
2. Click the server name.
3. Under the Security section, click **JAX-WS and JAX-RPC security runtime**.
4. Click **Custom properties**.

5. Click **New**.
6. Enter the property name, `com.ibm.ws.wssecurity.distributedcache.PreV70InteropMode`, in the Property name field. Enter the property value of `true` in the Property value field, then click **OK**.
7. When the Properties panel is refreshed, the com.ibm.ws.wssecurity.distributedcache.PreV70InteropMode property appears in the properties table.
8. Click **Save** to commit the change.
9. Restart the server to load the new property.

Results

Secure conversation is now enabled for Web Services Security applications running in a mixed cluster environment.

Enabling distributed cache and session affinity when using Secure Conversation:

WebSphere Application Server provides message-level protection in a cluster environment. You can use Web Services Secure Conversation (WS-SecureConversation) for message-level protection of Java API for XML Web Services 2.0 (JAX-WS) Web services in a cluster environment.

Before you begin

A web services request that is protected with a Security Context Token (SCT) is routed to one server in a cluster, but that SCT might have been issued or renewed by a different server in the cluster. If the WebSphere Application Server distributed cache is not configured to replicate or does not replicate quickly enough, the server processing the request might not have access to the SCT. The task steps described in this topic need to be performed only if the replication setting for cluster members is set to asynchronous update for the Web Services Security distributed cache.

For more information on cache update settings, read the topic Enabling the distributed cache using synchronous update and token recovery. You can also enable the Web Services Security distributed cache with the default setting, which enables synchronous update of cluster members.

About this task

Perform the following high-level steps to enable distributed cache and session affinity when using secure conversation for message-level protection in a cluster environment.

Procedure

1. Enable the distributed cache for the Security Context Token.
   a. In the administrative console for WebSphere Application Server, click **Services** > **Security cache**.
   b. Select the **Enable distributed caching** check box.
   c. Click the radio button to select **Asynchronous update of cluster members**.
   d. Click **Apply** and then click **Save** to save the configuration.
2. Create a replication domain. Perform the following steps:
   a. In the Administrative Console, click **Environment** > **Replication domains** > **New**.
   b. Enter a name. For example, `ABCDomain`.
   c. Under Number of replicas, select the **Entire Domain** option.
   d. Click **OK** and then click **Save** to save the configuration.
3. Enable the dynamic cache. Perform the following steps for each server in the cluster:
   a. In the Administrative Console, click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** > **Container Services** > **Dynamic Cache Service**.
   b. Select the **Enable cache replication** option.

c. Select the replication domain name that you created. For example, `ABCDomain`.
d. Select the replication type as **Both push and pull**.
e. Click **OK** and then click **Save** to save the configuration.

4. Optional: Change the distributed cache batch update interval. By default, the distributed cache batch update interval is 1,000 milliseconds. However, you can set this interval to a value that is less than 1,000 milliseconds. To change the value, complete the following steps for each server in the cluster:
   a. In the Administrative Console, click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** > **Java and Process Management** > **Process Definition** > **Java Virtual Machine** > **Custom Properties** > **New**.
   b. Enter the `com.ibm.ws.cache.CacheConfig.batchUpdateInterval` property name.
   c. Enter the property value.
   d. Click **OK** and then click **Save** to save the configuration.
5. Install and configure a web server or proxy server that supports session affinity. The IBM HTTP Server and WebSphere Application Server proxy server support session affinity. In the WebSphere Application Server Information Center, read the topic “Communicating with Web servers”. for information on installing and configuring the IBM HTTP Server.
6. Configure the client systems to send the web services requests to the host and port where the web server or proxy server is running. The web server or proxy server then routes the requests to the proper cluster member.
7. On the services that are receiving the web services requests, which are protected by using Web Services Secure Conversation, select the HTTP transport Session enabled policy option. Complete the policy set configuration by following these steps:
   a. Add the HTTP Transport policy to the policy set that is being used by the services.
   b. In the configuration panel for the HTTP Transport policy, select **Session enabled**.
   c. Click **OK** and then click **Save** to save the configuration.
8. On the client systems that are sending the web services requests and are protected by Secure Conversation, enable the HTTP transport Maintain session property. Complete the policy set configuration or set the property programmatically. If you are using a policy set with your configuration, follow these steps:
   a. Add the HTTP Transport policy to the policy set that is being used by the clients.
   b. At the HTTP Transport policy configuration panel, select the **Session enabled** option.
   c. Click **OK** and then click **Save** to save the configuration.

Results

After the configurations are completed, you have enabled the distributed cache and session affinity when using secure conversation in a cluster environment. If the server processing the request does not have access to the SCT, it will fail the request with the error of `Either null SCT or invalid SCT`.

Example

The following example, which is a code snippet, demonstrates how to programmatically set the Maintain session property on the correct JAX-WS object:

```
Map<String> rc = ((BindingProvider) port).getRequestContext();
...
rc.put(BindingProvider.SESSION_MAINTAIN_PROPERTY, Boolean.TRUE);
... </String>
```

Flow for establishing a security context token to secure conversations:

This example scenario describes the flow of how the initiator establishes the security context token (SCT) by using the WS-Trust protocol for session-based security with the recipient. After establishing the security context token, derived keys from the security context token are used to sign and encrypt the SOAP

message to provide message-level protection. This examples focuses on the message exchanges using the security context token in the overall flow of the SOAP messages.

The Organization for the Advancement of Structured Information Standards (OASIS) Web Services Secure Conversation (WS-SecureConversation) specification describes ways to establish a secure session between the initiator and recipient of SOAP messages. The WS-SecureConversation specification also defines how to use Web Services Trust (WS-Trust) protocol to establish a security context token. The product supports both Version 1.3, and the draft version, of the WS-SecureConversation specification.

WebSphere Application Server supports the ability of an endpoint to issue a security context token for WS-SecureConversation and thereby provides a secure session between the initiator and recipient of SOAP messages.

The following figure describes the flow that is required to establish a secured context and to use session-based security.

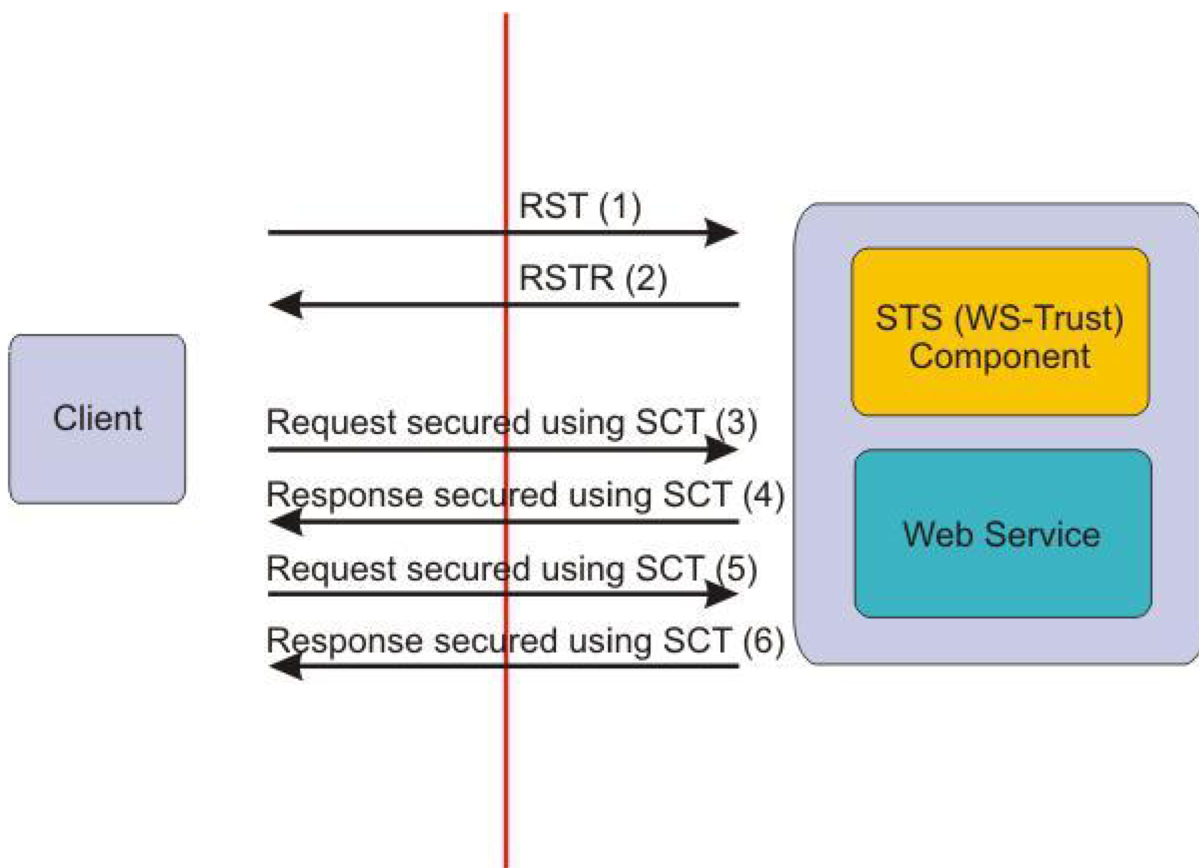


Figure 24. Displaying the flow between the client and the Web service and security token service

Exchanging messages between the initiator and the recipient

The following figure shows how the messages are exchanged between the initiator and the recipient to establish the security context token. The two WS-Trust protocols, RequestSecurityToken (RST) and RequestSecurityTokenResponse (RSTR), are used to request the security context token from the recipient endpoint.

The bootstrap policy is used to secure the RST and validate the RSTR request, which is typically different from the application security policy.

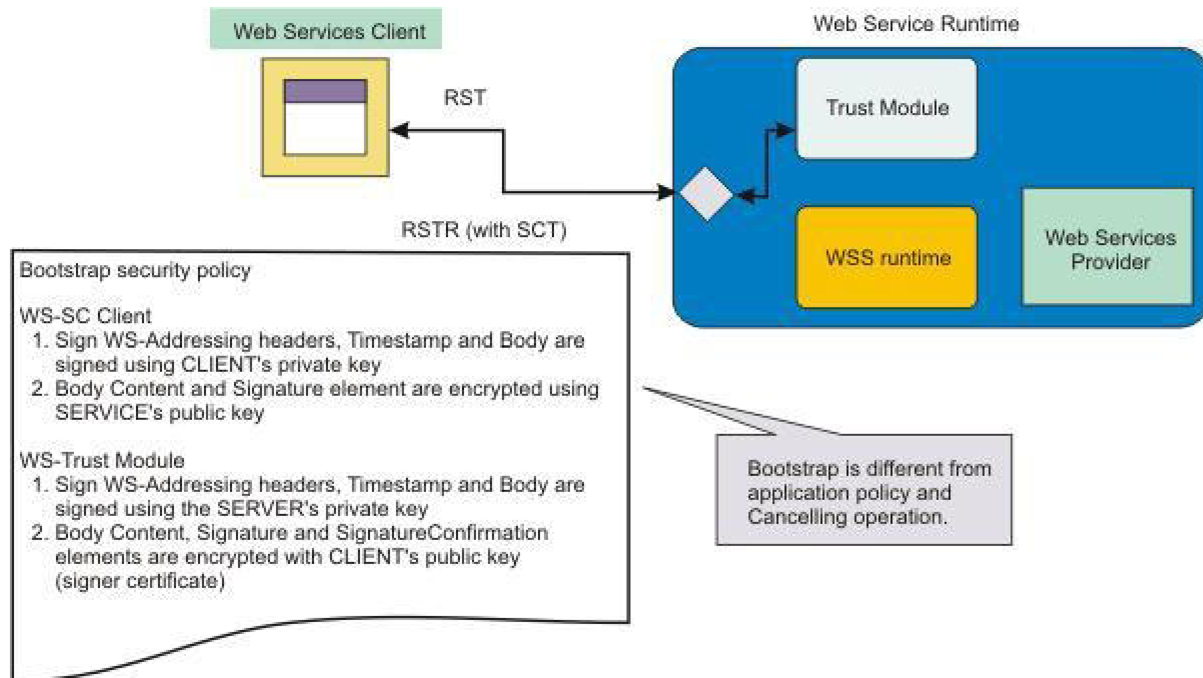


Figure 25. Using WS-Trust protocols RST and RSTR to establish the SCT between the initiator and the recipient

Scenario describing how to use secure conversation

Typically, to use secure conversation, the following steps are involved;

1. The client sends a RequestSecurityToken (RST) trust request for a security context token to an application endpoint with its secret key (entropy and target key size) and requests the target service secret key.

   This request is typically secured with asymmetric Web Service Security that is defined in the bootstrap policy.
2. The RST is processed by the trust service and, if the request is trusted based on the security policy, the trust service returns the security context token with the target service secret key by using a WS-Trust RequestSecurityTokenResponse (RSTR).

   This request is typically secured with asymmetric Web Service Security. The client verifies whether the RSTR can be trusted, based on the bootstrap policy.
3. If the RequestSecurityTokenResponse is trusted, the client secures (signs and encrypts) the subsequent application messages by using the session keys.

   The session keys are derived from secret of the security context token that is obtained from the initial WS-Trust RequestSecurityToken and RequestSecurityTokenResponse messages that are exchanged between the initiator and the recipient.
4. The specification defines an algorithm of how to derive the key based on the initial secret. The target web service calculates the derive key from the metadata contained in the security header of the SOAP message and the initial secret.
5. The target web service uses the derived key to verify and decrypt the message based on the application security policy.
6. The target web service uses the derived key from the secret to sign and encrypt the response based on the application security policy.
7. Repeat of steps 3 through 6 until the message exchange has completed.

Using keys that are derived from the secret of the security context token

After the security context token is established, the application messages are secured with message protection by using keys that are derived from the secret of the security context token. The derived keys are used to secure the application messages by signing and encrypting the application messages. The security context token contains a UUID, which is used as identification of a shared secret. The token UUID can be used in the SOAP message to identify the security context token for the message exchanges. The secret must be kept in memory by the session participants (in this case the initiator and the recipient) and protected. Compromising the secret undermines the secure conversation between the participants.

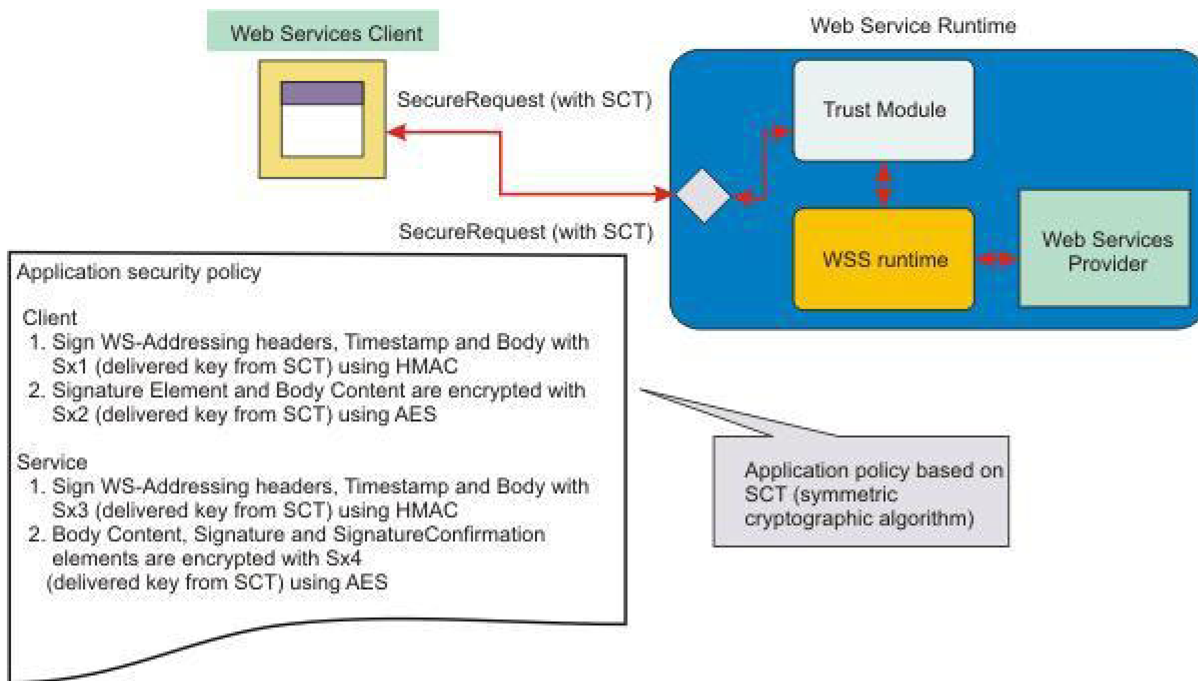


Figure 26. Securing application messages with keys derived from secret of the security context token

A similar scenario except with Web Services Reliable Messaging (WS-ReliableMessaging) is possible from the WS-SecureConversation prospective. See the example for establishing a security context token to secure reliable messaging.

Flow for establishing a security context token to secure reliable messaging:

This example scenario includes functions that are required for the composite scenario of Web Services Reliable Messaging (WS-ReliableMessaging), WS-SecureConversation, and WS-Trust. The scenario describes how to use WS-SecureConversation with WS-ReliableMessaging, the scenario is described from the WS-SecureConversation perspective.

The flow of this Web Services Reliable Messaging (WS-ReliableMessaging) scenario is very similar to the flow of the WS-SecureConversation scenario, and the exchange of the application messages is very similar to the Secure Conversation scenarios. The main difference in the two example scenarios is that the WS-ReliableMessaging sequence is secured with the security context token and scopes the WS-ReliableMessaging sequence to the security context token.

The following figure describes a summary of the message flows that are required to establish a security context token to secure reliable messaging.

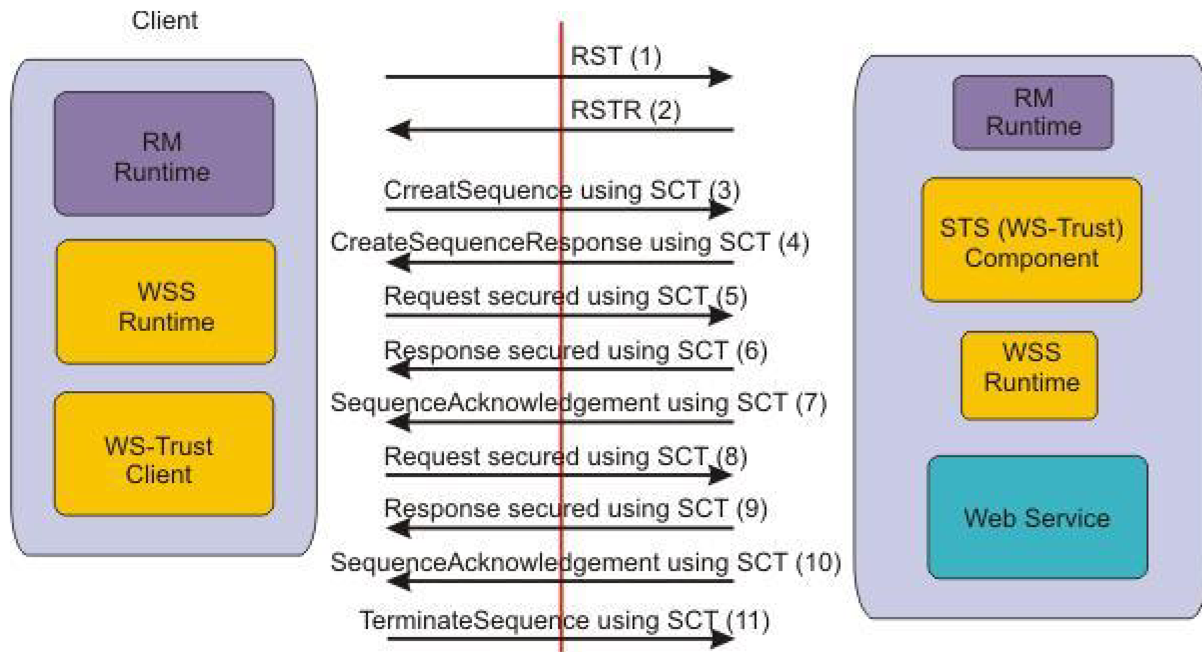


Figure 27. Messages exchange for the SCT and reliable messaging

Scenario

The WS-ReliableMessaging sequence is secured with the security context token and is scoping the WS-ReliableMessaging sequence to the security context token. This scenario focuses on the message exchanges that are using the security context token in the overall flow.

Note: The exact detail of how WS-ReliableMessaging is validating the WS-ReliableMessaging sequence, with respect to the security context token scoping, is not described.

Typically, to use secure conversation and a security context token to secure reliable messaging, the following steps are involved;

- The WS-ReliableMessaging run time calls APIs from the Web Services Security run time to get the UUID of the security context token for the session and also the API for serializing and deserializing the security context token for managed persistent for reliable recovery.

  Because of the security nature of the security context token, the WS-ReliableMessaging protocol makes sure that the serialized security context token in persistent store is protected.
- If there is already a security context token established the UUID of the existing security context token is returned to WS-ReliableMessaging. If there is no security context token already established, the Web Services Security run time initiates a call to the recipient to establish the security context token.

  The latter case is similar to the Secure Conversation scenario.
- After the WS-ReliableMessaging run time acquires the UUID of the security context token, the WS-ReliableMessaging run time scopes the CreateSequence message to the security context token by using the SecurityTokenReference (STR) argument in the CreateSequence message and responds with the CreateSequenceResponse message.

  The exchange of the application messages is very similar to the WS-SecureConversation scenario.
- The WS-ReliableMessaging run time responds with the CreateSequenceResponse message.

  The exchange of the messages is very similar to the exchange in the WS-SecureConversation scenario.
- The WS-ReliableMessaging run time sends a SequenceAcknowledgement message to acknowledge that the message is properly delivered and secured by the security context token.

- Finally, the WS-ReliableMessaging run time sends a TerminateSequence message to terminate the sequence and is secured by the security context token.

Enabling the distributed cache using synchronous update and token recovery:

To support secure conversation in a cluster environment, the distributed cache stores the shared state information. Version 7.0 and later of WebSphere Application Server uses MBeans to improve synchronous update of the cache across the cluster. In addition, persistent token support is provided by storing the token data in a database.

About this task

Synchronous update of shared information in the distributed cache is implemented in the product using an MBean solution. When update of the shared state information across cluster members is required, a synchronous blocking call is issued to replicate the token state changes to all the servers in the cluster. This solution removes the limitations of using session affinity for secure conversation in a cluster environment.

Perform the following high-level steps to enable distributed cache when using secure conversation for message-level protection in a cluster environment.

Procedure

1. In the administrative console for WebSphere Application Server, click **Services** > **Security cache**.
2. Click the check box to select the **Enable distributed cache** setting.
3. The distributed cache setting has three options. The first option is **Synchronous update of cluster members**. This option is selected by default, enabling the runtime to update all the cluster members with token information synchronously. If this is selected, then session affinity does not have to be enabled.

   The second option is **Asynchronous update of cluster members**, which you can select by clicking the corresponding radio button. For this option to work successfully, session affinity must be enabled. For information on enabling session affinity, read the topic Enable distributed cache and session affinity when using Secure Conversation. If Asynchronous update of cluster members is selected, skip steps 4 and 5.

   The third option is **Token recovery support**. To enable this option, click the corresponding radio button, then select a data source (database) from the **Cell level data sources** menu list. To create a data source, click the **Manage data sources** link. If Token recovery support is selected, skip steps 4 and 5.
4. This step is needed only if Synchronous update of cluster members is selected. Create a replication domain, as follows:
   a. In the administrative console, click **Environment > Replication domains > New**.
   b. Enter a name for the domain. For example, **ABCDomain**.
   c. In the Number of replicas section, click the radio button to select the **Entire Domain** option.
   d. Click **OK**, then **Save**, to save the modified configuration.
5. This step is needed only if Synchronous update of cluster members is selected. Enable the dynamic cache by performing the following steps for each server in the cluster:
   a. In the administrative console, click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** > **Container Services** > **Dynamic cache service**.
   b. Select the **Enable cache replication** option.
   c. Select the replication domain name that you created in the previous step. For example, **ABCDomain**.
   d. Under Replication type. select **Both push and pull** from the menu list.
   e. Click **OK**, then click **Save** to save the modified configuration.

Different clusters should use different replication domains. Likewise, cluster members from the same cluster should use the same replication domain. This ensures that synchronous update of cluster members performed by the Web Services Security runtime, and dynamic replication service updates of cluster members performed by the WebSphere Application Server dynamic cache runtime, are in sync.

Results

When the configuration steps are complete, you have enabled the distributed cache with either the default option, which is synchronous update of cluster members, or with asynchronous cluster update or with token recovery support. The token recovery support option uses a JDBC database to store the token state. This provides failover support for high availability of the token. If the server processing the request does not have access to the secure conversation token, the request fails, producing an error such as “null secure conversation token” or “invalid secure conversation token”.

Configuring the token generator and token consumer to use a specific level of WS-SecureConversation:

Use the administrative console to configure the token generator or token consumer to use a specific level of the WS-SecureConversation OASIS specification standard. Select one of the two levels of token types supported: Secure Conversation Token v200502, or Secure Conversation Token v1.3.

About this task

WebSphere Application Server supports two levels of the OASIS standard for WS-SecureConversation including both the submission draft version (February 2005 draft specification) and version 1.3 of the standard, which was approved on March 1, 2007. Using the administrative console, configure the token generator so that the appropriate token type for a specific level of the standard is issued when a security token is requested.

Procedure

1. Log on to the administrative console and navigate to the panel where the token generator is configured by clicking **Services** > **Policy sets** > **General provider policy set bindings** or **General client policy set bindings**.
2. Click on the name of the binding you want to edit.
3. Click the **WS-Security** policy in the Policies table.
4. Click the **Authentication and protection** link in the Main message security policy bindings section.
5. Click **New token** to create a new token generator or consumer, or click an existing token link from the Protection Tokens table.
6. Enter a token name, then use the Token type drop-down menu to select a secure conversation token type.
   - To specify a submission draft token type, select **Secure Conversation Token v200502**.
   - To specify a version 1.3 token type, select **Secure Conversation Token v1.3**.
7. The local name is populated according to the token type you selected, as follows:
   - Local name for the submission draft token type: http://schemas.xmlsoap.org/ws/2005/02/sc/sct
   - Local name for the version 1.3 token type: http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512

   The URI field is also filled in based on the token type.
8. Click to deselect the option **Tolerate Secure Conversation Token v200502** if you want to enforce use of only the version 1.3 tokens. This option specifies whether the provider should handle both Secure Conversation Token version 1.3 and Secure Conversation Token v200502. By default, the provider handles both versions.
9. Click **Apply** to create a secure conversation token of the selected type.

Web Services Secure Conversation standard:

Web Services Secure Conversation (WS-SecureConversation) is a proposed Organization for the Advancement of Structured Information Standards (OASIS) standard that defines mechanisms for establishing and sharing security contexts, and deriving keys from security contexts, to enable a secure conversation.

The base Web Services Security (WS-Security) standard from OASIS defines how to digitally sign and encrypt the SOAP message to provide message level protection. The standard also defines how to attach and reference a security token for digital signature and encryption. However, it does not provide session-based protection when a long series of related messages were exchanged. The WS-Security specification focuses on the *message authentication model*. This approach, while useful in many situations, could be subject to several forms of attack.

The WS-SecureConversation specification introduces the concept of a security context and its usage. The security context token is a new WS-Security token type that represents the security context abstract concept. The token is identified by a URI and consists of negotiated keys as well as other security related properties. The *context authentication model* authenticates a series of messages and, therefore, addresses these concerns. The context authentication model increases the overall performance and security of the subsequent exchanges, but it requires additional communications when authentication happens prior to normal application exchanges.

Version 1.0 of the OASIS WS-SecureConversation specification defines extensions that build on the Web Services Security (WS-Security) and Web Services Trust (WS-Trust) standards to provide secure communication across one or more messages.

IBM, Microsoft, and other vendors have been working on the WS-SecureConversation specification since 2004. A draft of this document was jointly published in February, 2005. The WS-SecureConversation draft was submitted to the OASIS Web Service Secure Exchange Technical Committee (WS-SX TC), which was formed in December 2005, along with Web Services Trust (WS-Trust) and Web Services Security Policy (WS-SecurityPolicy) drafts in order to begin the standardization process.

A revised Version 1.1 draft of the WS-SecureConversation specification standard was submitted to OASIS in February 2005 and further defines the extensions in Version 1.0. This specification defines extensions to allow security context establishment and sharing, and session key derivation. These extensions allow contexts to be established and potentially more efficient keys or new key material to be exchanged.

The most recent version of the specification standard is version 1.3, which was approved by the WS-SX TC on March 1, 2007. Key requirements in this level of the specification include derived keys and per-message keys, and extensible security contexts. WebSphere Application Server adds support for version 1.3 of WS-SecureConversation, providing improved error handling using the standard fault codes as defined in the specification.

The Web Services Secure Conversation (WS-SecureConversation) standard is a building block that is used in conjunction with the other web service and application-specific protocols such as Web Services Security and Web Services Trust to accommodate a wide variety of security models and technologies. WS-SecureConversation is built on top of the WS-Security and WS-Trust models to provide secure communication between services. The WS-SecureConversation draft specification describes how to establish a security context token between two parties, and the WS-Trust specification describes how to issue and exchange security tokens.

This WS-SecureConversation draft specification includes extensions to Web Services Security and the following:

- Describes the security context token.
- Defines how security contexts are established.

- Describes how security contexts are amended, renewed, and cancelled. Amending context is not supported by WebSphere Application Server.
- Specifies how derived keys are computed.
- Specifies how to associate a specific security context with an action, if multiple security contexts exist.

WebSphere Application Server supports the client establishing a secured conversation with the target service endpoint.

WebSphere Application Server supports the OASIS Version 1.1 submission draft, which became available in February 2005. The WebSphere Application Server does not support all of the functions in the submission draft. WebSphere Application Server support of WS-SecureConversation focuses on:
- A security context token that is established between the initiating party and the recipient party.
- The operations that are supported on security context token, such as Issue token, Renew token, and Cancel token.
- The derived key (both explicit and implied)

Secure conversation provided with WebSphere Application Server does not provide support for a security context token (SCT) that is acquired from a third-party trust server, and does not provide support for a security context token that is created by the client.

For information about WS-SecureConversation:
- See the IBM developerWorks website.
- See the schema for this specification: WS-SecureConversation schema
- Refer to the following namespace prefixes that are used for WS-SecureConversation: http://schemas.xmlsoap.org/ws/2005/02/sc and http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512

Trust service:

The security token service that is provided by WebSphere Application Server is called the trust service. The WebSphere Application Server trust service uses the secure messaging mechanisms of Web Services Trust (WS-Trust) to define additional extensions for the issuance, exchange, and validation of security tokens.

Web Services Trust (WS-Trust) is an OASIS standard that enables security token interoperability by defining a request/response protocol. This protocol allows SOAP actors, such as a web services client, to request of some trusted authority that a particular security token be exchanged for another.

WebSphere Application Server is not providing a full security token service that implements all the contents of the WS-Trust draft specification. The WebSphere Application Server support of WS-Trust focuses on establishing a security context token for secure conversation. WebSphere Application Server supports many of the security features described in version 1.3 of the WS-Trust OASIS standard, dated March 19, 2007.

Third party WS-Trust client

WebSphere Application Server does not provide a WS-Trust client implementation. You can choose to use a third-party WS-Trust-enabled client but, if you do, WebSphere Application Server does not support a third-party trust-enabled client. A trust client can facilitate the generation of these soap messages and the processing of the response, but the client is not required.

WebSphere Application Server focuses on the issuing, renewing, and canceling of the security context token for Web Services Secure Conversation (WS-SecureConversation).

The WS-Trust specification must be followed to make requests of the trust service. This specification includes the use of Web Services Addressing (WS-Addressing) headers. The WS-Addressing headers are specified in both the August 2004 or the August 2005 specifications. Per the specification, the SOAP body must consist of a single RequestSecurityToken (RST) element. This element can contain sub-elements as defined in the WS-Trust and WS-SecureConversation specifications.

You can secure the WS-Trust SOAP messages by using the bootstrap policy that is defined in the policy set. The bootstrap security policy is invoked in the process of an initiator establishing communication with an application service. Initial requests to services other than the application service are secured by using the bootstrap policy. These initial requests typically involve one or more requests to a security token service (STS), such as the WebSphere Application Server trust service. An example of a request might be acquiring the security context token necessary for WS-SecureConversation. An *initiator* is the role that initiates the original request and, in most cases, it is the client. The client bootstrap policy set must correspond to the trust service issue and renew attached policy sets for the endpoint. The trust service cancel and validate attached policy sets for the endpoint must correspond to the client's application policy set.

Websphere Application Server provides two ways to secure SOAP messages that are destined for the trust service. One way is to use the bootstrap policy that is defined in the policy set. A second way is to use the Web Services Security API (WSS API). Your application might use the WSS API to acquire the security context token for the programmatic, API-based WS-SecureConversation.

For Secure Conversation, a request from the client to an endpoint service is suspended while a new (second) request is generated and processed by the trust service. The security context token returned with the second request is used to derive keys that secure communications with the service.

High-level trust service functions

The following list includes WS-Trust-related functions that are currently supported in WebSphere Application Server. The list is not exhaustive and it focuses only on the high-level functions.

- The trust service component is embedded into and available on each WebSphere Application Server that processes the WS-Trust protocol messages.
- Communication is accomplished through the RequestSecurityToken (RST), RequestSecurityTokenCollection (RSTC), RequestSecurityTokenResponse (RSTR), and RequestSecurityTokenResponseCollection (RSTRC).

  Note: An RST request can be made to an external security token service (trust service). However, the restriction is that security context token, which is needed for WS-SecureConversation, must be provided by the WebSphere Application Server trust service.
- A security policy for each of the WS-Trust operations (issue, cancel, validate, and renew).
- Pre-configured Security Context Token provider which issues tokens for specific URL.
- Specification of a token provider's token-specific parameters (for example, expiration time).
- A security context token for WS-SecureConversation.
- Caching support for the security context token in both cluster and non-cluster environments. WebSphere Application Server issues security context tokens when requested if the request meets the security requirements. However, WS-SecureConversation provided by WebSphere Application Server only processes security context tokens that are issued by WebSphere Application Server.
- Note that WebSphere Application Server trust service only supports the security context token.
- Trust service supports both the Submission specification (2004/08) and the final specification (2005/08) versions of WS-Addressing.
- Trust service uses a default policy set called TrustServiceSecurityDefault, which includes WS-Security and WS-Addressing and provides default security for the issue and renew operations.

- Trust service uses a second default policy set called TrustServiceSymmetricDefault, which includes WS-Security and WS-Addressing and provides default security for the cancel and validate operations.

Trust service functions that are not supported

The following high-level WS-Trust functions that are not supported in WebSphere Application Server. The list is not exhaustive, and the list focuses only on key functions:

- No negotiation and exchange protocols are supported.
- No other token types are currently supported out of the box; only the security context token is supported.
- No Trust10 specifications from WS-SecurityPolicySet are supported.
- No unsolicited RequestSecurityTokenResponse (RSTR) is supported.
- A Request Security Token (RST) request cannot be issued to an external security token service (STS) to establish a secure conversation; only the embedded trust service is currently supported.
- Policy requests that are contained in the RST are not honored.
- The ability to amend a token (the amend operation) is not supported.
- A dedicated external endpoint for access to the token service is not supported; only the embedded trust service is currently supported.
- The trust services does not support the entropy element that contains an EncryptedKey.
- Delegation and forwarding are not supported.
- The OnBehalfOf element is not supported.
- The Key Exchange Token (KET) binding is not supported.

Trust service operations

WebSphere Application Server specifically supports the ability of the trust service, on behalf of the endpoint, to issue a security context token for WS-SecureConversation. The token-issuing support is currently limited only to the security context token. There is also trust policy management for defining a policy for the trust service to issue, cancel, validate, or renew tokens.

The token service supports the WS-Trust schema namespace. Within this namespace the following actions are supported:

- http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue
- http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel
- http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate
- http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew

The token service also supports the WS-SecureConversation schema namespace. Within this namespace the following actions are supported:

- http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT
- http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT/Cancel
- http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT/Renew

An inbound RST for the security context token issue operation must contain an Entropy element. The Entropy element must contain a BinarySecret. The trust services does not support the Entropy element that contains an EncryptedKey.

Note that the trust service does not support unsolicited RSTR actions. In addition, the ability to amend a token is not supported by WebSphere Application Server. Also, see the section titled Trust service functions that are not supported.

Trust policy set-related files

The default trust service policy set for issue and renew is TrustServiceSecurityDefault. You can set up the corresponding policy set and binding for each service endpoint URL.

Security context token:

Web Services Trust (WS-Trust) and Web Services Secure Conversation (WS-SecureConversation) support in the application server provides the ability to issue a security context token (SCT). Requests for a security context token are processed by the security token service.

The security token service for WebSphere Application Server is called the trust service. However, the application server does not provide a full security token service that implements all the contents of the WS-Trust specification.

The secure session is referred to as *secure conversation* because the message protocols that are used are defined by WS-SecureConversation and WS-Trust. WebSphere Application Server supports secure conversation.

To request a security context token, a RequestSecurityToken (RST), which is defined by WS-Trust and WS-SecureConversation protocols, is sent to the service endpoint to which you are setting up a secure conversation. These requests are transparently rerouted to the trust service. The trust service processes the RST and responds with a RequestSecurityTokenResponse (RSTR). This response is returned to the requestor as if it was generated by the endpoint service.

The WebSphere Application Server token provider support is limited to the Security Context Token provider. WS-SecureConversation in the application server focuses on the establishing of the security context token between the initiating party and the recipient party for secure conversation.

WebSphere Application Server includes caching support for the Security Context Token in both cluster and non-cluster environments as well as on both the client and server. WebSphere Application Server also provides trust policy set management for each of the trust service operations: issue, cancel, validate, and renew. Trust system policy sets can be managed for each of these trust operations relative to an explicit service endpoint or the trust service default. The default trust service policy set for a trust operation is enforced when there is not an explicit attachment.

See the information about Web Services Trust for the WS-Trust functions that are supported.

For the security context token, you can:
- Configure the security context token provider for WS-SecureConversation, providing issue, renew and cancel operations.
- Configure the trust service to issue a security context token for access to a specific endpoint service (target).
- Configure the security requirements for access to the trust service and applications. WebSphere Application Server provides pre-configured application policy sets and trust service policy sets to assist with this configuration.
- Define a system policy for each of the four trust service operations: issue, cancel, validate, and renew. These policies are configured for the default or a specific endpoint service. Note that the amend operation is not supported.

The Security Context Token provider does not support the following operations:
- WS-SecureConversation amend
- Negotation to establish Secure Conversation
- WS-Trust key exchange requests

- Client-initiated RequestSecurityTokenResponse (RSTR) and RequestSecurityTokenResponseCollection (RSTRC) requests
- WS-SecurityPolicy trust assertions

Definitions

To better understand security tokens, the following terms are defined:

security token
A security token represents a collection of claims.

security context
A security context is an abstract concept that refers to an established authentication state and negotiated key or keys that can have additional security-related properties. A security context needs to be created and shared by the communicating parties before being used. A security context is shared among the communicating parties for the lifetime of a communications session and a security context token is the wire representation of this abstract security context.

WebSphere Application Server does not support a security context token created by one of the communicating parties and propagated with a messageWebSphere Application Server does not support creating a security context token through negotiation and exchanges.

security context token
A security context token is a wire representation of that security context abstract concept, which allows a context to be named by a URI and to be used with Web Services Security. A secured communication with a security context token between two parties is realized with WS-Trust and WS-SecureConversation.

security token service
A security token service (STS) is a web service that issues security tokens, meaning it makes assertions that are based on evidence that it trusts, to whoever trusts it (or to specific recipients).

Trust service
The trust service is the security token service and supporting code that is provided by Websphere Application Server.

RequestSecurityToken (RST)
A RST is a message sent to a security token service to request a security token.

RequestSecurityToken Response (RSTR)
A RSTR is a response to a request for a security token from a security token service to a requestor after receiving an RST message.

To communicate trust, a service requires proof, such as a signature, to prove knowledge of a security token or set of security tokens. A service itself can generate tokens or it can rely on a separate security token service to issue a security token with its own trust statement. Note that, for some security token formats, communicating trust can just be a re-issuance or a co-signature that forms the basis of trust brokering.

Syntax for the <wsc:SecurityContextToken> element

A security context is shared among the communicating parties for the lifetime of a communications session and a security context token is the wire representation of this abstract security context.

In the WS-SecureConversation specification, a security context is represented by the <wsc:SecurityContextToken> security token. The following URI represents the security context token type that is required to establish a secure conversation.

```
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
```

The syntax for <wsc:SecurityContextToken> element is as follows:

```
<wsc:SecurityContextToken wsu:Id="..." ...>
    <wsc:Identifier>...</wsc:Identifier>
    <wsc:Instance>...</wsc:Instance>
    ...
</wsc:SecurityContextToken>
```

The security context token does not support references to it by using key identifiers or key names. All references must use an ID (to a *wsu:Id* attribute) or use a URI reference, <wsse:Reference>, to the <wsc:Identifier> element in the security context token.

RST and RSTR examples to issue a security token

This example shows a RST request to issue a security token. The URI `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct`, which is used in this example, represents the token type:

```
<wsc:SecurityContextToken>
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

   <soapenv:Header>
  <wsa:To xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   http://localhost:80/WSSampleSei/EchoService
  </wsa:To>
  <wsa:MessageID xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   fc0632828e1252b4:487cee53:11cbfa7916e:-7fb6
  </wsa:MessageID>
  <wsa:Action xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT
  </wsa:Action>
 </soapenv:Header>

 <soapenv:Body>

  <wst:RequestSecurityToken
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
   Context="http://www.ibm.com/login/">
   <wst:TokenType>
    http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
   </wst:TokenType>
   <wst:RequestType>
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
   </wst:RequestType>
   <wsp:AppliesTo
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    -
    <wsa:EndpointReference
     xmlns:wsa="http://www.w3.org/2005/08/addressing">
     <wsa:Address>
      http://localhost:80/WSSampleSei/EchoService
     </wsa:Address>
    </wsa:EndpointReference>
   </wsp:AppliesTo>
   <wst:Entropy>
    <wst:BinarySecret
     Type="http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce">
     zb//KsawV6DmfC8kB6vNOQ==
    </wst:BinarySecret>
   </wst:Entropy>
   <wst:KeySize>128</wst:KeySize>
  </wst:RequestSecurityToken>
 </soapenv:Body>
</soapenv:Envelope>
```

This example shows a RSTR request to issue a security token:

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

 <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <wsa:Action>
   http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/SCT
  </wsa:Action>
```

```
  <wsa:RelatesTo>
   fc0632828e1252b4:487cee53:11cbfa7916e:-7fb6
  </wsa:RelatesTo>
 </soapenv:Header>

 <soapenv:Body>
  <wst:RequestSecurityTokenResponseCollection
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
   <wst:RequestSecurityTokenResponse
    Context="http://www.ibm.com/login/">
    <wst:RequestedSecurityToken>
     <wsc:SecurityContextToken
      xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
      wsu:Id="uuid:FFA51A32EB818FB6EA1222986227363">
      <wsc:Identifier>
       uuid:FFA51A32EB818FB6EA1222986227346
      </wsc:Identifier>
      <wsc:Instance>
       uuid:FFA51A32EB818FB6EA1222986227345
      </wsc:Instance>
     </wsc:SecurityContextToken>
    </wst:RequestedSecurityToken>
    <wsp:AppliesTo
     xmlns:wsp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
     <wsa:EndpointReference
      xmlns:wsa="http://www.w3.org/2005/08/addressing">
      <wsa:Address>
       http://localhost:80/WSSampleSei/EchoService
      </wsa:Address>
     </wsa:EndpointReference>
    </wsp:AppliesTo>
    <wst:RequestedProofToken>
     <wst:ComputedKey>
      http://docs.oasis-open.org/ws-sx/ws-trust/200512/CK/PSHA1
     </wst:ComputedKey>
    </wst:RequestedProofToken>
    <wst:Entropy>
     <wst:BinarySecret
      Type="http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce">
      rF1Yp5zhRhamLQNPAOm4TA==
     </wst:BinarySecret>
    </wst:Entropy>
    <wst:Lifetime>
     <wsu:Created
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
      2008-10-02T22:23:44.765Z
     </wsu:Created>
     <wsu:Expires
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
      2008-10-02T22:35:44.765Z
     </wsu:Expires>
    </wst:Lifetime>
    <wst:RequestedAttachedReference>
     <wsse:SecurityTokenReference
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <wsse:Reference
       URI="#uuid:FFA51A32EB818FB6EA1222986227363"
       ValueType="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct" />
     </wsse:SecurityTokenReference>
    </wst:RequestedAttachedReference>
    <wst:RequestedUnattachedReference>
     <wsse:SecurityTokenReference
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <wsse:Reference
       URI="uuid:FFA51A32EB818FB6EA1222986227346"
       ValueType="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct" />
     </wsse:SecurityTokenReference>
    </wst:RequestedUnattachedReference>
    <wst:Renewing Allow="true" OK="false" />
    <wst:KeySize>128</wst:KeySize>
   </wst:RequestSecurityTokenResponse>
  </wst:RequestSecurityTokenResponseCollection>
 </soapenv:Body>
</soapenv:Envelope>
```

RST and RSTR examples to cancel a security token

This example shows a RST request to cancel a security token.

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

 <soapenv:Header>
  <wsa:To xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   http://localhost:80/WSSampleSei/EchoService
  </wsa:To>
  <wsa:MessageID xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   fc0632828e1252b4:-270287b7:11cc22c16ed:-7fa8
  </wsa:MessageID>
  <wsa:Action xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel
  </wsa:Action>
 </soapenv:Header>

 <soapenv:Body>
  <wst:RequestSecurityToken
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
   Context="http://www.ibm.com/login/">
   <wst:TokenType>
    http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
   </wst:TokenType>
   <wst:RequestType>
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
   </wst:RequestType>
   <wsp:AppliesTo
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsa:EndpointReference
     xmlns:wsa="http://www.w3.org/2005/08/addressing">
     <wsa:Address>
      http://localhost:80/WSSampleSei/EchoService
     </wsa:Address>
    </wsa:EndpointReference>
   </wsp:AppliesTo>
   <wst:CancelTarget>
    <wsc:SecurityContextToken
     xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
     wsu:Id="uuid:AC4764EB4BE91011501223028453769">
     <wsc:Identifier>
      uuid:AC4764EB4BE91011501223028453768
     </wsc:Identifier>
     <wsc:Instance>
      uuid:AC4764EB4BE91011501223028453751
     </wsc:Instance>
    </wsc:SecurityContextToken>
   </wst:CancelTarget>
  </wst:RequestSecurityToken>
 </soapenv:Body>
</soapenv:Envelope>
```

This example shows a RSTR request to cancel a security token:

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:wsa="http://www.w3.org/2005/08/addressing">

 <soapenv:Header>
        <wsa:Action>
          http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Cancel
      </wsa:Action>
      <wsa:RelatesTo>
          fc0632828e1252b4:-270287b7:11cc22c16ed:-7fa8
      </wsa:RelatesTo>
   </soapenv:Header>

  <soapenv:Body>
      <wst:RequestSecurityTokenResponse
           Context="http://www.ibm.com/login/"
                xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
```

```
        <wst:RequestedTokenCancelled>
      </wst:RequestSecurityTokenResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

RST and RSTR examples to renew a security token

This example shows a RST request to renew a security token.

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

 <soapenv:Header>
  <wsa:To xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   http://localhost:80/WSSampleSei/EchoService
  </wsa:To>
  <wsa:MessageID xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   fc0632828e1252b4:487cee53:11cbfa7916e:-7f8e
  </wsa:MessageID>
  <wsa:Action xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew
  </wsa:Action>
 </soapenv:Header>

  <soapenv:Body>
  <wst:RequestSecurityToken
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
   Context="http://www.ibm.com/login/">
   <wst:TokenType>
    http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
   </wst:TokenType>
   <wst:RequestType>
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/Renew
   </wst:RequestType>
   <wst:RenewTarget>
    <wsc:SecurityContextToken
     xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
     wsu:Id="uuid:FFA51A32EB818FB6EA1223026418869">
     <wsc:Identifier>
      uuid:FFA51A32EB818FB6EA1223026418868
     </wsc:Identifier>
     <wsc:Instance>
      uuid:FFA51A32EB818FB6EA1223026418867
     </wsc:Instance>
    </wsc:SecurityContextToken>
   </wst:RenewTarget>
   <wsp:AppliesTo
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsa:EndpointReference
     xmlns:wsa="http://www.w3.org/2005/08/addressing">
     <wsa:Address>
      http://localhost:80/WSSampleSei/EchoService
     </wsa:Address>
    </wsa:EndpointReference>
   </wsp:AppliesTo>
   <wst:Entropy>
    <wst:BinarySecret
     Type="http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce">
     U8rH9l/wLV1gpsBf/yCooA==
    </wst:BinarySecret>
   </wst:Entropy>
   <wst:KeySize>128</wst:KeySize>
  </wst:RequestSecurityToken>
 </soapenv:Body>
</soapenv:Envelope>
```

This example shows a RSTR request to renew a security token:

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

 <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <wsa:Action>
```

```
   http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal
  </wsa:Action>
  <wsa:RelatesTo>
   fc0632828e1252b4:487cee53:11cbfa7916e:-7f8e
  </wsa:RelatesTo>
 </soapenv:Header>

 <soapenv:Body>
  <wst:RequestSecurityTokenResponse
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
   Context="http://www.ibm.com/login/">
   <wst:RequestedSecurityToken>
    <wsc:SecurityContextToken
     xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
     wsu:Id="uuid:FFA51A32EB818FB6EA1223026990448">
     <wsc:Identifier>
      uuid:FFA51A32EB818FB6EA1223026418868
     </wsc:Identifier>
     <wsc:Instance>
      uuid:FFA51A32EB818FB6EA1223026990447
     </wsc:Instance>
    </wsc:SecurityContextToken>
   </wst:RequestedSecurityToken>
   <wst:Entropy>
    <wst:BinarySecret
     Type="http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce">
     lFkKSI/pajtTZzRpQalNMA==
    </wst:BinarySecret>
   </wst:Entropy>
   <wst:Lifetime>
    <wsu:Created
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
     2008-10-03T09:43:07.421Z
    </wsu:Created>
    <wsu:Expires
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
     2008-10-03T09:55:07.421Z
    </wsu:Expires>
   </wst:Lifetime>
   <wst:RequestedAttachedReference>
    <wsse:SecurityTokenReference
     xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
     <wsse:Reference
      URI="#uuid:FFA51A32EB818FB6EA1223026990448"
      ValueType="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct">
     </wsse:Reference>
    </wsse:SecurityTokenReference>
   </wst:RequestedAttachedReference>
   <wst:Renewing Allow="true" OK="false"></wst:Renewing>
  </wst:RequestSecurityTokenResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

RST and RSTR examples to validate a security token

This example shows a RST request to validate a security token.

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

 <soapenv:Header>
  <wsa:To xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   http://localhost:80/WSSampleSei/EchoService
  </wsa:To>
  <wsa:MessageID xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   fc0632828e1252b4:-673f2c18:11cc328886a:-7fa7
  </wsa:MessageID>
  <wsa:Action xmlns:wsa="http://www.w3.org/2005/08/addressing"
   soapenv:mustUnderstand="0">
   http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate
  </wsa:Action>
 </soapenv:Header>

  <soapenv:Body>
  <wst:RequestSecurityToken
```

```
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
   Context="http://www.ibm.com/login/">
   <wst:TokenType>
    http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
   </wst:TokenType>
   <wst:RequestType>
    http://docs.oasis-open.org/ws-sx/ws-trust/200512/Validate
   </wst:RequestType>
   <wst:ValidateTarget>
    <wsc:SecurityContextToken
     xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
     wsu:Id="uuid:6B77A2DA28C1E523BD1223045150688">
     <wsc:Identifier>
      uuid:6B77A2DA28C1E523BD1223045150687
     </wsc:Identifier>
     <wsc:Instance>
      uuid:6B77A2DA28C1E523BD1223045150670
     </wsc:Instance>
    </wsc:SecurityContextToken>
   </wst:ValidateTarget>
   <wsp:AppliesTo
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsa:EndpointReference
     xmlns:wsa="http://www.w3.org/2005/08/addressing">
     <wsa:Address>
      http://localhost:80/WSSampleSei/EchoService
     </wsa:Address>
    </wsa:EndpointReference>
   </wsp:AppliesTo>
  </wst:RequestSecurityToken>
 </soapenv:Body>
</soapenv:Envelope>
```

This example shows a RSTR request to validate a security token:

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

 <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <wsa:Action>
   http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal
  </wsa:Action>
  <wsa:RelatesTo>
   fc0632828e1252b4:-673f2c18:11cc328886a:-7fa7
  </wsa:RelatesTo>
 </soapenv:Header>

 <soapenv:Body>
  <wst:RequestSecurityTokenResponse
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
   Context="http://www.ibm.com/login/">
   <wst:Status>
    <wst:Code>
     http://docs.oasis-open.org/ws-sx/ws-trust/200512/status/valid
    </wst:Code>
   </wst:Status>
  </wst:RequestSecurityTokenResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

For additional information, review the two example scenario topics that discuss establishing the security context token.

System policy sets:

A policy set is a named collection of Quality of Service (QoS) policies. You can use either the administrative console or the wsadmin commands to manage system policy sets. Policy sets can be created, deleted, copied, imported or exported.

A policy set can be shared by multiple resources, such as applications, services, inbound or outbound service endpoints, and operations. Default policy sets are installed using profile augmentation. A policy set

can also be imported. A policy set does not have its own bindings. You must attach a policy set to a resource, and then assign a binding to the attachment.

Note: When attempting to connect to a web service from a thin client, verify that the resources that you are specifying are valid before running the updatePolicySetAttachment command. No configuration changes are made if the requested resource does not match a resource in the attachment file for the application.

A client application can dynamically select a policy suite (reference by name from an application-level policy suites list). Options shown in the administrative console list are based on the type of template that is selected to create the policy set. For example, the SecureConversation policy type is made up of policies for both WSSecurity and WSAddressing.

There are two types of policy sets:
- Application policy sets
- System/trust policy sets

WebSphere Application Server provides predefined system policy sets. For example, WebSphere Application Server provides the following system policy sets by default for the security trust service:
- TrustServiceSecurityDefault

  This trust policy set specifies the asymmetric algorithm as well as the public and private keys to provide message security. Message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using RSA. Message confidentiality is provided by encrypting the body and signature using RSA. This policy set follows the WS-Security specifications for the issue and renew trust operation requests.
- TrustServiceSymmetricDefault

  This policy set specifies the symmetric algorithm as well as the derived keys to provide message security. Message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using HMAC-SHA1. Message confidentiality is provided by encrypting the body and signature using AES. This policy set follows the WS-Security and Secure Conversation specifications for validate and cancel trust operation requests.
- SystemWSSecurityDefault

  This policy set specifies the asymmetric algorithm and both the public and private keys to provide message security. Message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using RSA encryption. Message confidentiality is provided by encrypting the body and signature using RSA encryption

You cannot edit default system policy sets. However, you can create your own custom system policy set, which can be edited later. Copy or export a default or existing custom system policy set to create the new custom policy set. System policy sets can also be imported from a predefined location, or from the default repository. Add one or more policies to each policy set. For example, add any of the following existing policies:
- Custom properties
- HTTP transport
- JMS transport
- SSL Transport
- WS-Addressing
- WS-Security

The HTTP transport policy can be used for HTTPS, basic authorization, compression, and binary encoding transport methods.

Web Services Trust standard:

Web Services Trust (WS-Trust) is a proposed Organization for the Advancement of Structured Information Standards (OASIS) standard that enables security token interoperability by defining a request/response protocol. This protocol allows SOAP actors, such as a web services client, to request of some trusted authority that a particular security token be exchanged for another. The trust service, which is provided with WebSphere Application Service, uses the secure messaging mechanisms of WS-Trust to define additional extensions for the issuance, exchange, and validation of security tokens.

WS-Trust defines a request and response protocol for security token exchange. A client sends a RequestSecurityToken (RST) to a security token service. The request includes the security token that the client is asking to be exchanged. The security token service responds back with a RequestSecurityTokenResponse (RSTR) that contains the new token.

In addition to the token exchange, the WS-Trust request/response protocol is general enough to support token *issuance*, where the client presents a claim to the trust service for the service to authorize through the issuance of a corresponding security token. Token *validation* is where the client presents a token to the trust service and asks that its validity be determined.

Also, WS-Trust enables the issuance and dissemination of credentials within different trust domains. To secure a communication between two parties, the two parties must exchange security credentials (either directly or indirectly). Each party must first determine if they can trust the asserted credentials of the other party.

The OASIS WS-Trust specification defines extensions to Web Services Security (WS-Security) for issuing and exchanging security tokens and for providing ways to establish and access the presence of trust relationships. Using these extensions, applications can engage in secure communication, and these extensions are designed to work with the general web services framework. The general web services framework includes the WSDL service descriptions, UDDI businessServices and bindingTemplates, and SOAP messages.

The WebSphere Application Server support of WS-Trust focuses on establishing a security context token for Web Services Secure Conversation (WS-SecureConversation). The WS-Trust support focuses on the four actions for the security context token: issue, renew, validate, and cancel. Also supported for WS-Trust Version 1.3 are collection requests for the same actions: issue, renew, validate and cancel. The major component for WS-Trust that WebSphere Application Server supports is the security token service, which is referred to as the trust service.

Support for submission draft and approved levels of the WS-Trust standard

Version 6.1 and later of WebSphere Application Server supports the WS-Trust 2005 Submission Draft specification (Version 1.1). However, WebSphere Application Server does not provide a full security token service that implements all the contents of the WS-Trust draft specification.

Support for the approved version 1.3 specification, which is dated March 2007, is provided for WebSphere Application Server version 7.0 and later. The Security Context Token (SCT) provider supports the OASIS version 1.3 specifications for WS-Trust and WS-SecureConversation. There is a configuration option that allows support for the two different levels of the WS-Trust standard to co-exist on the same server. This provides interoperability between systems and products that support different specification levels. See the topic Configuring the security context token provider for the trust service using the administrative console for details.

A setting is also provided to specifically disable support for the WS-Trust 2005 Submission Draft specification (Version 1.1) for the Security Context Token provider. For more information about this property, refer to the topic Disabling the draft standard level for the Security Context Token.

Processing a trust service request depends on the specifications referenced in the request. Also, the trust service response is determined by the level of the specification used in the request.

For more information about WS-Trust:

- See the IBM developerWorks website.
- See the schema for the specification: http://docs.oasis-open.org/ws-sx/ws-trust/200512
- Refer to the `wst` namespace prefix that is used for WS-Trust in the Web Services Trust Language (WS-Trust) specification dated March 2007.

Configuring system policy sets using the administrative console:

By defining a custom policy set or defining assertions about how services are defined, you can configure Web Services Security. You can use the administrative console to manage custom policy sets.

Before you begin

A policy set specifies a set of common message policy assertions that can be specified within a policy. For example, a policy set can define general security policy assertions that apply to other protocols, such as Web Services Security (WS-Security), SOAP messages, Web Services Secure Conversation (WS-Secure Conversation) and Web Services Trust (WS-Trust).

There are two main types of policy sets; application policy sets and system policy sets. Application policy sets are used for business-related assertions. These assertions are related to the business operations that are defined in the Web Services Description Language (WSDL) file. System policy sets, on the other hand, are used for non-business-related system messages. These messages are defined in other specifications which apply qualities of service (QoS). Examples of QoS are the request security token (RST) messages that are defined in WS-Trust, the create sequence messages that are defined in WS-Reliable Messaging, and the metadata exchange messages defined by WS-MetadataExchange.

Important: Use system policy sets with the trust service, or Web Services MetadataExhange (WS-MEX). The requestor (client) must utilize Java API for XML-Based Web Services (JAX-WS) only. Requestors which use Java API for XML-based remote procedure calls (JAX-RPC) are incompatible with the policy set QOS.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

About this task

Only custom policy sets can be modified. Default system policy sets are read only and cannot be changed.

Procedure

1. To define system policy sets, click **Services** > **Policy sets** > **System policy sets**.
2. Click one of the following actions to work with the system policy set configurations:

   New To create a system policy set configuration. Enter a unique name for the system policy set configuration in the Name field. For example, you might specify `EcommerceTrustServiceSecurity`.

   Delete To delete an existing configuration. Select the check box next to an existing policy set name, and click **Delete**.

   Copy To copy an existing configuration. Select the check box next to an existing policy set name, and click **Copy**.

   Import
   To import an existing configuration. Select the check box next to an existing policy set name, and click **Import**. For more information, read about importing policy sets using the administrative console.

Export
To export an existing configuration. Select the check box next to an existing policy set name, and click **Export**. For more information, read about exporting policy sets using the administrative console.

3. To edit the settings of an existing policy set configuration, click the link for the existing custom system policy set that you want to change. Use the administrative console to modify existing custom policy sets that have been created.
4. Optional: If creating a policy set, enter a short description for the new policy set. Default policy sets can only be viewed. For a custom policy set, edit the brief description of the policy set in the Description field. This description displays in the list on the System policy sets panel. The description should be meaningful to you and other potential users of this policy set.
5. If creating a new policy set, click **Apply**. The policy set name must be applied before you can add policy types to the new policy set.
6. Optional: If needed, add the policy type information, or change the policy types for an existing system policy set. You can add, delete, enable, or disable policy types for the selected policy set. You can add any valid policy types to the policy set collection. The following are available policy types for system policy sets:
   - HTTP transport - for HTTP transport policies
   - SSL transport - for HTTPS transport policies
   - WS-Addressing - for endpoint addressing policies
   - WS-Security - for secure SOAP messages policies
7. Click **OK** and then click **Save** to save the information directly to the master configuration.

Results

You have provided the basic information to create a system policy set. You can also create a new or update an existing system policy set for the WebSphere Application Server trust service, or Web Services MetadataExhange (WS-MEX), using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

What to do next

After creating a system policy set and adding the policy types, attach the system policy set to a trust service operation for an endpoint, or attach it to one of the trust service default operations.

Defining a new system policy set using the administrative console:

Use policy sets, or assertions, to define system service operations, for your Web Services Security configuration. Whenever you create a new policy set, you must add policy types to the policy set. You can add HTTP Transport, WS-Addressing, WS-Security, and SSL Transport policy types to the system policy set collection.

Before you begin

A policy set specifies a set of common message policy assertions that can be specified within a policy. For example, a policy set can define general security policy assertions that apply to other protocols such as Web Services Security (WS-Security), SOAP messages, Web Services Trust (WS-Trust), and Web Services Secure Conversation (WS-SecureConversation).

Important: Use system policy sets with the trust service only. The requestor (client) must utilize Java API for XML-Based Web Services (JAX-WS) only. Requestors which use Java API for XML-based remote procedure calls (JAX-RPC) are incompatible with the policy set QOS.

About this task

Use the system policy sets to configure access to the WebSphere Application Server trust service. You can create and define a custom system policy set.

Procedure

1. Using the administrative console, click **Services > Policy sets > System policy sets** .
2. To create a system policy set and add a policy type, click **New**.
3. Enter a name for the policy set in the **Name** field. The name must be unique for the new system policy set. For example: EcommerceTrustServiceSecurity
4. Enter a brief description of the policy set in the **Description** field. This description displays in the System Policy Sets collection. The description should be descriptive enough for you and other potential users to identify the policy set.
5. Click **Apply** to apply the name and description information.
6. Click **Add** to add a trust policy by selecting one from the policies listed. The following policies are available to use for system policy sets:
   - HTTP transport - for HTTP transport policies
   - SSL transport - for HTTPS transport policies
   - WS-Addressing - for endpoint addressing policies
   - WS-Security - for secure SOAP messages policies
7. Click **Save** to save directly to the master configuration.

Results

You have provided the basic information to create or modify a policy set. You can also create a new or update an existing policy set for the WebSphere Application Server trust service using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

What to do next

After creating or modifying a system policy set and adding the policy types, attach the policy set to an endpoint operation or attach it to one of the trust service default operations.

System policy set collection:

Use this panel to create and manage policy sets. A policy set is a named collection of policies. System policy sets, or assertions about how services are defined, are used to configure access to the trust service.

There are two main types of policy sets; application policy sets and system policy sets. Application policy sets are used for business-related assertions. These assertions are related to the business operations that are defined in the Web Services Description Language (WSDL) file. System policy sets, on the other hand, are used for non-business-related system messages. These messages are defined in other specifications which apply qualities of service (QoS). Examples of QoS are the request security token (RST) messages that are defined in WS-Trust, the create sequence messages that are defined in WS-Reliable Messaging, and the metadata exchange messages defined by WS-MetadataExchange.

To view this administrative console page, click **Services** > **Policy sets** > **System policy sets**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Select:

Provides a check box next to the name of an existing system policy set that you want to select for further actions.

To manage existing system policy sets, select the check box for a system policy set and then select one of the following actions:

Actions	Description
Delete	Removes one or more selected system policy sets.
Copy	Opens a new panel where you can create a copy of the selected existing policy set. Provide a unique name and, optionally, a description for the copied policy set. You must also specify whether to transfer the attachment and binding from the original version to the copy. You can select only one policy set to be copied at one time.
Import	Imports a policy set. This is a menu item with the option of importing a policy set from a default repository or a selected location. You can select and import the default policy sets from the default repository. The default repository for the import function in the administrative console is the directory which contains the default policy sets. The administrative console also displays the default policy sets in a list which includes descriptions, to allow you to select the desired policy set that you want to import.
Export	Opens a new panel where you can export the selected policy set. You can select only one policy set to be exported at one time.

New:

Specifies to create and define a custom system policy set.

Name:

Provides a list of available system policy sets.

This column displays a list of default and custom system policy set names. WebSphere Application Server provides several default system policy sets:

- **TrustServiceSecurityDefault** is a default trust policy set. This trust policy set specifies the asymmetric algorithm as well as the public and private keys to provide message security. Message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using RSA. Message confidentiality is provided by encrypting the body and signature using RSA. This policy set follows the WS-Security specifications for the issue and renew trust operation requests.
- **TrustServiceSymmetricDefault** is a default trust policy set. This trust policy set specifies the symmetric algorithm as well as the derived key algorithms to provide message security. Message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using HMAC-SHA1. Message confidentiality is provided by encrypting the body and signature using AES. This policy set follows the WS-Security and WS-SecureConversation specifications for the validate and cancel trust operation requests.
- **SystemWSSecurityDefault** is a default system policy set that specifies the asymmetric algorithm and both the public and private keys to provide message security. Message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using RSA encryption. Message confidentiality is provided by encrypting the body and signature using RSA encryption.

All custom system policy sets are also displayed in the list. Click the system policy set name to view additional details about the selected policy set.

Information	Value
Data type:	String
Defaults:	TrustServiceSecurityDefault, TrustServiceSymmetricDefault or SystemWSSecurityDefault

Editable:

Provides information as to whether the system policy set can be edited.

This column shows whether the policy set is a user-defined, custom policy set that can be edited or whether the policy set is a default policy set that is not editable. Values displayed in this field are: `Editable` or `Not editable`. You can change the properties for a default, not editable policy set by copying it, and then modifying the properties of the copy. For more information, read about copying default policy set and bindings settings.

Important: Even though a policy set is identified as not editable, it is deletable. For example, you cannot edit information for the default system policy set, but you can delete the policy set.

Information	Value
Data type:	String
Default:	Not editable

Description:

Provides brief descriptions of the system policy sets that currently exist.

This column provides a brief description of the policy sets that are available. You cannot edit information for the default system policy sets. For custom policy sets that you create, you can create the description when you create the policy set. Or, you can edit any custom policy set and modify the description on the details panel at any time. The description field is optional.

System policy set settings:

Use this panel to create a new system policy set or to edit information about an existing custom system policy set. System policy sets, or assertions about how services are defined, are used to configure access to the trust service.

To view this administrative console page, complete one of the following procedures:

- **Services** > **Policy sets** > **System policy sets** > ***policyset_name***
- **Services** > **Policy sets** > **System policy sets** > **New**

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Important: You can edit the fields on this page only if the policy set is a custom trust policy set. You cannot edit default trust policy sets.

Name:

Specifies the name of the trust policy set.

This field displays the name of the existing custom policy set that you selected. If a new policy set is being created, this field is blank. Enter a policy set name.

Information	Value
Data type:	String

Description:

Specifies a brief description of the new or existing custom policy set.

This field provides a brief description for the existing policy set that is displayed in the Name field. If a new policy set is being created, this field is blank. Enter a brief policy set description to help distinguish it from other policy sets. You cannot change the descriptions of the default policy sets.

Information	Value
Data type:	String

Policies:

Specifies a collection of trust-related policies.

The Policies section displays a list of pre-configured system-level trust policies. If the system policy set is a default trust policy set, policies cannot be added, deleted, enabled, or disabled. If the system policy set is an existing custom trust policy set, you can change the policies. If you are creating a new custom trust policy set, you can add new policies. You can also delete, enable, or disable any policies that are added.

Select:

Specifies that you want to select an existing policy for further actions.

Click `Add` to display a list of valid policies that you can add to the named trust policy set.

To manage existing system policies, select the check box for a policy and select one of the following actions:

Actions	Description
Delete	Removes one or more policies from the named custom policy set.
Enable	Specifies that the policy is enabled for the policy set and displays `Enabled` in the State column.
Disable	Specifies that the policy is disabled for the policy set. The policy remains in the list but displays `Disabled` in the State column.

Policy:

Specifies the name of the policy.

This column displays one or more of these trust policies:
- **Custom properties** – for custom property policies
- **HTTP transport** – for HTTP transport policies
- **SSL transport** – for HTTPS transport policies
- **WS-Addressing** – for endpoint addressing policies
- **WS-Security** – for secure SOAP messages policies

State:

Specifies an enabled or disabled state for each policy.

This column displays whether the policy is enabled or disabled for each of the policies that are listed in the Policy column. For example, to change the state of the policy from enabled to disabled, select the check box for the policy, and click **Disable**.

Description:

Displays a brief description of the policy.

You can view these descriptions only.

Configuring attachments for the trust service using the administrative console:

You can attach the trust service operations for a service endpoint to a system policy set and binding. Each new endpoint that is specified initially has the following four operations: issue, renew, cancel, and validate. By default, all endpoints inherit the policy set and binding that are attached to the respective trust service operation under Trust Service Defaults. However, you can explicitly attach a different policy set.

Before you begin

First you must define your policy sets and bindings. *Policies* describe the protection or quality of service that is provided (such as message security, transport and so forth). *Bindings* specify some details about how to implement the policy, such as: the path for the keystore file, the class name of the token generator, or the JAAS configuration name.

Important: Use system policy sets with the trust service only. The requestor (client) must utilize Java API for XML-Based Web Services (JAX-WS) only. Requestors which use Java API for XML-based remote procedure calls (JAX-RPC) are incompatible with the policy set QOS.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

About this task

You can attach the trust service operations for a new endpoint to an existing policy set and binding. For each new service endpoint that is specified, four trust service operations (cancel, renew, validate and issue) change from having inherited attachments to being explicitly attached. The four operations are attached to the respective policy set and binding as specified in Trust Service Defaults. Then you can change the attachment to the desired existing policy set and binding.

An endpoint policy set consists of two sections: a bootstrap section and an application section. The system policy set attached to the Issue and renew trust service operations for a specific endpoint must correspond to the bootstrap section of the policy set for that endpoint. The system policy set attached to the Cancel and Validate trust service operations for a specific endpoint must correspond to the application section of the policy set for that endpoint.

This task describes how to manage trust service operations for service endpoint URLs that you want to attach to a system policy set and binding. To complete the configuration of the WebSphere Application Server trust service, you must also complete the following task:

- Create or manage targets. You can create explicit assignments for new service endpoints (targets) or manage endpoints that have a security token explicitly assigned or that inherit the Trust Service Default token.

The sample general bindings that are provided with the product are initially set as the global security (cell) default bindings. The default service provider binding and the default service client bindings are used when no application specific bindings or trust service bindings are assigned to a policy set attachment. For trust service attachments, the default bindings are used when no trust specific bindings are assigned. If you do not want to use the provided Provider sample as the default service provider binding, you can select an existing general provider binding or create a new general provider binding to meet your business needs. Likewise, if you do not want to use the provided Client sample as the default service client binding, you can select an existing general client binding or create a new general client binding. To specify your global security (cell) default bindings, use the administrative console and click **Services** > **Policy sets** > **Default policy set bindings**. For environments with multiple security domains, you can optionally choose the general provider and general client bindings that you want to use as the default bindings for a domain. For more information about default bindings see the topic Setting default policy set bindings.

Procedure

1. To manage system policy set attachments for trust service operations, click **Services** > **Trust service** > **Trust service attachments**. The list displays all endpoints that have at least one operation with a policy set attached as well as Trust Service Defaults. The list also displays the system policy set and the binding for each operation.
2. Select one or more of the following actions to configure the trust service attachments:

   New Attachment

   Opens a new panel where you can specify the service endpoint URL. For each new service endpoint that is specified, four trust service operations (cancel, renew, validate and issue) change from having inherited attachments to being explicitly attached. The four operations are attached to the respective policy set and binding as specified in Trust Service Defaults. These initial attachments can be changed.

   Attach

   Displays a list of existing system policy sets, including the default trust-related system policy sets, to which each of the four trust service operations for a service endpoint can be attached. First, select the operation (for example, Cancel token) and then click **Attach** to display the list of available system policy sets. Select a default or custom system policy set to attach. When you change the policy set attachment, the binding automatically changes to **Default**. Select the operation and click **Assign Binding** to change the binding.

   The pre-configured system policy sets that you can select include:

   - **TrustServiceSecurityDefault**

     This trust policy set specifies the asymmetric algorithm as well as the public and private keys to provide message security. Message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using RSA. Message confidentiality is provided by encrypting the body and signature using RSA. This policy set follows the WS-Security specification for the issue and renew trust operation requests.
   - **TrustServiceSymmetricDefault**

     This trust policy set specifies the symmetric algorithm as well as the derived key algorithms to provide message security. Message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using HMAC-SHA1. Message confidentiality is provided by encrypting the body and signature using AES. This policy set follows the WS-Security and WS-SecureConversation specifications for the validate and cancel trust operation requests.
   - **SystemWSSecurityDefault**

     This system policy set specifies the asymmetric algorithm and both the public and private keys to provide message security. Message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using RSA encryption. Message confidentiality is provided by encrypting the body and signature using RSA encryption.

Inherit Operation Defaults
: Sets the operation to inherit the respective trust service default trust service policy set attachment and binding. If you select the attachments to modify and then click **Inherit Operation Defaults**, the explicit attachment for both the policy set and the binding is removed. Thereafter, the operation inherits any change to the default trust service policy set and binding.

Assign Binding
: Changes the existing binding. You can create and assign a new binding, assign the Default binding, or assign an existing trust service specific binding to each of the selected trust service attachments.

Update Runtime
: Updates the trust service runtime with any configuration changes that are made to the trust service attachments, token providers, and targets.

3. Optional: Modify the custom policy set by clicking the name of a custom policy set from the list. Edit the settings for custom policy sets, as needed. Default trust service policy set information can only be viewed.

   You cannot edit the default policy sets: TrustServiceSecurityDefault and TrustServiceSymmetricDefault, or SystemWSSecurityDefault. TrustServiceSecurityDefault is the default for the issue and renew operations. TrustServiceSymmetricDefault is the default for the cancel and validate operations.

   At least one trust service operation for the endpoint service URL must be explicitly attached for the endpoint service URL to be displayed. If an operation is explicitly attached, the system policy set name appears. If no policy set is explicitly attached, the respective default trust service policy set appears, followed by the text `(inherited)`.
4. Optional: Modify the trust service specific binding by clicking the name of a binding from the list, as needed. Edit the settings for the trust service specific binding, as needed. Any modifications to a trust service binding affect all trust service attachments that reference the binding.

   If the resource has a policy set directly attached, either the bindings name appears or `Default` appears.
5. Save your changes before applying the changes to the trust service runtime configuration.
6. Click **Update Runtime** to update the trust service runtime configuration with any data changes for token providers, trust service attachments, and targets. Whether the confirmation window appears depends on whether you select the **Show confirmation for update runtime command** check box. Expand **Preferences** to view the check box.
7. Optional: Confirm or cancel if the confirmation window appears. If you deselected the **Show confirmation for update runtime command** check box, all changes are made immediately without displaying the confirmation window.

Results

You have provided the basic information to create or update a trust service attachment. You have configured trust service operation attachments to system policy sets and bindings.

What to do next

You can also create a new attachment for the WebSphere Application Server trust service using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

Creating a service endpoint attachment using the administrative console:

You can attach the trust service operations for a new service endpoint URL to system policy sets and bindings. The operations for each new endpoint are attached to the Trust Service Default policy sets and bindings. Each new endpoint initially has the following four operations: issue, renew, cancel, and validate.

Before you begin

First you must define your policy sets and their bindings. *Policy sets* describe the protection or quality of service that is provided (such as message security, transport and so forth). *Bindings* specify some details about how to implement the policy set, such as: the path for the keystore file, the class name of the token generator, or the JAAS configuration name.

Important: Only use system policy sets with the trust service. The requestor (client) must utilize only Java API for XML-Based Web Services (JAX-WS). Requestors that use Java API for XML-based remote procedure calls (JAX-RPC) are incompatible with the policy set QOS.

About this task

Attaching the trust service operations for a new endpoint to existing policy sets and bindings requires two steps. After initially attaching the endpoint, the following four operations are configured: issue, renew, cancel, and validate. These four operations explicitly attach to Trust Service Defaults. You can then modify these attachments to existing policy sets and bindings.

This task describes how to create or manage service endpoint URLs that you want to attach to the policy set and binding. To complete the configuration for the WebSphere Application Server trust service, you must also create or manage targets.

If no explicit bindings are attached, WebSphere Application Server uses the cell-level default binding, referred to as Default.

Procedure

1. To view existing trust service attachments, click **Services** > **Trust service** > **Trust service attachments**. Until you create the first attachment, only the default attachments for each operation are displayed.
2. To create an attachment, click **New Attachment**.
3. Enter the service endpoint URL in a valid format. Note that when the URL in the trust service attachment does not match the URL, including matching the case, to which the trust service request is sent, the policy set that is defined in the attachment is not applied. Instead, IBM WebSphere Application Server uses the policy set that is attached to the default for the trust operation.

   For example, where `demo` is the endpoint, you might enter: http://localhost:9080/wssamplebeta/demo
4. Click **Attach** to attach the URL and to return to the Trust service attachments panel. After you click **Attach**, the Trust service attachments panel displays the new service endpoint URL and the initial four operations. The service endpoint URL that you specified is listed in the Trust service attachments collection. These four token operations (cancel, renew, validate and issue) for the specified endpoint are initially attached to Trust Service Defaults.
5. On the Trust service attachments panel, change the policy set or binding attachment, as needed. You can return any operation to its initial state by inheriting Trust Service Defaults.

   Note: Changing the policy set forces the binding to change to Default.
6. Save your changes before applying the changes to the Web Services Security runtime configuration.
7. Click **Update Runtime** to update the Web Services Security runtime configuration with any data changes for token providers, trust service attachments, and targets. Whether the confirmation window appears depends on whether you selected the **Show confirmation for update runtime command** check box. Expand **Preferences** to view the check box.
8. Optional: Confirm or cancel if the confirmation window appears. If you deselected the **Show confirmation for update runtime command** check box, all changes are made immediately without displaying the confirmation window.

Results

You have provided the basic information to create a trust service attachment and to configure a policy set, a binding, and the operation information.

What to do next

You can also create a new attachment for the trust service using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

Next, configure the security context token provider or configure targets to complete the trust service configuration.

Trust service attachments collection:

Use this page to view information about or manage system policy set attachments and bindings. Endpoints with at least one operation directly attached to a policy set are displayed.

This page displays each endpoint that has at least one operation that is directly attached to a system policy set. The operations for other endpoints inherit the trust service default policy set and binding data. You can click **New Attachment** to create explicit attachments for endpoints not displayed, or click **Attach** to change the policy set for an operation. Changing the system policy set for an operation removes the binding data for that operation, and resets that data to the system default binding settings. You can also click **Assign Binding** to create a new binding configuration or change the existing binding configuration for the selected operation.

To view this administrative console page, click **Services > Trust service > Trust service attachments**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Show confirmation for update runtime command:

Specifies whether to enable or disable the display of the confirmation window before the Web Services Security runtime configuration is updated for supported tokens, targets, and trust service attachments.

Click **Preferences** to expand the information. You can select or clear the **Show confirmation for update runtime command** check box. If you do not select this check box, updates to the security runtime configuration are made without first displaying a confirmation window. If you select the check box, the confirmation window is displayed before updates to the security runtime configuration are made.

Information	Value
Data type:	Check box
Default:	Enabled (check box is selected)

Retain filter criteria:

Specifies whether to retain the filter criteria.

Click **Preferences** to expand the information. You can select or deselect the **Retain filter criteria** check box. This check box determines whether **Endpoint URL** is used as the filter criteria to reduce the displayed list of endpoints.

Information	Value
Data type:	String

Information	Value
Default:	All (check box is not selected)

Search terms:

Specifies the search criteria to use to reduce the displayed list of endpoints.

Click **Preferences** to expand the information. Type the search term you want to use in the **Search terms** field. Use the asterisk (*) as a wildcard character for all terms. You can also search for multiple unknown or partial characters within the term. For example, typing the search term `par*` returns `partly`, `participate`, `partial`, and all other terms beginning with the letters `par`.

Information	Value
Data type:	String
Default:	* (search for all)

Select:

Specifies that you want to select an existing resource, such as an endpoint or an operation, for further actions.

For existing endpoints, select the check box next to an operation, and then select one of the following actions:

Actions	Description
Attach	Displays a list of policy sets that are available to be attached to an endpoint operation (cancel, reset, validate, or issue) or to one of the trust service default operations. Highlight and click the policy set to attach the policy set to the selected operation. You cannot attach a policy set to an endpoint.
Inherit operation defaults	Detaches the currently attached policy set and binding for each selected operation and sets the operation to inherit the trust service default policy set and binding for each operation.
Assign binding	Lists the bindings that are available to select for the policy set to which you want to attach the binding. You can also create a new binding. • Select **Default** to create and assign the system default binding to the selected policy set attachment. When you select this binding the runtime uses the default binding for the server, cell or in the multiple security domain environment to which the service resource is deployed. • Select **New Trust Service Specific Binding** to create a binding that is specific to the policy set and shares the characteristics of the policy set. This type of binding is reusable only for trust service attachments. • Select an existing general binding to assign the binding to the selected policy set attachment. Multiple selection is valid only when all the resources have the same policy set attached.

New attachment:

Specifies that you want to create an explicit policy set attachment.

Click **New Attachment** to access a new panel where you can enter an endpoint URL to create attachments for each of the four endpoint operations of the provided URL. Initially, the attachment consists of the policy set and binding that are listed as the Trust Service Default for that operation.

Information	Value
Data type:	Button

Update runtime:

Updates the trust service configuration for any changed attachments, targets, and token information.

If the **Show confirmation for update runtime command** preference is enabled, then a panel is displayed where you can confirm that you want to update the trust service configuration. If the preference is disabled, the trust service configuration is updated immediately without any confirmation.

Information	Value
Data type:	Button

Service endpoint URL / Operation:

Displays a list of the trust service default operation attachments and every service endpoint URL that has at least one operation with a policy set attached.

Each endpoint has four operations: issue, cancel, renew, and validate. Each of the operations for all other endpoints inherits the trust service default policy set and binding.

When the URL in the trust service attachment does not match the URL to which the trust service request is sent, the policy set that is defined in the attachment is not applied. Instead, IBM WebSphere Application Server uses the policy set that is attached to the default for the trust operation.

Information	Value
Data type:	String
Default:	Trust Service Default

Policy set:

Displays the attached or inherited policy set for each operation of all endpoint URLs. Any endpoint URL that is not displayed inherits the trust service default policy set for each operation. Provides a list of default and custom system policy sets that are attached to the service endpoint URL.

The policy set names are displayed in this column for each operation. If the policy set is inherited from the trust service default, rather than being explicitly attached, **inherited** is displayed in parentheses following the policy set name. Because only operations can have a policy set attachment, the Policy Set column for each endpoint URL row displays **Not applicable**.

Click the system policy set name to view or edit the policy set details information. Note that you can view, but not edit, the default policy sets. Default policy sets cannot be changed.

Information	Value
Data type:	String
Defaults:	TrustServiceSecurityDefault, TrustServiceSymmetricDefault or SystemWSSecurityDefault

Binding:

Displays the binding that is assigned to each policy set attachment for each operation of the listed endpoint URLs. Any endpoint URL that is not displayed inherits the trust service default binding for each of the four operations.

The name of the assigned binding for each policy set attachment is displayed in this column for each operation. If the attachment is inherited from the trust service default, **inherited** is displayed in parentheses following the binding name. If you select **Assign Binding > Default**, the system default binding is applied to the policy set attachment, and the word **Default** is displayed in this column. If the system default binding is inherited, then **inherited** is displayed in parentheses following **Default**.

The system default binding is also assigned when you attach a new policy set to an operation. Because only operations can have policy set attachments, the binding column for each endpoint URL row displays **Not applicable**. Rows that are not directly related to a token and display the trust service default, display the text, **Not applicable**, for the binding. Additionally, rows that are not directly related to a token and display only the service endpoint URL display the text, **Not applicable**, for the binding.

Click the trust service specific binding name to view or edit the binding information. You can view, but not edit, the TrustServiceSecurityDefault, TrustServiceSymmetricDefault or SystemWSSecurityDefault bindings.

Information	Value
Data type:	String
Default:	TrustServiceSecurityDefault, TrustServiceSymmetricDefault or SystemWSSecurityDefault

Trust service attachments settings:

Use this page to create a new attachment to the current Trust Service Defaults policy set and binding for the four token operations: cancel, issue, renew, and validate.

To view this administrative console page, complete the following procedure:

- Click **Services** > **Trust service** > **Trust service attachments** > **New Attachment** .

Service endpoint URL:

Specifies the service endpoint URL that you want to attach to the policy set and binding for the trust service default operations.

Use this field to specify a service endpoint URL. The URL must be specified in a valid format, such as `http://www.mybusiness.com`.

Note that when the URL in the trust service attachment does not match the URL to which the trust service request is sent, the policy set that is defined in the attachment is not applied. Instead, IBM WebSphere Application Server uses the policy set that is attached to the default for the trust operation.

After you enter the URL and click **Attach**, the custom service endpoint URL is displayed in a list of explicitly attached service endpoint URLs on the Trust service attachments panel. In addition to the new service endpoint URL, the Trust service attachments panel displays a list of the corresponding four operations (cancel, issue, renew and validate).

On the Trust service attachments panel, you can change the Trust Service Default policy set and binding attachments for any of the four operations. These policy sets apply to any URL not displayed, and therefore not explicitly attached to a policy set and binding. Changing the policy set for a URL operation resets a custom binding setting to the default value.

On the Trust service attachments panel, if you want to remove the explicit policy set attachments and binding assignments, select each of the URL operations, and click **Inherit Operation Defaults**. If all four operations are changed to inherit the Trust Service Default policy set and binding, then the URL no longer displays on this panel.

Information	Value
Data type:	String (URL format)

Configuring the security context token provider for the trust service using the administrative console:

Configure the WebSphere Application Server trust service to issue a specific security token to the requestor for communication with an endpoint. Use the administrative console to configure the security context token provider that the trust service provides.

Before you begin

WebSphere Application Server provides a trust service. The trust service provides both a security token service and additional WebSphere Application Server trust-related functionality. To configure the trust service, in addition to managing the security context token provider, you must first complete the following tasks:

- Create or manage supported targets. You can create explicit assignments for new service endpoints (targets) or manage endpoints that have the security context token provider explicitly assigned or that inherit the token provider designated as the Trust Service default.
- Create or manage the attachment of token operations for service endpoints to policy sets and bindings.

The order in which you complete these tasks is not important.

About this task

This task describes how to manage the security context token provider and how to define or modify the properties of the security context token provider.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Procedure

1. To manage the security context token provider, click **Services > Trust service > Token providers**.
2. To edit the settings of the security context token provider configuration, click the link for the token provider name. You cannot edit the name, class name, or token type schema URI when modifying the token provider information.
   a. The format of the token type schema Uniform Resource Identifier (URI) is in the standard URI format. For example, for a version 1.3 security context token, the URI is: http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
   b. Change the amount of time, in minutes, in the **Time in cache after timeout** field that the expired token is kept in cache and where the token can still be renewed. The default value is 120 minutes. This value cannot be less than 10 minutes.
   c. Change the amount of time, in minutes, in the **Token timeout** field that the issued token is valid. The default value is 10 minutes. This value cannot be less than 10 minutes.

d. Select the **Allow renewal after timeout** check box to enable the renewal of a token after the token has expired. If selected, the amount of time, within which an expired token can still be renewed, is specified in minutes in the **Time in cache after expiration** field.
e. Select the **Allow postdated tokens** check box to enable postdated tokens. Use postdated tokens to specify whether a client can request a token to become valid at a later time.
f. Select the **Support Secure Conversation Token v200502** to enable use of the older draft submission specification level of the security context token. The correct URI for this level of the token type schema appears in the field under the check box: http://schemas.xmlsoap.org/ws/2005/02/sc/sct.
g. Click **New** to define a new custom property or click **Edit** to modify the custom property. Specify these settings using the Custom Properties setting. Custom properties are used to set internal system configuration properties. Custom properties are arbitrary name-value pairs of data, where the name might be a property key or a class implementation, and where the value might be a string or the value might be a true or false value.
h. If you define a custom property, type a name. Refer to the documentation for the token provider for valid custom property names.
i. If you define a custom property, type a value. Refer to the documentation for the token provider for the values for a property name.
j. Repeat defining the name and the value for each custom property that you add.
k. Click **OK**. You are returned to the Token providers panel.

3. Save your changes before applying the changes to the Web Services Security runtime configuration.
4. Click **Update Runtime** to update the Web Services Security runtime configuration with any data changes for token providers, trust service attachments, and targets. Whether the confirmation window is displayed depends on whether you select the **Show confirmation for update runtime command** check box. Expand **Preferences** to view the check box.
5. Optional: Confirm or click **Cancel** when the confirmation window appears. If you deselected the **Show confirmation for update runtime command** check box, all changes are made immediately without displaying the confirmation window.

Results

You have completed the required steps to modify the security context token provider configuration and to update the Web Services Security runtime configuration. You can also update the security context token provider configuration for the trust service using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

What to do next

Next, if you have not done so already, you must also configure targets or configure attachments to complete the trust service configuration.

Modifying the security context token provider configuration for the trust service using the administrative console:

WebSphere Application Server provides a pre-configured token, the Security Context Token (SCT). Use the administrative console to modify the configuration of the security context token provider.

Before you begin

WebSphere Application Server provides a trust service. The trust service provides both a security token service and additional WebSphere Application Server trust-related functionality. To configure the trust service, in addition to managing the security context token provider, you must first complete the following tasks:

- Create or manage supported targets. You can create explicit assignments for new service endpoints (targets) or manage endpoints that have a security token provider explicitly assigned or that inherit the token provider designated as the Trust Service default.
- Create or manage the attachment of token operations for service endpoints to policy sets and bindings.

The order in which you complete these tasks is not important.

About this task

This task describes how to configure the security context token provider and how to define the token provider properties.

Procedure

1. To configure the security context token provider, click **Services > Trust services > Token providers**.
2. To change the configuration of the security context token provider, click the link for the token provider name (Security Context Token). For an existing token, the token name, class name and URI are displayed, but are not editable.
3. Optional: Change the amount of time, in minutes, in the **Time in cache after expiration** field that the expired token is kept in cache and where the token can still be renewed. The default value is 120 minutes, and you cannot type a value that is less than 10 minutes.
4. Optional: Change the amount of time, in minutes, in the **Token timeout** field that the issued token is valid. The default value is 120 minutes, and you cannot type a value that is less than 10 minutes.
5. Optional: Select the **Allow renewal after timeout** check box to enable the renewal of a token, after the timeout time has expired. If selected, the amount of time, within which an expired token can still be renewed, is specified in the **Time in cache after expiration** field.
6. Optional: Select the **Allow postdated tokens** check box to enable postdated tokens. Use postdated tokens to specify whether a client can request a token to become valid at a later time.
7. Optional: Select the **Support Secure Conversation Token v200502** check box to enable use of the older draft submission specification level of the security context token. The correct URI for this level of the token type schema appears in the field under the check box: http://schemas.xmlsoap.org/ws/2005/02/sc/sct.
8. Click **New** if you want to define a new custom property. Specify additional configuration using the **Custom Properties** setting. Custom properties are used to set internal system configuration properties. Custom properties are arbitrary name-value pairs of data, where the name might be a property key or a class implementation, and where the value might be a string or Boolean value.
   a. If defining a new custom property, type a name. For example, for a custom property, type: com.ibm.wsspi.wssecurity.trust.keySize
   b. If defining a new custom property, type a value. For example, the following value: 128
   c. Repeat the name and value steps for each new custom property.
9. Click **OK**. You are returned to the Token provider panel.
10. Save your changes before applying the changes to the Web Services Security runtime configuration.
11. On the Token provider panel, click **Update Runtime** to update the Web Services Security runtime configuration with any data changes for token providers, trust service attachments, and targets. Whether the confirmation window is displayed depends on whether you select the **Show confirmation for update runtime command** check box. Expand **Preferences** to view the check box.
12. Optional: Confirm or click **Cancel** when the confirmation window appears. If you deselected the **Show confirmation for update runtime command** check box, all changes are made immediately without displaying the confirmation window.

Results

You have completed the required steps to modify the configuration of the security context token provider and to update the Web Services Security runtime configuration. You can also modify the configuration of the security context token provider for the trust service using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

What to do next

If you have not done so already, you must also configure targets or configure attachments to complete the trust service configuration.

Trust service token custom properties:

WebSphere Application Server trust service provides several custom properties by default to define the default security context token (SCT).

Custom properties are name-value pairs of data that are passed to the token provider during configuration.

The Property name column displays the name of the custom property. The name must match the name of a configuration property or setting that the provider understands and expects. The Property value column displays the configuration setting that is passed to the provider during configuration.

These custom properties are provided by default by WebSphere Application Server, for you to configure when using the **Services > Trust service > Token providers > Security Context Token** page.

algorithm:

The value is AES.

keySize:

The value is 128.

Provider:

The value is IBMJCE.

Disabling the submission draft level for the security context token provider:

Use the administrative console to configure the security context token provider that the trust service provides. Two levels of the token are supported on WebSphere Application Server: the token defined by the WS-Trust February 2005 Submission Draft specification, and the token defined by the OASIS WS-Trust Standard version 1.3. You can disable a setting so that the server will not accept a trust request that specifies the submission draft level of the token.

About this task

Disable the Security Context Token provider support for the submission draft specification using the administrative console.

Procedure

1. Log on to the administrative console and navigate to the Token providers panel by clicking **Services** > **Trust service** > **Token providers**.
2. Click on **Security Context Token**.
3. Click to clear the **Support Secure Conversation Token v200502** check box.

4. Click **Apply** to save the changed setting.

Results

For more information, see the topic Modifying the security context token provider configuration for the trust service using the administrative console.

Trust service token provider settings:

Use this page to modify information for an existing token provider.

To view this administrative console page, complete the following actions:
- **Services** > **Trust service** > **Token providers** > ***token_provider_name***

Name:

Specifies the name of the token provider.

This field displays the unique name of the token provider (for example, `Security Context Token`). You cannot change the name for any existing token provider.

Information	Value
Data type:	String

Class name:

Specifies the package and class name of the trust service's Security Context Token provider.

This field displays the configuration class name, including the package information (for example, `com.ibm.ws.wssecurity.trust.server.sts.ext.sct.SCTHandlerFactory`).

You cannot change the class name for any existing token provider.

Information	Value
Data type:	String

Token type schema URI:

Specifies the Uniform Resource Identifier (URI) for the token type schema.

This field displays the unique token type schema URI. Use a valid URI format, such as: `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct`.

You cannot change the schema URI for any existing token provider.

Information	Value
Data type:	String

Time in cache after expiration:

Specifies the number of minutes that a token remains in the token cache after the token expires.

This field displays the time, in minutes, that the expired token is kept cached and can still be renewed.

Information	Value
Data type:	Integer
Default:	120
Minimum:	10
Maximum:	2147483647

Token timeout:

Specifies the amount of time, in minutes, that the issued token is valid.

This field displays the maximum timeout, in minutes, for a token to be considered valid.

Information	Value
Data type:	Integer
Default:	120
Minimum:	10
Maximum:	2147483647

Allow renewal after timeout:

Specifies to enable or disable the renewal of a token.

This check box specifies whether to allow a client to renew an expired token. Note the **Time in cache after expiration** field specifies the amount of time within which an expired token can still be renewed.

Information	Value
Data type:	Check box
Default:	Do not allow (unchecked)

Allow postdated tokens:

Specifies to enable or disable the use of postdated tokens.

This check box specifies whether a client can request a token to become valid at some point in the future.

Information	Value
Data type:	Check box
Default:	Do not allow (unchecked)

Support Secure Conversation Token v200502: This check box specifies whether support for the WS-Trust and WS-Secure Conversation Feb 2005 Submission Draft OASIS specification is enabled. The default URI for the token type schema is provided in the non-editable field below the check box.

Information	Value
Data type:	Check box
Default:	Enabled (checked)

Custom Properties:

Specifies additional configuration settings that the token provider might require.

This table lists custom properties. Use custom properties to set internal system configuration properties.

The Secure Context Token default configuration settings are :

Property Name	Property Value
com.ibm.wsspi.wssecurity.trust.algorithm	AES
com.ibm.wsspi.wssecurity.trust.keySize	128
com.ibm.wsspi.wssecurity.trust.provider	IBMJCE

Select:

Specifies custom properties that you can add to, edit, or delete from the token provider.

Click **New** to add and define a new custom property.

For existing custom properties, first select the check box for the name of the custom property, and click one of the following actions:

Actions	Description
Edit	Specifies whether to modify existing custom properties. This action requires one or more custom properties to be selected.
Delete	Removes the selected existing property from the listing in the Name column. This action requires one or more custom properties to be selected.

Name:

Displays the names of the custom properties that have been defined for the token provider.

This column displays the name of the custom property (for example, `com.ibm.wsspi.wssecurity.trust.keySize`). Custom properties are name-value pairs of data that are passed to the token provider during configuration. The name that you specify must match the name of a configuration property or setting that the provider understands and expects.

Information	Value
Data type:	String

Value:

Specifies the value for the custom property.

This column displays the value for the custom property (for example, `true`). Custom properties are name-value pairs of data. The value, which is represented as a string, is a configuration setting that is passed to the provider during configuration.

Information	Value
Data type:	String or Boolean

Trust service token providers collection:

Use this page to view information about or manage token providers for the trust service.

To view this administrative console page, click **Services > Trust service > Token providers**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Show confirmation for update runtime command:

Specifies to enable or disable the display of the confirmation window before the trust service configuration is updated when you click **Update Runtime**.

Click **Preferences** and then select the **Show confirmation for update runtime command** check box. If you select this check box, the confirmation window is displayed before updates to the security trust service configuration are made. If you do not select this check box, clicking **Update Runtime** updates the security trust service configuration without first displaying a confirmation window.

Information	Value
Data type:	Check box
Default:	Enabled (checked)

Update Runtime:

Updates the trust service configuration for any changed attachments, targets, and token information.

If the **Show confirmation for update runtime command** preference is enabled, then a panel is displayed where you can confirm that you want to update the trust service configuration. If the preference is disabled, updates to the trust service configuration are applied immediately without any confirmation.

Information	Value
Data type:	Button

Token Provider Name:

Lists available token providers.

This column displays the names of the pre-configured token providers. The pre-configured token provider is the Security Context Token (SCT). Click a token provider name link to view additional details.

Information	Value
Data type:	String
Default:	Security Context Token

Token Type Schema URI:

Provides the Uniform Resource Identifier (URI) for the token type schema.

This column displays the URIs of all pre-configured token providers.

Information	Value
Data type:	String

Configuring trust service endpoint targets using the administrative console:

The Trust Service manages tokens on behalf of service endpoints. A token provider is either explicitly or implicitly associated with each service endpoint. A specific token can be explicitly assigned to be issued when access to an endpoint is requested. Otherwise, the Trust Service Default token is issued.

Before you begin

The Web Services Secure Conversation specification defines the protocol for a client to establish a secure session with a target service. The security token service that WebSphere Application Server provides, referred to as the trust service, issues only the Security Context Token (SCT). The security context token is used for Web Services Secure Conversation (WS-SecureConversation).

About this task

This task describes how to create new or manage existing assignments of tokens to be issued for endpoint targets. You can create explicit assignments for new service endpoints (targets) or manage existing token assignments.

To complete the configuration for the trust service, you must have performed the following tasks:

- Manage the security context token provider.
- Create or manage service endpoint URLs that you want to attach to the policy set and binding.

The order in which you complete these tasks is not important.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Procedure

1. To configure new and existing trust service endpoint targets, click **Services > Trust service > Targets**. A list of all service endpoints that have a security token provider explicitly defined is displayed. The token provider assigned to the Trust Service Default by default handles requests to issue tokens to access an endpoint.
2. Click one of the following actions to manage a new or existing endpoint target configuration:

   New Assignment

   Opens a new panel where you can specify a custom service endpoint URL and explicitly assign the token provider, which is specified as the Trust Service Default, to be issued for access to the endpoint.

   Change Token

   Changes an explicitly assigned token to be issued for the service endpoint to the security context token. Select an endpoint and then click **Change Token**. Select the Security Context Token.

   Also, removes the explicit assignment of a token to be issued; therefore, the token that is issued is inherited from the Trust Service Default. Select an endpoint and then click **Change Token**. Click **Inherit Default** to remove a token provider assignment for the selected endpoint and to return the issued token to be the token that is specified as the Trust Service Default. If the token that is issued is inherited, the endpoint is no longer displayed in the list because the token provider is no longer explicitly assigned to the endpoint.
3. Click the token name link for an existing endpoint target to modify the token provider configuration information. You can modify the token type schema URI, or change custom properties.
4. Save your changes before applying the changes to the Web Services Security runtime configuration.
5. Click **Update Runtime** to update the Web Services Security runtime configuration with any data changes for token providers, trust service attachments, and targets. Whether the confirmation window is displayed depends on whether you select the **Show confirmation for update runtime command** check box. Expand **Preferences** to view the check box.
6. Optional: Confirm or click **Cancel** when the confirmation window appears. If you deselected the **Show confirmation for update runtime command** check box, all changes are made immediately without displaying the confirmation window.

Results

When you complete these steps, the service endpoint URL displays in the Targets collection, unless you changed the token to inherit the default value. You can also configure the trust service to issue tokens for individual endpoint targets using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

What to do next

You have completed the required steps to create or manage existing trust service targets, to assign the security token provider to an endpoint target, and to update the Web Services Security runtime configuration. Next, if you have not competed these tasks already, configure the security context token provider or configure attachments to the policy set and binding to complete the trust service configuration.

Assigning a new target for the trust service using the administrative console:

You can associate a security token provider with a service endpoint using the administrative console. After entering the service endpoint URL, the token provider configured as the Trust Service Default is explicitly associated with the service endpoint.

Before you begin

The Web Services Secure Conversation specification defines the protocol for a client to establish a secure session with a target service. The security token service that WebSphere Application Server provides, referred to as the trust service, issues the Security Context Token (SCT). The security context token is required for Web Services Secure Conversation (WS-SecureConversation).

About this task

This task describes how to register a service endpoint (target) with the trust service. Registration of an service endpoint with the trust service initially associates the token provider configured as the Trust Service Default with that service endpoint.

To complete the configuration for the trust service, you must have completed the following tasks:
- Manage the Security Context Token.
- Create or manage service endpoint URLs that you want to attach to the policy set and binding.

The order in which you complete these tasks is not important.

Procedure
1. To configure a custom endpoint target, click **Services > Trust service > Targets > New Assignment**.
2. At the New assignment panel, enter the Universal Resource Locator (URL) for the service endpoint, and click **Assign**. You are returned to the Targets panel where the custom service endpoint URL is displayed in the list. Initially, the token that is explicitly assigned to the custom endpoint is the token that is assigned as the Trust Service Default.
3. At the Targets panel, select the check box for a service endpoint, click **Change Token**, and select one of the following:
   a. Security Context Token (SCT). A security context token is defined by the WS-SecureConversation specification.
   b. **Inherit Default** if you want the token that is issued to be the token assigned as the Trust Service Default. The endpoint is not displayed in the list when the assignment is inherited because the token is no longer explicitly assigned to the endpoint.
4. At the targets panel, click the token name link for an existing endpoint target to modify the token provider configuration information.

5. Save your changes before applying the changes to the Web Services Security runtime configuration.
6. Click **Update Runtime** to update the Web Services Security runtime configuration with any data changes for token providers, trust service attachments, and targets. Whether the confirmation window is displayed depends on whether you select the **Show confirmation for update runtime command** check box. Expand **Preferences** to view the check box.
7. Optional: Confirm or click **Cancel** when the confirmation window appears. If you deselected the **Show confirmation for update runtime command** check box, all changes are made immediately without displaying the confirmation window.

Results

When you complete these steps, service endpoints explicitly associated with a token provider are displayed in the Targets collection. Service endpoints that have been changed to inherit the token provider configured as the Trust Service Default are not displayed. You can also configure the security token service to issue a specific token for access to a target using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

What to do next

You have completed the required steps to create a service endpoint URL, to assign the token to be issued for access to the target, and to update the Web Services Security runtime configuration. Next, if you have not completed these tasks already, configure the Security Context Token provider or configure attachments to the policy set and binding to complete the trust service configuration.

Trust service targets collection:

Use this page to view a list of targets, which are application server service endpoints. You can manage tokens by specifying which token is to be issued when access to a specific endpoint is requested.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

To view this administrative console page, click **Services** > **Trust service** > **Targets**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Show confirmation for update runtime command:

Specifies to enable or disable the display of the confirmation window before the WebSphere Application Server trust service configuration is updated when you click **Update Runtime**.

Click **Preferences** and then select the **Show confirmation for update runtime command** check box. If you select this check box, the confirmation window is displayed before updates to the trust service configuration are made. If you do not select this check box, clicking Update Runtime updates the trust service configuration without first displaying a confirmation window.

Information	Value
Data type:	Check box
Default:	Enabled (checked)

Select:

Specifies a check box for the service endpoint Universal Resource Locator (URL) that you want to select for further actions.

For existing endpoints, select the checkbox for the service endpoint and select one of the following actions:

Actions	Description
Change Token	Changes the token that is issued when access to an endpoint is requested. Selecting **Inherit Default** in the Change Token menu causes the following actions to occur: • The security token assignment is removed for the endpoint. • The token assigned as the Trust Service Default is issued for access to the endpoint. • The endpoint is no longer displayed in the list of endpoints that have tokens explicitly assigned. Only endpoints that are explicitly assigned a security token are displayed in the list. Endpoints that inherit the default do not display in the list.

New Assignment:

Defines a new service endpoint.

Initially, each endpoint is explicitly assigned the Trust Service Default token. By default, the pre-configured Security Context Token (SCT) is assigned, but that can be changed.

Information	Value
Data type:	Button

Update Runtime:

Updates the trust service configuration for any changed attachments, targets, and token information.

If the **Show confirmation for update runtime command** preference is enabled, then a panel is displayed where you can confirm that you want to update the trust service configuration. If the preference is disabled, updates the trust service configuration immediately without any confirmation.

Information	Value
Data type:	Button

Service Endpoint URL:

Specifies the Universal Resource Locator (URL) of the service endpoint for the explicitly assigned token.

This column lists the default service endpoint, Trust Service Default, and any custom service endpoints that have a token that is explicitly assigned to the endpoint, such as: `http://localhost:9080/EcommerceSTS`.

Information	Value
Data type:	String
Default:	Trust Service Default

Token Name:

Displays the name of the token to be issued when access to the endpoint is requested.

To inherit the default token, select the check box for a custom service endpoint URL, click **Change Token** > **Inherit Default**.

You can change the token type that is explicitly assigned as the Trust Service Default, but the token type cannot be left unassigned. If the token is not explicitly assigned, then the endpoint inherits the token that is assigned as the Trust Service Default token.

Click a token name link to access detailed information about the token. You can modify the token information, except for the token name. It is recommended that you do not modify the class name or the token type schema URI for the default token type, Security Context Token.

Changes to token properties apply to all tokens of this type that are issued for any endpoint.

Information	Value
Data type:	String
Default:	Security Context Token

Token Type Schema URI:

Specifies the schema Uniform Resource Identifier (URI) for the token type.

This column displays the schema URI for the explicitly assigned token type (for example, Security Context Token) in a valid URI format. The token type schema URI is a property of the token name and describes the version of the specification that is implemented for the security token.

Information	Value
Data type:	String
Default value:	http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct

Trust service targets settings:

Use this page to specify a custom service endpoint Universal Resource Locator (URL) and to assign a custom token type to the endpoint URL.

To view this administrative console page, click **Services** > **Trust service** > **Targets** > **New Assignment**.

Service endpoint URL:

Specifies the URL for the service endpoint.

Use this field to specify a custom service endpoint URL. The URL must be specified in a valid format, such as `http://localhost:9080/EcommerceSTS`. After you enter the URL and click **Assign**, the endpoint URL is explicitly assigned to the security token that is assigned the Trust Service Default.

The service endpoint URL is added to the list that displays on the Targets panel. Only endpoints that are explicitly assigned a security token are displayed in the list. Endpoints that inherit the default do not display in the list.

By default, the Trust Service Default token is the Security Context Token (SCT).

After clicking **Assign** and returning to the Targets panel, if you want to remove the explicit token assignment (and thereby change the token to be issued back to the default value), select the custom endpoint URL, and click **Inherit Default**. Then the following actions occur:

- The security token assignment is removed for the endpoint.
- The token assigned as the Trust Service Default is issued for access to the endpoint.
- The endpoint is no longer displayed in the list of endpoints that have tokens explicitly assigned.

Information	Value
Data type:	String (URL format)

Updating the Web Services Security runtime configuration:

Update Web Services Security runtime configuration with any data changes that you make and save for token providers, trust service attachments, and targets.

Before you begin

Before you update the Web Services Security runtime configuration, make your required data changes for token providers, trust service attachments, and targets. Save your changes before applying the changes to the Web Services Security runtime configuration.

About this task

Whether the confirmation window appears depends on whether the **Show confirmation for update runtime command** check box is selected. Expand **Preferences** from the Token providers panel, the Trust service attachments panel, or the Targets panel to see the **Show confirmation for update runtime command** check box. Preferences are collapsed by default.

- If you select the check box, then the confirmation window is displayed before updates to the security runtime configuration are made. The check box is selected by default.
- If you do not select the check box, then updates to the security runtime configuration are made immediately, without first displaying the confirmation window. The confirmation window does not appear again until you re-select the checkbox located under Preferences.

  Or, instead of deselecting the check box under Preferences, you can select the **Do not show this message again** check box on the Update runtime confirmation panel.

Procedure

1. Perform one of the following tasks:
   - **Services > Trust service > Trust service attachments**
   - **Services > Trust service > Token providers**
   - **Services > Trust service > Targets**
2. Click **Update Runtime**. If you did not select the **Show confirmation for update runtime command** check box, all changes are made immediately without displaying the confirmation window.
3. Optional: Confirm or cancel when the confirmation window appears. If you selected the **Show confirmation for update runtime command** check box, all changes require confirmation and the confirmation window is displayed.

Results

After clicking **Update Runtime**, the configuration changes you made on the Token providers, the Trust service attachments, or the Targets panel are updated for the Web Services Security runtime configuration.

What to do next

You can also manage and administer the trust service using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

After configuring the trust service targets, attachments, and tokens, next configure the secure conversation client cache, if you have not already completed this step.

Web services update runtime settings:

Use this page to confirm that the web services trust service runtime should be updated with the most recent configuration changes.

To view this administrative console page, complete one of the following procedures:
- **Services > Trust service > Trust service attachments**
- **Services > Trust service > Supported tokens**
- **Services > Trust service > Targets**

Make changes to attachments, tokens, or targets, save the changes, and click .**Update Runtime**.

Do not show this message again:

Specifies to enable or disable the confirmation panel before the Web Services Security runtime configuration is updated.

Click **OK** to confirm that you want to make the configuration changes effective immediately for the web services trust service runtime.

Information	Value
Data type:	Check box
Default	Enabled (checked)

Configuring the Web Services Security distributed cache using the administrative console:

You can configure the Web Services Security runtime to use the security distributed cache to store security tokens.

About this task

Web Services Security functions such as secure conversation, trust, and nonce use the distributed cache to store security tokens when the distributed cache is enabled. If the distributed cache option is not selected, then a local cache is used to store the tokens. WebSphere Application Server Version supports distributed caching for the tokens in both cluster and non-cluster environments. In a cluster environment, you can configure the security cache to be distributed. If the cache is distributed, then all servers in the cluster share information about issued tokens.

Procedure

1. To configure the secure conversation client cache, click **Services** > **Security cache**.
2. Change the time in minutes in the **Time token is in cache after timeout** field. The default value is 120 minutes. The minimum allowable time is 10 minutes, meaning you cannot enter a value that is less than 10 minutes. This field specifies the number of minutes that the token is in cache after the token expiration time expires (cache persist period).
3. Change the time in minutes in the **Renewal interval before token timeout** field. The default value and minimum allowable time is 10 minutes. You cannot enter a value that is less than 10 minutes. This field specifies the time period before the token expires when the client attempts to renew the token.

This window of time is just before token expires where, if the token is accessed, then the client attempts to renew the token so that a downstream call can complete.

It is important that this setting be set to a length of time that is longer than the longest possible transaction. This value must include the time it takes to transport to and from the server, the time that is needed by the server to process the request, and the time that is cached by reliable messaging, if appropriate. Setting this value to a length of time that is too small might result in the token expiring in the middle of a transaction and might prevent the transaction from completing.

If the Security Context Token is renewed too often, it might cause Web Services Secure Conversation (WS-SecureConversation) to fail or even cause an out-of-memory error to occur. It is required that you set the renewal interval before the token expires value for the Secure conversation client cache to a value less than the token timeout value for the Security Context Token. It is also suggested that the token timeout value be at least two times the renewal interval before the token expires value.

4. Select the **Enable distributed caching** check box, if you want to share the tokens across the cluster. When the checkbox is selected to enable distributed caching, choose one of the following settings for updating the caches.
   - Synchronous update of cluster members: performs synchronous update of cache objects on cluster members (default).
   - Asynchronous update of cluster members: performs a non-synchronous update of the cache on cluster members. This setting allows interoperability with cluster members that use the older style of updating as implemented in versions of WebSphere Application Server prior to version 7.0.
   - Token recovery support: assigns a shared data source as the distributed cache.

   If token recovery support is selected as the update method, then you must select a cell level data source using the drop-down list. Token state data is saved in the database defined as the data source. If there are no available data sources in the list, click on Manage data sources to add one or more new data source objects. The data source object supplies an application with connections for accessing the database.
5. To create a new custom property, click **New**. For example, you might add the `cancelActionRST` custom property with a value of `http://schemas.xmlsoap.org/ws/2005/02/trust/RST/SCT/Cancel`.
6. To edit an existing custom property, select the check box for the name of the existing custom property, and then click **Edit**. For example, you might change the name or the value of the cancelActionRST custom property.
7. Click **Apply** to save and apply the changes.

Results

You have provided the basic information to configure the Web Services Security distributed cache. Use either the administrative console or the wsadmin tool to modify the security cache configuration.

What to do next

You can also add or delete custom properties for the trust service using the wsadmin tool. The wsadmin tool examples are written in the Jython scripting language.

Security cache settings:

Use this page to configure the Web Services Secure Conversation (WS-SecureConversation) security local and distributed cache settings using the administrative console.

To view this administrative console page, click **Services** > **Security cache**.

Time token is in cache after timeout:

Sets the time that the token remains in cache after the token times out.

This field specifies the number of minutes for the time the token is in cache after the token expiration time expires (cache persist period). For example, if you specify 30 minutes, the token is kept in cache for this time period after the token expiration time. The default value is 10 minutes, which is the minimum number of minutes that is allowed.

Information	Value
Data type:	Integer
Default:	10 (minutes)

Renewal interval before token timeout:

Sets the time period before expiration that the client attempts to renew the token.

This field specifies the period of time, in minutes, before expiration that the client attempts to renew the token. This setting must specify a period of time that is longer than the time for the longest transaction or else the token might expire during the transaction. This time must include time for transport to and from the server, processing by the server, and any time delay that is because of time used for reliable messaging, when applicable. The default value is 10 minutes, which is the minimum number of minutes that is allowed.

If the Security Context Token is renewed too often, it might cause Web Services Secure Conversation (WS-SecureConversation) to fail or even cause an out-of-memory error to occur. It is required that you set the renewal interval before the token expires value for the security cache to a value less than the token timeout value for the Security Context Token. It is also suggested that the token timeout value be at least two times the renewal interval before the token expires value.

Information	Value
Data type:	Integer
Default:	10 (minutes)

Enable distributed caching:

Specifies whether distributed caching is enabled or disabled. If distributed caching is enabled, select distributed cache settings.

Use this check box to specify whether to use distributed caching when the server is in a clustered environment and when the tokens are shared across the cluster.

Information	Value
Data type:	Check box
Default:	No distributed caching (unchecked)

When the checkbox is selected to enable distributed caching, choose one of the following settings for updating the caches.

Button	Resulting Action
Synchronous update of cluster members	Performs synchronous update of cache objects on cluster members (default).
Asynchronous update of cluster members	Performs a non-synchronous update of the cache on cluster members. This setting allows interoperability with cluster members that use the older style of updating as implemented in versions of IBM WebSphere Application Server prior to version 7.0.
Token recovery support	Assigns a shared data source as the distributed cache.

If token recovery support is selected as the update method, then you must select a cell level data source using the drop-down list. Token state data is saved in the database defined as the data source. If there are no available data sources in the list, click on **Manage data sources** to add one or more new data source objects. The data source object supplies an application with connections for accessing the database.

Custom Properties:

Specifies additional configuration settings that the secure conversation client might require.

This table lists custom properties. Use custom properties to set internal system configuration properties. This collection is empty until the first custom property is defined.

Information	Value
Data type:	String

Select:

Specifies that you want to select further actions.

Use this check box to select custom properties for further actions. To manage existing custom properties, select the check box beside the name, and then select one of the following actions:

Actions	Description
Edit	Select to modify an existing custom property. This action is not displayed until you have added at least one custom property.
Delete	Select to remove an existing custom property.

Information	Value
Data type:	Check box

New:

Specifies that you want to add and define a new custom property.

Click **New** to define a new custom property.

Information	Value
Data type:	Button

Name:

Lists available custom properties.

This column displays the names of the custom properties that you can use with the secure conversation client (for example, `exampleProperty`). Custom properties are name-value pairs of data, where the name is a string representation of a property that is expected by the secure conversation client.

Information	Value
Data type:	String

Value:

Lists the values of the custom properties.

This column displays the values of the custom properties (for example, `true`). Custom properties are name-value pairs of data, where the value is a string representation of the property setting.

Information	Value
Data type:	String

Configuring the Kerberos token for Web Services Security:

Use this topic to configure the Kerberos token for message-level Web Services Security.

Before you begin

Before you can use Kerberos with Web Service Security, you must configure Kerberos in the IBM WebSphere Application Server. You do not need to enable Kerberos as the authentication mechanism. However, the Kerberos configuration file, `krb5.conf` or `krb5.ini`, and the Kerberos keytab file, `krb5.keytab`, are required.

The initial setup and configuration processes to use Kerberos with Web Services Security are identical to the configuration processes for using Kerberos with the security function. Therefore, you must set up and configure Kerberos before continuing with the steps in this topic.

The "Kerberos (KRB5) authentication mechanism support for security" topic provides an overview of the Kerberos functionality and provides the initial steps for setting up and configuring Kerberos for authentication purposes. Within this topic, you must complete the steps in the section "Setting up Kerberos as the authentication mechanism for WebSphere Application Server". Use that topic to configure Kerberos, the service principal, and the keytab files. In addition, that topic provides references to the process for configuring Kerberos as the authentication mechanism using the administrative console or commands. You can also find information on how to setup up Kerberos when the Key Distribution Center (KDC) and the Application Server do not use the same user registry.

About this task

The Kerberos token for JAX-WS applications is configured using policy sets and bindings. The JAX-WS application is attached with a custom policy and the Kerberos token is configured as a message protection token or an authentication token.

The implemented Kerberos functionality for Web Services Security also leverages existing tools and frameworks for the Kerberos token profile configuration for authentication and message protection. The support for Kerberos with Web Services Security in the product is based on the OASIS Web Services Security Kerberos Token Profile 1.1 specification.

To configure Kerberos with Web Service Security, complete the following steps:

Procedure

1. Enable the Kerberos token profile for JAX-WS applications.

   The JAX-WS application is attached with a custom policy that has a Kerberos token, which is configured with a message protection token or an authentication token. For more information, see "Configuring the Kerberos token policy set for JAX-WS applications" on page 779.
2. Select the customized Kerberos token type. You can define key bindings for request message protection and response message protection. You can use the key type, such as the key identifier or security token reference, for the outbound key information. If you use a derived key, use a security token reference in both the outbound and inbound key information. If you use a Kerberos session key, you can use a security token reference in the outbound key information and a key identifier in the

inbound key information for the client bindings. Then, use a key identifier in the outbound key information and a security token reference in the inbound key information for the provider bindings.
3. Select the customized Kerberos token types for the token generator or token consumer.
4. Configure the bindings for Kerberos message protection for JAX-WS applications. For more information, see the “Configuring the bindings for message protection for Kerberos” on page 781.

What to do next

Using this task, you have configured the Kerberos token for WebSphere Application Server.

Configuring the Kerberos token policy set for JAX-WS applications:

Use this topic to enable the Kerberos token policy set for JAX-WS applications.

Before you begin

Prior to beginning this task, you must specify the Kerberos configuration information for IBM WebSphere Application Server. For more information, see Kerberos (KRB5) authentication mechanism support for security.

The configuration model for the Kerberos token enables you to choose from the following existing WebSphere Application Server frameworks:
- For JAX-RPC applications, the deployment descriptor and bindings are used in the configuration. JAX-RPC application includes the deployment descriptor for a Kerberos custom token, which is configured with authentication tokens.
- For JAX-WS applications, the configuration uses a policy set and bindings. The JAX-WS application is attached by a custom policy with the Kerberos token configured with authentication tokens, message protection tokens, or both.

Note: Fix packs that include updates to the Software Development Kit (SDK) might overwrite unrestricted policy files. Back up unrestricted policy files before you apply a fix pack and reapply these files after the fix pack is applied.

About this task

Complete the following steps to configure the Kerberos token policy set for JAX-WS applications using the administrative console for WebSphere Application Server. In these steps, the Main policy configuation panel references the administrative console panel that is available after you complete the first five steps.

Procedure
1. Expand **Services** > **Policy sets** and click **Application policy sets** > **New** to create a new policy set.
2. Specify a name and a short description for the new policy set and click **Apply**.
3. From the Policies heading, click **Add** and then select the **WS-Security** security policy type.
4. Click **OK** and click **Save** to save the new configuration directly to the master configuration.
5. In the **Policies** field, click **WS-Security** and click **Main policy** on the WS-Security panel to configure the main policy for the Kerberos token policy set.
6. From the Key Symmetry heading, select **Use symmetric tokens** for message protection.
7. Click **Symmetric signature and encryption policies** to configure the Kerberos custom token type or clear the **Message level protection** check box if you are configuring an authentication token only.

Important: You do not need to configure the request token policy if you are using the Kerberos token for message protection. If you are configuring the authentication token only, proceed to the next step. If you are not configuring the request token policy for the authentication token, skip the next step.

8. On the Main policy configuration panel, configure the policy for the request token if you are configuring the authentication token.
   a. From the Policy Details heading, click **Request token policies**.
   b. Click **Add token type** and select **Custom**.
   c. Specify the name of the custom token in the **Custom token name** field.
   d. Specify the local part value in the **Local part** field. For interoperability with other web services technologies, specify the following local part: `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ`. If you are not concerned with interoperability issues, you can specify one of the following local name values:
      - `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ`
      - `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510`
      - `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510`
      - `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120`
      - `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120`

      These alternative values depend on the specification level for the Kerberos AP-REQ token that is generated by the Key Distribution Center (KDC). For more information about when to use these values, see Token type settings.
   e. Do not specify a value for the **Namespace URI** field if you are generating a Kerberos token.
   f. Click **OK** and **Save** to save the configuration directly to the master configuration.

   This step completes the configuration process for configuring the request token policy for the authentication token. You do not need to complete the next two steps. Complete the next steps to configure encryption and symmetric signature policies.
9. Return to the main policy configuration panel for the application policy set and click **Symmetric signature and encryption policies** to configure the encryption and symmetric signature policies.
   a. From the Message Integrity heading, click the **Action** menu list beside the **Token type for signing and validating messages** field and select **Custom**.
   b. From the Message Confidentiality heading, select the **Use same token for confidentiality that is used for integrity** option.
   c. Click **OK** and **Save** to save the configuration changes.
   d. From the Message Integrity heading, click the **Action** menu list beside the **Token type for signing and validating messages** field and select **Edit Selected Type Policy**.
   e. Edit the custom token type for the signature and encryption by specifying the local part for the Kerberos custom token.

      For example, specify `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ` for the local part value. Do not specify a Namespace URI value.
   f. Click **OK** and then click the **Save** link to save the configuration changes.
10. Return to the main policy configuration panel for the application policy set and click **Algorithms for symmetric tokens** to configure the symmetric token algorithm.
    a. Select the algorithm suite to use for the symmetric tokens from the **Algorithm suite** menu list. Select the Advanced Encryption Standard (AES) algorithms for a Kerberos token that is compliant with RFC-4120.

The symmetric key wrap, or private key cryptography, algorithms include:

- Triple DES key wrap: http://www.w3.org/2001/04/xmlenc#kw-tripledes
- AES key wrap (aes128): http://www.w3.org/2001/04/xmlenc#kw-aes128
- AES key wrap (aes256): http://www.w3.org/2001/04/xmlenc#kw-aes256

Restriction: To use the 256–bit AES encryption algorithm, you must apply the unlimited jurisdiction policy files. To remain in compliance, see Basic Security Profile compliance tips.

Before downloading these policy files, mount the product HFS as read/write. Back up the existing policy files prior to overwriting them, in case you want to restore the original files later. The existing policy files, which are the `local_policy.jar` and `US_export_policy.jar` files, are located in the *WAS_HOME*`/java/lib/security/` directory.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

For application server platforms using IBM Developer Kit, Java Technology Edition Version 5, you can obtain unlimited jurisdiction policy files by completing the following steps:

1) Visit the IBM developerWorks: Security Information website.
2) Click **Java 5**.
3) Click **IBM SDK Policy files**.

   The Unrestricted JCE Policy files for SDK 5 website is displayed.
4) Enter your user ID and password or register with IBM to download the policy files. The policy files are downloaded onto your workstation.
5) Re-mount your product HFS as read/only.

For more information on the algorithm suite components, see Algorithms settings.

b. Select either the **Exclusive cannonicalization** or **Inclusive cannonicalization** value for the **Cannonicalization algorithm** menu list. For more information, see XML digital signature.
c. Specify the **XPath 1.0** or **XPathfilter 2.0** version to use from the **XPath version** menu list.

What to do next

Configure the bindings for message protection for Kerberos for JAX-WS applications. For more information, see "Configuring the bindings for message protection for Kerberos."

Configuring the bindings for message protection for Kerberos:

To set up bindings for message protection with JAX-WS applications, you must create a custom binding. Complete this task to set the bindings for a Kerberos token as defined in the OASIS Web Services Security Specification for Kerberos Token Profile Version 1.1.

Before you begin

You must configure Kerberos for IBM WebSphere Application Server. For more information, see Kerberos (KRB5) authentication mechanism support for security. In addition, you must configure the Kerberos token policy set for JAX-WS applications. For more information, see Configuring the Kerberos token policy set for JAX-WS applications.

About this task

You can leverage existing frameworks including the policy set and bindings for JAX-WS applications.

You can configure a symmetric protection token or an authentication token. Both symmetric protection token and authentication token configurations use similar configuration data. However, you do not need to configure the authentication token if you intend to use a Kerberos symmetric protection token. For whichever token type you use, configure the token generator and the token consumer as indicated in the following list:

- Symmetric protection token
  - Token generator
  - Token consumer
- Authentication token
  - Token generator
  - Token consumer

Use the administrative console to configure the application-specific bindings to use a Kerberos token in web services message protection.

Procedure

1. Expand **Applications** > **Application Types**.
2. Click **WebSphere enterprise applications** > ***application name*** .
3. From the Web Services Properties heading, click **Service provider policy sets and bindings** to configure the service bindings or click **Service client policy sets and bindings** to configure the client bindings.
4. Select the resource to attach to the Kerberos token policy set and select **Attach Policy Set** > ***policy set name***. To configure the Kerberos token policy set, see "Configuring the Kerberos token policy set for JAX-WS applications" on page 779.
5. Click **Assign bindings** and select the application-specific binding or select **New Application Specific Binding** to create a new binding. To create a new binding, complete the following actions.
   a. Enter a name for the new binding in the **Binding configuration name** field and optionally enter a description for the binding in the **Description** field.
   b. Click **Add** and select **WS-Security** to specify a new policy set.
   c. Click **Authentication and protection** > **New**.
   d. Optional: Define a symmetric protection token for the token generator.

      Important: If you configure a symmetric protection token for the token generator, you must define a complimentary symmetric protection token for the token consumer.
      1) From the Protection tokens heading, click **New** and select **Token Generator**.
      2) Specify the name of the protection token in the **Name** field.
      3) Select **Custom** from the values in the **Token type** menu list.
      4) Specify the local name value in the **Local name** field.

         For interoperability with other web services technologies, specify the following local name: `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ`. If you are not concerned with interoperability issues, you can specify one of the following local name values:
         - `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ`
         - `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510`

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

These alternative values depend on the specification level for the Kerberos token that is generated by the Key Distribution Center (KDC). For more information about when to use these values, see Protection token settings (generator or consumer).

5) Do not specify a value for the **Namespace URI** field.
6) Select the **wss.generate.KRB5BST** value from the JAAS login menu list.

   If you have previously defined your own Java Authentication and Authorization Service (JAAS) login module, you can select your login module to handle the Kerberos custom token. To define a custom JAAS login module, click **New Application Login** > **New**, specify an alias for the new module, and click **Apply**. For more information, see Login module settings for Java Authentication and Authorization Service.

   Attention: Although the information in the “Login module settings for Java Authentication and Authorization Service” topic refers to security and not Web Services Security, the configuration for a login module for Web Services Security is identical to security.

7) Specify the token generator custom properties for the target service name, host, and realm.

   The combination of the target service name and host values forms the Service Principal Name (SPN), which represents the target Kerberos service principal name. The Kerberos client requests the initial Kerberos AP_REQ token for the SPN. Specify the following custom properties.

Table 154. Target service custom properties. Use these properties to specify the token generator information.

Name	Value	Type
com.ibm.wsspi.wssecurity.krbtoken.targetServiceName	Specify the name of the target service.	Required
com.ibm.wsspi.wssecurity.krbtoken.targetServiceHost	Specify the host name that is associated with the target service in the following format: myhost.mycompany.com	Required
com.ibm.wsspi.wssecurity.krbtoken.targetServiceRealm	Specify the name of the realm that is associated with the target service.	(Optional: If the targetServiceRealm property is not specified, the default realm name from the Kerberos configuration file is used as the realm name.)

   To use Kerberos token security in a cross or trusted realm environment, you must provide a value for the targetServiceRealm property.

   To specify multiple custom property name and value pairs, click **New**.

8) Click **Apply**.
9) From the Additional bindings heading, click **Callback handler**.
10) From the Class Name heading, select the **Use custom** option and specify `com.ibm.websphere.wssecurity.callbackhandler.KRBTokenGenerateCallbackHandler` in the associated field.
11) From the Basic Authentication heading, specify the appropriate values for the **User name**, **Password**, and **Confirm password** fields.

    The user name specifies the default user ID that is passed to the constructor of the callback handler; for example, `kerberosuser`.

12) Specify the token generator custom properties for Kerberos client principal name and password to initiate the Kerberos login.

These custom properties control the prompt and establish the token based on the credential cache. Specify the following custom properties.

Table 155. Kerberos login custom properties. Use this property to specify the token generator information.

Name	Value	Type
com.ibm.wsspi.wssecurity.krbtoken.loginPrompt	Enables the Kerberos login when the value is True. The default value is False.	Optional

To specify multiple custom property name and value pairs, click **New**.

13) Click **Apply** and **OK**.

When you return to the Authentication and protection panel in the next step, a new protection token is defined for the token generator. To edit the configuration for this new token, click its name on the panel.

e. Optional: Return to the Authentication and protection panel to define a symmetric protection token for the token consumer. To return to the Authentication and protection panel, click the **Authentication and protection** link after the messages section of the panel.

Important: If you configure a symmetric protection token for the token consumer, ensure that you have previously defined a complimentary symmetric protection token for the token generator.

1) From the Protection tokens heading, click **New** and select **Token Consumer**.
2) Specify the name of the protection token in the **Name** field.
3) Select **Custom** from the values in the **Token type** menu list.
4) Specify the local name value in the **Local name** field.

For interoperability with other web services technologies, specify the following local name: `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ`. If you are not concerned with interoperability issues, you can specify one of the following local name values:

- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120`

These alternative values depend on the specification level for the Kerberos token that is generated by the Key Distribution Center (KDC). For more information about when to use these values, see Protection token settings (generator or consumer).

5) Do not specify a value for the **Namespace URI** field.
6) Select the **wss.consume.KRB5BST** value from the JAAS login drop-down menu.

If you have previously defined your own Java Authentication and Authorization Service (JAAS) login module, you can select this login module to handle the Kerberos custom token. To define a custom JAAS login module, click **New Application Login** > **New**, specify an alias for the new module, and click **Apply**. For more information, see Login module settings for Java Authentication and Authorization Service.

Attention: Although the information in the Login module settings for Java Authentication and Authorization Service topic refers to security and not Web Services Security, the configuration for a login module for Web Services Security is identical to security.

7) Click **Apply**.
8) From the Additional bindings heading, click **Callback handler**.
9) From the Class Name heading, select the **Use custom** option and specify `com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler` in the associated field.
10) Click **Apply** and **OK**.

When you return to the Authentication and protection panel in the next step, you will see a new protection token defined for the token consumer. To edit the configuration for this new token, click its name on the panel.

f. Optional: Return to the Authentication and protection panel to define an authentication token configuration for the token generator. To return to the Authentication and protection panel, click the **Authentication and protection** link after the messages section of the panel.

Authentication tokens are sent in messages to prove or assert an identity.

Important: If you configure an authentication token for the token generator, you must define a complimentary authentication token for the token consumer.

1) From the Authentication tokens heading, click **New** and select **Token Generator**.
2) Specify the name of the authentication token in the **Name** field.
3) Select **Custom** from the values in the **Token type** menu list.
4) Specify the local name value in the **Local name** field.

For interoperability with other web services technologies, specify the following local name: `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ`. If you are not concerned with interoperability issues, you can specify one of the following local name values:

- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120`

These alternative values depend on the specification level for the Kerberos token that is generated by the Key Distribution Center (KDC). For more information about when to use these values, see Authentication generator or consumer token settings.

5) Do not specify a value for the **Namespace URI** field.
6) Select the **wss.generate.KRB5BST** value from the JAAS login menu list.

If you have previously defined your own Java Authentication and Authorization Service (JAAS) login module, you can select this login module to handle the Kerberos custom token. To define a custom JAAS login module, click **New Application Login** > **New**, specify an alias for the new module, and click **Apply**. For more information, see Login module settings for Java Authentication and Authorization Service.

Attention: Although the information in the Login module settings for Java Authentication and Authorization Service topic refers to security and not Web Services Security, the configuration for a login module for Web Services Security is identical to security.

7) Specify the token generator custom properties for the target service name, host, and realm.

The combination of the target service name and host values forms the Service Principal Name (SPN), which represents the target Kerberos service principal name. The Kerberos client requests the initial Kerberos AP_REQ token for the SPN. Specify the following custom properties.

Table 156. Target service custom properties. Use these custom properties to specify the token generator information.

Name	Value	Type
com.ibm.wsspi.wssecurity.krbtoken.targetServiceName	Specify the name of the target service.	Required
com.ibm.wsspi.wssecurity.krbtoken.targetServiceHost	Specify the host name that is associated with the target service in the following format: myhost.mycompany.com	Required
com.ibm.wsspi.wssecurity.krbtoken.targetServiceRealm	Specify the name of the realm that is associated with the target service.	Optional

To specify multiple custom property name and value pairs, click **New**.

8) Click **Apply**.
9) From the Additional bindings heading, click **Callback handler**.
10) From the Class Name heading, select the **Use custom** option and specify `com.ibm.websphere.wssecurity.callbackhandler.KRBTokenGenerateCallbackHandler` in the associated field.
11) From the Basic Authentication heading, specify the appropriate values for the **User name**, **Password**, and **Confirm password** fields.

    The user name specifies the default user ID that is passed to the constructor of the callback handler. For example: `kerberosuser`
12) Specify the token generator custom properties for Kerberos client principal name and password to initiate the Kerberos login.

    These custom properties control the prompt and establish the token based on the credential cache. Specify the following custom properties name and value pairs.

Table 157. Kerberos login custom properties. Use the custom properties to specify the token generator information.

Name	Value	Type
com.ibm.wsspi.wssecurity.krbtoken.loginPrompt	Enables the Kerberos login when the value is True. The default value is False.	Optional
com.ibm.wsspi.wssecurity.krbtoken.clientRealm	Specify the name of the Kerberos realm associated with the client	(Optional: The clientRealm property is optional for a single Kerberos realm environment.)

When implementing Web Services Security in a cross or trusted Kerberos realm environment, you must provide a value for the clientRealm property.

If an application generates or consumes a Kerberos V5 AP_REQ token for each web services request message, set the `com.ibm.wsspi.wssecurity.kerberos.attach.apreq` custom property to **true** in the token generator and the token consumer bindings for the application

To specify multiple custom property name and value pairs, click **New**.

13) Click **Apply** and **OK**.

When you return to the Authentication and protection panel in the next step, you will see a new authentication token is defined for the token generator. To edit the configuration for this new token, click its name on the panel.

g. Optional: Return to the Authentication and protection panel to define an authentication token configuration for the token consumer. To return to the Authentication and protection panel, click the **Authentication and protection** link after the messages section of the panel.

Important: If you configure an authentication token for the token consumer, ensure that you have previously defined an authentication token for the token generator.

1) From the Authentication tokens heading, click **New** and select **Token Consumer**.

2) Specify the name of the authentication token in the **Name** field.
3) Select **Custom** from the values in the **Token type** menu list.
4) Specify the local name value in the **Local name** field.

For interoperability with other web services technologies, specify the following local name: `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ`. If you are not concerned with interoperability issues, you can specify one of the following local name values:

- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120`

These alternative values depend on the specification level for the Kerberos token that is generated by the Key Distribution Center (KDC). For more information conditions under which to use these values, see the related link for the “Authentication generator or consumer token settings” topic.

5) Do not specify a value for the **Namespace URI** field.
6) Select the **wss.consume.KRB5BST** value from the JAAS login drop-down menu.

If you have previously defined your own Java Authentication and Authorization Service (JAAS) login module, you can select this login module to handle the Kerberos custom token. To define a custom JAAS login module, click **New Application Login** > **New**, specify an alias for the new module, and click **Apply**. For more information, see Login module settings for Java Authentication and Authorization Service.

Attention: Although the information in the Login module settings for Java Authentication and Authorization Service topic refers to security and not Web Services Security, the configuration for a login module for Web Services Security is identical to security.

7) Click **Apply**.
8) From the Additional bindings heading, click **Callback handler**.
9) From the Class Name heading, select the **Use custom** option and specify `com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler` in the associated field.
10) Click **Apply** and **OK**.

When you return to the Authentication and protection panel in the next step, you will see a new authentication token is defined for the token consumer. To edit the configuration for this new token, click its name on the panel.

What to do next

You can optionally define key bindings for the request message protection and response message protection. If you choose to derive a key from the Kerberos token, configure the derived key information when you configure the key information for signature and encryption.

Return to the steps in the Configuring the Kerberos token for Web Services Security topic to ensure you have completed the steps for configuring the Kerberos token.

Updating the system JAAS login with the Kerberos login module:

Update the Kerberos system JAAS login module for JAX-WS applications.

About this task

If the Kerberos authentication mechanism is configured in the WebSphere Application Server security configuration for JAX-WS applications, the JAAS login wss.caller must be updated with the system JAAS login module for Kerberos. The login module is specified as com.ibm.ws.security.auth.kerberos.WSKrb5LoginModule.

There are two methods to update the Kerberos system JAAS login module: using the administrative console, or by running a Jython script.

Procedure

1. Using the administrative console, follow these steps:
   a. Click **Security > Global security > Java Authentication and Authorization Service > System logins**.
   b. Click on **wss.caller**, then click **New** to create a new JAAS login module.
   c. In the **Module class name** field, type `com.ibm.ws.security.auth.kerberos.WSKrb5LoginModule`.
   d. Click **OK**.
   e. In the wss.caller panel, click **Set Order**, then click on **WSKrb5LoginModule**.
   f. Move WSKrb5LoginModule up in the list of modules so that it is after com.ibm.ws.wssecurity.impl.auth.module.WSWSSLoginModule but before com.ibm.ws.security.server.lm.ltpaLoginModule. The order of the modules in the list is important. The finished list of modules should look like this:

```
com.ibm.ws.wssecurity.impl.auth.module.PreCallerLoginModule                 1
com.ibm.ws.wssecurity.impl.auth.module.UNTCallerLoginModule                 2
com.ibm.ws.wssecurity.impl.auth.module.X509CallerLoginModule                3
com.ibm.ws.wssecurity.impl.auth.module.LTPACallerLoginModule                4
com.ibm.ws.wssecurity.impl.auth.module.LTPAPropagationCallerLoginModule     5
com.ibm.ws.wssecurity.impl.auth.module.KRBCallerLoginModule                 6
com.ibm.ws.wssecurity.impl.auth.module.WSWSSLoginModule                     7
com.ibm.ws.security.auth.kerberos.WSKrb5LoginModule                        8
com.ibm.ws.security.server.lm.ltpaLoginModule                               9
com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule                10
```

   g. Click **OK**, then click **Save** to save the changes.
   h. Restart the server.
2. You can also run a Jython script to update the module. For each cell, run the script **addKrbLoginModuleWSSCaller.py**, located in the *app_server_root*\bin directory, to update the WSKrb5LoginModule login module in the security configuration.
   a. Run the following command, where *app_server_root* is C:\WebSphere\AppServer:

```
wsadmin -conntype NONE -lang jython -f  C:\WebSphere\AppServer\bin\addKrbLoginModuleWSSCaller.py
```

   b. If the script is successful, the following message is displayed:

```
System JAAS login entry wss.caller has been updated.
```

   c. Restart the server.

Configuring Kerberos policy sets and V2 general sample bindings:

Configure the Kerberos policy sets and V2 general sample bindings that are included with WebSphere Application Server Version 7.0.0.1 and later.

Before you begin

In order to use the additional Kerberos policy sets and V2 general sample bindings that are included with the product, you must create a new profile after installing the product. Existing profiles are not automatically updated, and do not contain the Kerberos policy sets and V2 general sample bindings. You can update existing profiles manually using the following steps.

About this task

To update existing profiles, perform the following manual steps. The deployment manager profile and the stand-alone application server profile are the only profiles that you need to update.

Procedure

1. Copy the directories containing the additional Kerberos policy sets from the profile templates directory, *app_server_root*/profileTemplates/default/documents/config/templates/PolicySets, to the profile configuration directory, *profile_root*/config/templates/PolicySets. Each additional Kerberos policy set is contained in a separate directory. The directories are:
   - Kerberos V5 SecureConversation
   - Kerberos V5 WSSecurity default
   - TrustServiceKerberosDefault
2. Unpackage and copy the general bindings, Client sample V2 and Provider sample V2.
   a. Extract the directories and files from the package file *app_server_root*/profileTemplates/default/configArchives/AppSrv.car into a temporary directory.
   b. Copy the general binding directories from the temporary directory `<temp_dir>/cells/defaultCell/bindings/`, to the profile configuration directory for the cell, *profile_root*/config/cells/*<cellName>*/bindings. Each general binding is contained in a separate directory. The directories are:
      - Provider sample V2
      - Client sample V2
3. Restart the server.

Securing messages using SAML:

Configure policy sets, bindings, and SAML-specific tokens to secure web services and messages.

About this task

To secure messages using SAML, you can import the SAML default policy sets and modify them to enable SAML function. Because WebSphere Application Server with SAML does not support attaching a policy set directly to a Web services client, you must specify the policy sets and bindings used to enable SAML as custom properties in the web services client binding document.

You can also create a SAML bearer token using the SAML library API. A bearer token contains a bearer assertion, which is used to facilitate web browser single sign-on (SSO). Other SAML set up tasks described in this section include configuring policy sets and bindings for a bearer token, or a holder-of-key token, or to communicate with a Security Token Service (STS).

See the following topics for more information about securing messages using SAML.

Signing SAML tokens at the message level:

Secure SAML tokens at the message level by enabling assertion signing.

Before you begin

Before configuring signing for SAML tokens, you must configure SAML policy sets and bindings to create SAML tokens as authentication supporting tokens, with message level integrity protection. For more information, read about securing messages using SAML. In addition, the attached SAML bindings must be application-specific bindings, not general bindings. The transform algorithm used for signing SAML assertions is different from other signed parts, while only one transform algorithm is used with general bindings.

About this task

This task specifically addresses steps for how to digitally sign a SAML token. This task does not address any of the SAML Token Profile OASIS standard requirements for SAML sender-vouches or SAML bearer tokens with regards to message parts that must be signed. To sign SAML assertions, a SOAP message must include a <wsse:SecurityTokenReference> element in the <wsse:Security> header block. The SecurityTokenReference (STR) is referenced by the message signature using a <ds:Reference> element. The security token reference must include a <wsse:KeyIdentifier> element with the ValueType value, http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLID, or http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0#SAMLAssertionID, specifying the referenced assertion identifier. The <ds:Reference> element must include the URI of the STR-transform algorithm, http://docs.oasis-open.org/wss/2004/01/oasis-200401-wsssoap-message-security-1.0#STR-Transform. Use of STR-transform ensures that the SAML assertion itself is signed, not only the <wsse:SecurityTokenReference> element.

Follow these configuration steps to enable signing SAML tokens at the message level.

Procedure

1. Configure the message parts.
   a. From the administrative console, edit the SAML policy set, then click **WS-Security > Main policy > Request message part protection**.
   b. Under **Integrity protection**, click **Add**.
   c. Enter a part name for **Name of part to be signed**; for example, `saml_part`.
   d. Under **Elements in Part**, click **Add**.
   e. Select **XPath Expression**.
   f. Add two XPath expressions.

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
and local-name()='Envelope']/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
and local-name()='Security']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
and local-name()='SecurityTokenReference']
```

```
/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
and local-name()='Envelope']/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
and local-name()='Security']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
and local-name()='SecurityTokenReference']
```

   g. Click **Apply** and **Save**.
   h. If an application has never been started using this policy, no further action is required. Otherwise, either restart the application server or follow the instructions in the *Refreshing policy set configurations using wsadmin scripting* article, for the application server to reload the policy set.
2. Modify the client bindings to sign the SAML token.
   a. From the Service client policy set and bindings panel, click **WS-Security > Authentication and protection**.
   b. Modify the currently configured outbound Signed message part bindings to include the new SAML part that you created.

      Under **Request message signature and encryption protection**, select the part reference whose status is set to `Configured`. This part reference will most likely be **request:app_signparts**.
      1) From the **Available** list under Message part reference, select the name of the part to be signed, as created in step 1; for example, `saml_part`.
      2) Click **Add**, and then click **Apply**.
      3) In the **Assigned** list under Message part reference, highlight the name of the part you added; for example, `saml_part`.
      4) Click **Edit**.
      5) For the **Transform algorithms** setting, click **New**.

6) Select **http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform**.
7) Click **OK**, click **OK**, and then click **OK** one more time.

c. Update the SAML token consumer with the custom property to indicate digital signature with Security Token Reference

Under Authentication tokens, select and edit the SAML token you want to sign.

1) Under Custom property, click **New**.
2) Enter `com.ibm.ws.wssecurity.createSTR` as the custom property name.
3) Enter `true` as the value of the custom property.
4) Click **Apply**, and then click **Save**.

d. Restart the application.

3. Modify the provider bindings to accept a signed SAML token.

a. From the Service provider policy sets and bindings panel, click **WS-Security > Authentication and protection**.

b. Modify the currently configured inbound Signed message part bindings to include the new SAML part that you created.

Under **Request message signature and encryption protection**, select the part reference whose status is set to `Configured`. This part reference will most likely be **request:app_signparts**.

1) From the **Available** list under Message part reference, select the name of the part to be signed, as created in step 1; for example, `saml_part`.
2) Click **Add**, and then click **Apply**.
3) In the **Assigned** list under Message part reference, highlight the name of the part you added; for example, `saml_part`.
4) Click **Edit**.
5) For the **Transform algorithms** setting, click **New**.
6) Select **http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform**.
7) Click **OK**, click **OK**, and then click **OK** one more time.
8) Click **Save**.

c. Restart the application.

Configuring policy sets and bindings to communicate with STS:

Configure policy sets and binding documents to enable a web services client to request SAML assertions from an external Security Token Service (STS).

Before you begin

After installing, you must create a new server profile, or add SAML configuration settings to an existing profile. Read about setting up the SAML configuration for more information.

About this task

WebSphere Application Server with SAML supports web services clients using the Web Services Security policy set and bindings when communicating with an external security token service (STS). Web services clients use policy set and bindings to communicate with the target web services provider. A web services client uses two sets of policy set attachments: one set of policy set attachments for communicating to the target web services provider; and the other set of policy set attachments for communicating to the STS. Policy sets and bindings that are used when communicating with the target web services provider are attached to the web services client. In contrast, policy sets and bindings that enable STS communication are not directly attached to the web services clients. Instead, policy sets and bindings that enable STS

communication are specified as custom properties in the web services client binding document. You can use general bindings or application-specific bindings to communicate with an STS. Using a general binding to access an STS is straightforward; simply specify the general binding name in the custom properties.

The procedure to configure application-specific bindings to access an STS is more involved. The administrative console is designed to manage policy set attachments to communicate with a web service provider. The console is not designed to manage a second set of policy set attachments to communicate to an STS. However, you can use the administrative console to manage a policy set attachment to access an STS, as described in the procedure.

Use the administrative console to attach the policy set that is used to access an STS to a web services client, and then create and modify an application-specific binding. Once the binding configuration is complete, detach the policy set and binding from the web services client. This procedure is necessary because the next step is to attach the policy set and bindings to communicate to the target web services provider. Detached application-specific bindings are not deleted from the file system, so the web services client bindings custom properties can successfully refer to the detached application-specific bindings.

The procedure uses a default application policy set, Username WSHTTPS default, as an example to describe the configuration steps to access the STS. The steps can also be applied to other policy sets. The web services application, JaxWSServicesSamples, is used in the example. JaxWSServicesSamples is not installed by default.

Procedure

1. Import the Username WSHTTPS default policy set. In this example, the Username WSHTTPS default policy is used to demonstrate the procedure, but you can use a different policy set to configure the bindings, if the policy set meets the policy requirements of the external STS.
   a. Click **Services > Policy sets > Application policy sets**.
   b. Click **Import**.
   c. Select **From Default Repository**.
   d. Select the **WSHTTPS default** policy set.
   e. Click **OK** to import the policy set.
2. Attach a policy set for the trust client. Click **Applications > Application types > WebSphere enterprise applications >** ***JaxWSServicesSamples*** **> Service client policy sets and bindings**. The steps which pertain to attaching and detaching the policy set, and configuring the trust client binding, are required only if an application-specific binding is used to access the external STS. You can skip these steps, and go to the step that discusses configuring communication with the STS, if you use a general binding to access the external STS.
   a. Select the check box for the web services client resource.
   b. Click **Attach Client Policy Set**.
   c. Select the policy set, **Username WSHTTPS default**.

   This step attaches the policy set to the web services trust client, as you would do to use this policy set for the application client to access the target web services. However, since you plan to use the Username WSHTTPS default policy set to access an external STS instead, the policy set is only temporarily attached to the Web services client. The purpose of this step is to allow you to use the administrative console to create or to modify the client binding document.
3. Configure the trust client binding.
   a. Select the web services client resource again.
   b. In the Service client policy sets and bindings panel, click **Assign Binding**.
   c. Click **New Application Specific Binding** to create an application-specific binding.
   d. Specify a binding configuration name for the new application-specific binding. In this example, the binding name is **SamlTCSample**.

   e. Add the **SSL transport** policy type to the binding. Optionally, you can modify the NodeDefaultSSLSettings settings. Click **Security > SSL certificate and key management > SSL configurations > NodeDefaultSSLSettings**.
4. Optional: You can create an HTTP transport binding using the previous steps if you want to configure a user name and password to add to the HTTP header, or if you want to configure a proxy. If you elect not to create an HTTP transport binding, the web services runtime environment uses the default HTTP transport settings.
5. Add the **WS-Security** policy type to the binding, then modify the authentication settings.
   a. Click **Applications > Application types > WebSphere enterprise applications >** ***JaxWSServicesSamples*** **> Service client policy sets and bindings >** ***SamlTCSample*** **> Add > WS-Security > Authentication and protection > request:uname_token**.
   b. Click **Apply**.
   c. Select **Callback handler**.
   d. Specify a user name and password (and confirm the password) to authenticate the web services client to the external STS.
   e. Click **OK** and **Save**.
6. After the binding settings are saved, return to the Service client policy sets and bindings panel to detach the policy set and bindings.
   a. Click **Applications > Application types > WebSphere enterprise applications >** ***JaxWSServicesSamples*** **> Service client policy sets and bindings**.
   b. Click the check box for the web services client resource.
   c. Click **Detach client policy set**.

   The application-specific binding configuration you created in the previous steps is not deleted from the file system when the policy set is detached. This means that you can still use the application-specific binding you created to access the STS.
7. Import the SSL certificate from the external STS.
   a. Click **Security > SSL certificate and key management > Manage endpoint security configurations >** ***server_or_node_endpoint*** **> Keystores and certificates > NodeDefaultTrustStore > Signer certificates**.
   b. Click **Retrieve from port**.
   c. Specify the host name and port number of the external STS server, and assign an alias to the certificate. Use the SSL STS port.
   d. Click **Retrieve signer information**.
   e. Click **Apply** and **Save** to copy the retrieved certificate to the NodeDefaultTrustStore object.
8. Optional: If further modifications to the wstrustClientBinding configuration are needed, and the wstrustClientBinding property is pointing to an application-specific binding, you must attach the application-specific binding to the web services client before you can complete the modifications. The attachment is temporary. As detailed in the previous steps, you can detach the modified application-specific binding from the web service client after the modification is completed.

Results

After successfully completing the steps, the web services client is ready to send requests to the external STS. To enable this function, the following conditions and settings were activated when you completed the procedure:

- Attach a policy set and bindings that propagate SAML tokens. For example, attach the SAML11 Bearer WSHTTPS default policy set and the Saml Bearer Client sample general binding to the web service client.
- The Username WSHTTPS default policy set and an application-specific binding, SamlTCSample, are referenced in the Saml Bearer Client sample binding, and are set up to access the external STS.

- The external STS SSL certificate has been retrieved and added to the NodeDefaultTrustStore truststore.
- To confirm that you successfully enabled the function, you can configure the trace setting, com.ibm.ws.wssecurity.*=all=enabled. The trace shows that SAML assertions are issued by the external STS, for example:


```
[8/23/09 18:26:59:252 CDT] 0000001f TrustSecurity 3   Security Token Service reponse:
[8/23/09 18:26:59:392 CDT] 0000001f TrustSecurity 3
<?xml version="1.0" encoding="UTF-8"?><s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
   xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <s:Header>
        <a:Action s:mustUnderstand="1">http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal</a:Action>
        <a:RelatesTo>urn:uuid:663A7B27BA8EB2CF9D1251070029934</a:RelatesTo>
        <o:Security xmlns:o="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" s:mustUnderstand="1">
            <u:Timestamp u:Id="_0">
                <u:Created>2009-08-23T23:26:57.664Z</u:Created>
                <u:Expires>2009-08-23T23:31:57.664Z</u:Expires>
            </u:Timestamp>
        </o:Security>
    </s:Header>
    <s:Body>
      <trust:RequestSecurityTokenResponseCollection xmlns:trust="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
        <trust:RequestSecurityTokenResponse>
          <trust:Lifetime>
            <wsu:Created
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">2009-08-23T23:26:57.648Z</wsu:Created>
            <wsu:Expires
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">2009-08-24T09:26:57.648Z</wsu:Expires>
          </trust:Lifetime>
        <wsp:AppliesTo xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
          <a:EndpointReference>
             <a:Address>https://taishan.austin.ibm.com:9443/WSSampleSei/EchoService12</a:Address>
              </a:EndpointReference>
        </wsp:AppliesTo>
        <trust:RequestedSecurityToken>
          <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion" MajorVersion="1" MinorVersion="1"
AssertionID="_3c656382-9916-4e5f-9a16-fe0287dfc409" Issuer="http://svt193.svt193domain.com/Trust" IssueInstant="2009-08-23T23:26:57.663Z">
          <saml:Conditions NotBefore="2009-08-23T23:26:57.648Z" NotOnOrAfter="2009-08-24T09:26:57.648Z">
          <saml:AudienceRestrictionCondition>
          <saml:Audience>https://taishan.austin.ibm.com:9443/WSSampleSei/EchoService12</saml:Audience>
          </saml:AudienceRestrictionCondition>
          </saml:Conditions>
          <saml:AuthenticationStatement AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
AuthenticationInstant="2009-08-23T23:26:57.640Z">
          <saml:Subject>
          <saml:SubjectConfirmation>
          <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:bearer</saml:ConfirmationMethod>
          </saml:SubjectConfirmation>
          </saml:Subject>
          </saml:AuthenticationStatement>
            <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <ds:SignedInfo>
            <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
            <ds:Reference URI="#_3c656382-9916-4e5f-9a16-fe0287dfc409">
            <ds:Transforms>
            <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
            <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>YGySZX4VPv25R+oyzFpE0/T/tjs=</ds:DigestValue>
            </ds:Reference>
            </ds:SignedInfo>
                 <ds:SignatureValue>eP68...Vr08=</ds:SignatureValue>
                   <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
                      <X509Data>
                        <X509Certificate>MII...ymqg3</X509Certificate>
                      </X509Data>
                    </KeyInfo>
                  </ds:Signature>
             </saml:Assertion>
         </trust:RequestedSecurityToken>
       <trust:RequestedAttachedReference>
          <o:SecurityTokenReference xmlns:o="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <o:KeyIdentifier
ValueType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0#SAMLAssertionID">_3c656382-9916-4e5f-9a16-fe0287dfc409</o:KeyIdentifier>
            </o:SecurityTokenReference>
        </trust:RequestedAttachedReference>
        <trust:RequestedUnattachedReference>
          <o:SecurityTokenReference xmlns:o="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
             <o:KeyIdentifier
ValueType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.0#SAMLAssertionID">_3c656382-9916-4e5f-9a16-fe0287dfc409</o:KeyIdentifier>
          </o:SecurityTokenReference>
        </trust:RequestedUnattachedReference>
        <trust:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1</trust:TokenType>
        <trust:RequestType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue</trust:RequestType>
        <trust:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer</trust:KeyType>
```

```
        </trust:RequestSecurityTokenResponse>
        </trust:RequestSecurityTokenResponseCollection>
    </s:Body>
</s:Envelope>
```

What to do next

Complete the web service client and web service provider configuration. Read about configuring client and provider bindings for the SAML bearer token for more information.

Configuring client and provider bindings for the SAML bearer token:

A SAML bearer token is a SAML token that uses the Bearer subject confirmation method. In a bearer subject confirmation method, a sender of SOAP messages is not required to establish correspondence that binds a SAML token with contents of the containing SOAP message. You can configure the client and provider policy set attachments and bindings for the SAML bearer token.

Before you begin

WebSphere Application Server with SAML provides numerous default SAML token application policy sets and several general client and provider binding samples. Before you can configure the client and provider bindings for the SAML bearer token, you must:

- Create one or more new server profiles that include SAML configuration settings, or add SAML configuration settings to an existing profile. Read about setting up the SAML configuration for more information about how to add SAML configuration settings to a profile.
- Import one of the following default policy sets:
  - SAML20 Bearer WSHTTPS default
  - SAML20 Bearer WSSecurity default
  - SAML11 Bearer WSHTTPS default
  - SAML11 Bearer WSSecurity default

  The SAML11 policy sets are almost identical to the SAML20 policy sets, except that the SAML20 policy sets support the SAML Version 2.0 token type, while the SAML11 policy sets support the Version 1.1 token type.

  Two default policy sets are required. Therefore, you must import the Username WSHTTPS default policy set, and one of the following bearer policy sets. Make sure that the bearer policy set you import corresponds to your scenario; for example SAML 1.1 or 2.0 and HTTPS or non-HTTPS.
  - SAML11 Bearer WSHTTPS default for SAML 1.1 tokens using HTTPS
  - SAML20 Bearer WSHTTPS default for SAML 1.1 tokens using HTTPS
  - SAML20 Bearer WSSecurity default for SAML 2.0 tokens using HTTP
  - SAML11 Bearer WSSecurity default for SAML 2.0 tokens using HTTP

  To import these policy sets, in the administrative console:
  1. Click **Services > Policy sets > Application policy sets**.
  2. Click **Import**.
  3. Select **From Default Repository**.
  4. Select the two desired default policy sets.

     The SAML default policy set chosen in this step will be referred to as your appropriate SAML policy in the following procedure steps.
  5. Click **OK** to import the policy sets, and then click **Save** to save your changes.
- If the SAML assertions will be signed by the STS and you will require trust evaluation of the issuer (the signer), a keystore file that can be used for trust evaluation of the issuer's X.509 certificate must be available. This keystore can either contain the issuer's public certificate or all the information required to build the certificate's path. Supported keys store types include: jks, jceks, and pkcs12. This file will be referred to as the trust store file in the following procedure.

- The Username WSHTTPS default policy will be used to communicate with the STS. If the STS issuer certificate for use with SSL has not been imported into NodeDefaultSSLSettings, see the topic *Retrieving signers from a remote SSL port* for a description of how import this certificate. You also might want to review the topic *Secure installation for client signer retrieval in SSL* for more general information about STS issuer certificates.

About this task

A SAML token policy is defined by a CustomToken extension in the application server. To create the CustomToken extension, you must define the SAML token configuration parameters in terms of custom properties in the client and provider binding document. The Saml Bearer Client sample and the Saml Bearer Provider sample for general bindings contain the essential configuration for the custom properties.

The client and provider sample bindings contain both SAML11 and SAML20 token type configuration information. These samples can be used with both SAML11 and SAML20 policy sets. Depending on how you plan to implement the SAML tokens, you must modify the property values in the installed binding samples. Examples of the properties and property values are provided in the following procedure.

As the following procedure indicates, to modifying the binding sample, you must first configure the web services client policy set attachment, and then modify the web services provider policy set attachment. The example provided in the procedure uses the sample web services application JaxWSServicesSamples.

Procedure

1. Configure the trust client

   If you will be using general bindings to access the external STS, skip to Attach the policy set and bindings to the client application step.

   If you will be using application specific bindings to access the external STS, complete the following substeps.

   a. Temporarily Attach a policy set for the trust client to the web services client application so that bindings can be configured.

      Attaching a policy set for the trust client allows you to use the administrative console to create, and then modify the client binding document bindings. You only have to complete this action if an application-specific binding is used to access the external STS.

      1) In the administrative console, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings**.
      2) Select the web services client resource (JaxWSServicesSamples).
      3) Click **Attach Client Policy Set**.
      4) Select the policy set **Username WSHTTPS default**.

   b. Create the trust client binding.

      1) Select the web services client resource again (JaxWSServicesSamples).
      2) Click **Assign Binding**.
      3) Click **New Application Specific Binding** to create an application-specific binding.
      4) Specify a binding configuration name for the new application-specific binding. In this example, the binding name is SamlTCSample.

   c. Add the SSL transport policy type to the binding,

      Click **Add > SSL transport** and then click **OK**.

   d. Add the WS-Security policy type to the binding, then modify the authentication settings for the trust client.

1) If the he WS-Security policy type is not already in the SamlTCSample binding definition, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings > SamlTCSample**.
2) Click **Add > WS-Security > Authentication and protection > request:uname_token**.
3) Click **Apply**.
4) Select **Callback handler**
5) Specify a user name and password for the web services client to authenticate to the external STS.
6) Click **OK**, and then click **Save**.

e. After the binding settings are saved, return to the **Service client policy sets and bindings** panel, and detach the policy set and bindings.

1) Click either **Service client policy sets and bindings** in the navigation for this page, or **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings**.
2) Select the web services client resource (JaxWSServicesSamples), and then click **Detach client policy set**.

The application-specific binding configuration you just created is not deleted from the file system when the policy set is detached. Therefore, you can still use the application-specific binding you created to access the STS with the trust client.

2. Attach the SAML policy set and bindings to the client application

a. Attach the desired SAML policy set to the web services client application.

1) If the SAML policy is not already on the Service client policy sets and bindings page for JaxWSServicesSamples, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings**
2) Select the web services client resource.
3) Click **Attach Client Policy Set**.
4) Select your appropriate SAML policy for the web services client.

b. Attach the Saml Bearer Client sample general binding to the client.

1) Select the web services client resource again.
2) Click **Assign Binding**.
3) Select **Saml Bearer Client sample**.

3. Configure the web services client bindings.

Configure the STS endpoint URL in the sample binding.

a. Click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings > Saml Bearer Client sample > WS-Security > Authentication and protection**.

b. Click either **gen_saml11token** or **gen_saml20token** in the Authentication tokens table.

c. Click **Callback handler**.

d. Modify the stsURI property to specify the STS endpoint.

This property is not required for the self-issuer in an intermediate server. However if the property is specified for the self-issuer in an intermediate server, it is set to `www.websphere.ibm.com/SAML/Issuer/Self`.

e. Verify that the following properties are set to the required values.

If any of these properties are set to some other value, you must change the property setting to the required value.

- The confirmationMethod property must be set to `Bearer`.
- The keyType property must be set to `http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer`.

- The wstrustClientPolicy property must be set to `Username WSHTTPS default`.
- The value specified for the wstrustClientBinding property must match the name of application specific binding of your trust client created in the previous steps. For example, in the previous steps, we created an application specific binding named SamlTCSample. In this scenario `SamlTCSample` must be specified as the value for the wstrustClientBinding property

f. Optional: If you want to change how the application server searches for the binding, you can specify the wstrustClientBindingScope property and set its value to either application or domain.

When the value is set to domain, the application server searches for the wstrustClientBinding at the file system location that contains general binding documents.

When the value is set to application, the application server searches for the wstrustClientBinding at the file system location that contains application-specific binding documents.

When the wstrustClientBindingScope property is not specified, the default behavior of the application server is to search for application-specific bindings and then search for general bindings.

If the wstrustClientBinding cannot be located, the application server uses the default bindings.

g. Optional: If you want to modify the default trust client SOAP version, which is the same as the application client, specify a new value for the wstrustClientSoapVersion custom property.

Set the wstrustClientSoapVersion custom property to 1.1 to change to SOAP Version 1.1.

Set the wstrustClientSoapVersion custom property to 1.2 to change to SOAP Version 1.2.

h. Click **Apply** and then click **Save**.

If further modifications to the wstrustClientBinding configuration are needed, and the wstrustClientBinding property is pointing to an application-specific binding, for example in this case, SamlTCSample, you must attach the application-specific binding to the web services client before you can complete the modifications. The attachment is temporary. As detailed in the previous steps, you can detach the modified application-specific binding from the web service client after the modification is completed.

Before proceeding to the next step, verify that the SSL certificate from the external STS exists in the NodeDefaultTrustStore. See the Before you begin section for more information.

4. Restart the web services client application so that the policy set attachment modifications can take effect.

   The policy set and binding attachment information are updated when the application restarts, but the updated information in the general binding is not reflected at run time until all the general bindings are refreshed.

   The application must be restarted after the general bindings are reloaded to take advantage of any updates. Refer to the Reload the Client and Provider general bindings and restart the applications step that follows for more information.

5. Attach the SAML policy set and bindings to the provider application.
   a. Attach the appropriate SAML policy set to the web services provider.
      1) In the administrative console, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service provider policy sets and bindings**.
      2) Select the web services provider resource (JaxWSServicesSamples).
      3) Click **Attach Policy Set**.
      4) Select the appropriate SAML policy for the web services provider.
   b. Assign the Saml Bearer Provider sample general binding.
      1) Select the web services provider resource again.
      2) Click **Assign Binding**.
      3) Select **Saml Bearer Provider sample**.
6. Configure the web services provider bindings.

a. If the web services provider bindings are not already on the service provider policy sets and bindings page for JaxWSServicesSamples, click **WebSphere enterprise applications > JaxWSServicesSamples > Service provider policy sets and bindings > Saml Bearer Provider sample**.
b. Click **WS-Security > Authentication and protection**.
c. In the Authentication tokens table, click either **con_saml11token**, or **con_saml20token**.
d. Click **Callback handler**.

   The Callback handler page of the administrative console is used to configure the SAML token issuer digital signature validation binding data for the external STS.
e. Optional: Set the signatureRequired custom property to `false` if you want to waive digital signature validation.

   You can set the signatureRequired custom property to `false`, if you want to waive digital signature validation. However, a good security practice is to require SAML assertions to be signed, and always require issuer digital signature validation. False is the default value for this property.
f. Optional: Set the trustAnySigner custom property to `true` if you want to allow no signer certificate validation.

   The Trust Any certificate configuration setting is ignored for the purposes of SAML signature validation. This property is only valid if the signatureRequired custom property is set to `true`, which is the default value for that property.
g. Complete the following actions if assertions are signed by the STS, the signatureRequired custom property is set to the default value of `true`, and the trustAnySigner custom property is set to the default value of `false`.
   - Add a certificate to the truststore for the provider that allows for the external STS signing certificate to pass the trust validation, such as the STS signing certificate itself or its root CA certificate.
   - Set the trustStorePath custom property to a value that matches the trust store file name. This value can be fully-qualified or use keywords such as `${USER_INSTALL_ROOT}`.
   - Set the trustStoreType custom property to a value that matches the key store type. Supported keys store types include: jks, jceks, and pkcs12.
   - Set the trustStorePassword custom property to a value that matches the truststore password. The password is stored as a custom property and is encoded by the administrative console.
   - **Optional:** Set the trustedAlias custom property to a value such as samlissuer. If this property is specified, the X.509 certificate represented by the alias is the only STS certificate that is trusted for SAML signature verification. If this custom property is not specified, the web services runtime environment uses the signing certificate inside the SAML assertions to validate the SAML signature and then verifies the certificate against the configured truststore.
h. Optional: Configure the recipient to validate either the issuer name or the certificate SubjectDN of the issuer in the SAML assertion, or both.

   You can create a list of trusted issuer names, or a list of trusted certificate SubjectDNs, or you can create both types of lists. If you create both issuer name and SubjectDN lists, both issuer name and SubjectDN are verified. If the received SAML issuer name or signer SubjectDN is not in the trusted lists, SAML validation fails, and an exception is issued.

   The following example shows how to create a list of trusted issuers and trusted SubjectDNs. For each trusted issuer name, use trustedIssuer_*n* where *n* is a positive integer. For each trusted SubjectDN, use trustedSubjectDN_*n* where *n* is a positive integer. If you create both types of lists, the integer *n* must match in both lists for the same SAML assertion. The integer *n* starts with 1, and increments by 1.

   In this example, you trust a SAML assertion with the issuer name WebSphere/samlissuer, regardless of the SubjectDN of the signer, so you add the following custom property:

   ```
   <properties value="WebSphere/samlissuer" name="trustedIssuer_1"/>
   ```

In addition, you trust a SAML assertion issued by IBM/samlissuer, when the SubjectDN of the signer is ou=websphere,o=ibm,c=us, so you add the following custom properties:

```
<properties value="IBM/samlissuer" name="trustedIssuer_2"/>
<properties value="ou=websphere,o=ibm,c=us" name="trustedSubjectDN_2"/>
```

i. Decrypt the SAML assertion.

If the SAML assertion is encrypted by the STS, the SAML token appears in the SOAP Security header as an EncryptedAssertion element instead of an Assertion element. To decrypt the SAML assertion, you must configure the private key that corresponds to the public key that was used to encrypt the assertion on the STS.

The following callback handler custom properties must be set to the value described in the following table for the recipient to decrypt the SAML assertion.

Custom property	Value
keyStorePath	Keystore location
keyStoreType	Matching keystore type Supported keystore types include: jks, jceks, and pkcs12
keyStorePassword	Password for the keystore
keyAlias	The alias of the private key used for SAML encryption
keyName	The name of the private key used for SAML encryption
keyPassword	The password for the key name

7. Optional: You can configure the caller binding to select a SAML token to represent the requester identity. The Web Services Security runtime environment uses the specified JAAS login configuration to acquire the user security name and group membership data from the user registry using the SAML token NameId or NameIdentifier as the user name.
   a. Click **WebSphere enterprise applications** > ***JaxWSServicesSamples*** > **Service provider policy sets and bindings** > **Saml Bearer Provider sample** > **WS-Security** > **Callers**.
   b. Click **New** to create the caller configuration
   c. Specify a **Name**, such as `caller`.
   d. Enter a value for the **Caller identity local part**.

      For SAML 1.1 tokens enter:`http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1`

      For SAML 2.0 tokens enter:`http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0`
   e. Click **Apply** and **Save**.
8. Reload the Client and Provider general bindings and restart the applications.

   When the information in general bindings is updated, the new settings are not immediately reflected at run time. An updated general binding must be reloaded by the policy set manager in the application server before any updates will take effect. You can reload any updated policy sets and general bindings by stopping and restarting the application server or using the refresh command on the PolicySetManager MBean in wsadmin. For more information on refreshing the policy set manager, see the topic *Refreshing policy set configurations using wsdamin scripting*.

   To reload the Client and Provider general bindings and restart the applications, complete one of the following actions:
   - Restart the application serve, or
   - Refresh the PolicySetManager MBean, and then restart the Client and Provider web services applications.

Results

When you have completed the procedure, the JaxWSServicesSamples web services application is ready to use the SAML Bearer default policy set, the Saml Bearer Client sample, and the Saml Bearer Provider sample general bindings.

Configuring client and provider bindings for the SAML holder-of-key symmetric key token:

Configure the client and provider policy set attachments and bindings for the SAML holder-of-key token. This configuration scenario uses a symmetric key.

Before you begin

After installing, you must create one or more new server profiles, or add SAML configuration settings to an existing profile. For example, in a network deployment environment, there are multiple profiles. Read about setting up the SAML configuration for more information.

About this task

WebSphere Application Server with SAML provides numerous default SAML token application policy sets and several general client and provider binding samples. Before you can configure the client and provider bindings for the SAML holder-of-key token, you must import one of these default policy sets: SAML20 HoK Symmetric WSSecurity default or SAML11 HoK Symmetric WSSecurity default. The SAML11 policy sets are almost identical to the SAML20 policy sets, except that SAML20 HoK Symmetic WSSecurity default policy set supports the SAML Version 2.0 token type, while the SAML11 HoK Symmetric WSSecurity default policy set supports the Version 1.1 token type.

The SAML token policy is defined by a CustomToken extension in the application server. To create the CustomToken extension, define the SAML token configuration parameters in terms of custom properties in the client and provider binding document. The Saml HoK Symmetric Client sample and the Saml HoK Symmetric Provider sample general bindings contain the essential configuration for the custom properties. The client and provider sample bindings contain both SAML11 and SAML20 token type configuration information and therefore can be used with both SAML11 and SAML20 policy sets. Depending on how you plan to implement the SAML tokens, you must modify the property values in the installed binding samples. Examples of the properties and property values are provided in the procedure.

The procedure for modifying the binding sample begins with configuring the web services client policy set attachment, then configuring the web services provider policy set attachment. The example presented in the procedure uses the sample web services application JaxWSServicesSamples.

Procedure

1. Import two default policy sets: SAML20 HoK Symmetric WSSecurity default, and the Username WSHTTPS default.
   a. Click **Services** > **Policy sets** > **Application policy sets**.
   b. Click **Import**.
   c. Select **From Default Repository**.
   d. Select the two default policy sets.
   e. Click **OK** to import the policy sets.

   If you do not want the server to automatically request a SAML token from the Security Token Service (STS) using the WS-Trust client, you can skip steps 2, 3, and 4, and continue to step 5. For example, you can skip steps 2, 3, and 4, if web services act as a client and self-issues a SAML token based on the original SAML token, or the web services client has already acquired a SAML token and cached the SAML token in the RequestContext.

2. Attach a policy set for the trust client. Click **Applications > Application types > WebSphere enterprise applications >** ***JaxWSServicesSamples*** **> Service client policy sets and bindings**. The steps which pertain to attaching and detaching the policy set, and configuring the trust client binding, are required only if an application-specific binding is used to access the external STS. You can skip these steps, and go to the step that discusses configuring communication with the STS, if you use a general binding to access the external STS.
   a. Select the check box for the web services client resource.
   b. Click **Attach Client Policy Set**.
   c. Select the policy set, **Username WSHTTPS default**.

   This step attaches the policy set to the web services trust client, as you would do to use this policy set for the application client to access the target web services. However, since you plan to use the Username WSHTTPS default policy set to access an external STS instead, the policy set is only temporarily attached to the Web services client. The purpose of this step is to allow you to use the administrative console to create or to modify the client binding document.
3. Configure the trust client binding.
   a. Select the web services client resource again.
   b. In the Service client policy sets and bindings panel, click **Assign Binding**.
   c. Click **New Application Specific Binding** to create an application-specific binding.
   d. Specify a binding configuration name for the new application-specific binding. In this example, the binding name is **SamlTCSample**.
   e. Add the **SSL transport** policy type to the binding. Optionally, you can modify the NodeDefaultSSLSettings settings. Click **Security > SSL certificate and key management > SSL configurations > NodeDefaultSSLSettings**.
4. Add the **WS-Security** policy type to the binding, then modify the authentication settings.
   a. Click **Applications > Application types > WebSphere enterprise applications >** ***JaxWSServicesSamples*** **> Service client policy sets and bindings >** ***SamlTCSample*** **> Add > WS-Security > Authentication and protection > request:uname_token**.
   b. Click **Apply**.
   c. Select **Callback handler**.
   d. Specify a user name and password (and confirm the password) to authenticate the web services client to the external STS.
   e. Click **OK** and **Save**.
5. After the binding settings are saved, return to the Service client policy sets and bindings panel to detach the policy set and bindings.
   a. Click **Applications > Application types > WebSphere enterprise applications >** ***JaxWSServicesSamples*** **> Service client policy sets and bindings**.
   b. Click the check box for the web services client resource.
   c. Click **Detach client policy set**.

   The application-specific binding configuration you created in the previous steps is not deleted from the file system when the policy set is detached. This means that you can still use the application-specific binding you created to access the STS.
6. Download the unrestricted jurisdiction policy file. The SAML20 HoK Symmetric WSSecurity default security policy uses the 256 bit encryption key size, which requires the unrestricted Java Cryptography Extension (JCE) policy file. For more information, read the section Using the unrestricted JCE policy files in the Tuning Web Services Security topic.
7. Attach the SAML20 HoK Symmetric WSSecurity default policy set and assign the Saml HoK Symmetric Client sample binding to the client resource.
   a. Click **Applications** > **Application types** > **WebSphere enterprise applications** > ***JaxWSServicesSamples*** > **Service client policy sets and bindings**.

b. Select the web services client resource.
c. Click **Attach Client Policy Set**.
d. Select the policy set, **SAML20 HoK Symmetric WSSecurity default**.
e. Select the web services client resource again.
f. In the Service client policy sets and bindings panel, click **Assign Binding**.
g. Select the **Saml HoK Symmetric Client sample** general binding.
h. Click **Save**.

8. Configure the STS endpoint URL and the user name and password to authenticate to the STS.
   a. Click **Applications** > **Application types** > **WebSphere enterprise applications** > ***JaxWSServicesSamples*** > **Service client policy sets and bindings** > **Saml HoK Symmetric Client sample** > **WS-Security** > **Authentication and protection**.
   b. Click **gen_saml20token** in the Protection tokens table.
   c. Click **Callback handler**.
   d. Modify the **stsURI** property and specify the STS endpoint. If you do not use an external STS, and you want the application server to self-issue a holder-of-key assertion with a symmetric key, do not complete this step and go to step 8i.
   e. If necessary, modify the **wstrustClientPolicy** property and change the value to **Username WSHTTPS default**.
   f. Modify the **wstrustClientBinding** property and change the value to match the application-specific binding created in the previous steps. For this example, the value is **SamlTCSample**. This step attaches the WS-Trust client policy set. You can skip this step if you do not want the server to automatically request a SAML token from the STS using the WS-Trust client.
   g. Change the value of the **wstrustClientBindingScope** property, which controls how the application server searches for the binding. Set the property value to either **application** or **domain**. When the value is set to domain, the application server searches for the wstrustClientBinding at the file system location that contains general binding documents. When the value is set to application, the application server searches for the wstrustClientBinding at the file system location that contains application-specific binding documents. When the wstrustClientBindingScope property is not specified, the default behavior of the application server is to search for application-specific bindings and then search for general bindings. If the wstrustClientBinding can not be located, the application server uses the default bindings.
   h. Verify that the value of the `confirmationMethod` property is `Holder-of-key`.
   i. Verify that the value of the `keyType` property value is `http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey` or the `symmetrickey` alias. The `wstrustClientWSTNamespace` property determines how the `symmetrickey` alias is interpreted. In this case it is assumed that it is set to the WS-Trust 1.3 namespace. If it had a value of WS-Trust 1.2, the `symmetrickey` alias is interpreted as `http://schemas.xmlsoap.org/ws/2005/02/trust/SymmetricKey`.
   j. Optional: You can modify the default trust client SOAP version, which is the same as the application client. Set the custom property **wstrustClientSoapVersion** to the value `1.1` to change to SOAP Version 1.1, or set the property to the value `1.2` to change to SOAP Version 1.2.
   k. Optional: If you are not using an external STS, and you want the application server to self-issue a holder-of-key assertion with a symmetric key, set the custom property **recipientAlias** to the value of the key alias of the target service. Specifying this property protects the symmetric key for the target service. This alias must be a valid key alias that is contained in the configured trust store of the SAML issuer. The TrustStorePath property specifies the location of the trust store file. The TrustStorePath property is defined in the `SAMLIssuerConfig.properties` file for the application server. For example, the location of the `SAMLIssuerConfig.properties` file at the server level on a WebSphere Application server is:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/nodes/$NODENAME/servers/$SERVERNAME/SAMLIssuerConfig.properties
```

The location of this file at the cell level on a WebSphere Application server is:

app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/sts/SAMLIssuerConfig.properties

l. Click **Apply** and **Save**.

9. Optional: If further modifications to the wstrustClientBinding configuration are needed, and the wstrustClientBinding property is pointing to an application-specific binding, you must attach the application-specific binding to the web services client before you can complete the modifications. The attachment is temporary. As detailed in the previous steps, you can detach the modified application-specific binding from the web service client after the modification is completed.
10. Import the SSL certificate from the external STS.
    a. Click **Security > SSL certificate and key management > Manage endpoint security configurations >** ***server_or_node_endpoint*** **> Keystores and certificates > NodeDefaultTrustStore > Signer certificates**.
    b. Click **Retrieve from port**.
    c. Specify the host name and port number of the external STS server, and assign an alias to the certificate. Use the SSL STS port.
    d. Click **Retrieve signer information**.
    e. Click **Apply** and **Save** to copy the retrieved certificate to the NodeDefaultTrustStore object.
11. Restart the web services client application so that the policy set attachment modifications can take effect.
12. Attach the **SAML20 HoK Symmetric WSSecurity default** policy set to the web services provider.
13. Download the unrestricted jurisdiction policy file. The SAML20 HoK Symmetric WSSecurity default security policy uses the 256 bit encryption key size, which requires the unrestricted Java Cryptography Extension (JCE) policy file. For more information, read the section Using the unrestricted JCE policy files in the Tuning Web Services Security applications topic.
14. Assign the **Saml HoK Symmetric Provider sample** general binding.
15. Click **Applications** > **Application types** > **WebSphere enterprise applications** > ***JaxWSServicesSamples*** > **Service provider policy sets and bindings** > **Saml HoK Symmetric Provider sample** > **WS-Security** > **Authentication and protection**.
    a. Click **con_saml20token** in the Authentication tokens table.
    b. Click the **Callback handler** link.
    c. Use this panel to configure the embedded symmetric key decryption configuration, and the SAML token issuer digital signature validation to the external STS, as described in the following step.
16. Configure the binding data to decrypt the embedded secret key, or the SAML assertion that is protected by the public key from the recipient. The STS must have access to the public key of the recipient. There are two options to configure the keys for decryption:
    - Option 1: Configure the keystore and a private key, as follows:
      a. Verify that the **Keystore name** field has the value `custom`.
      b. Click **Custom keystore configuration** to view and edit the keystore configuration.
      c. Verify that the initial value for the key file is *app_server_root*/etc/ws-security/samples/enc-service.jceks.
    - Option 2: Set the custom properties in the callback handler as follows:

Custom property	Value
keyStorePath	Keystore location
keyStoreType	Matching keystore type Supported keystore types include: jks, jceks, and pkcs12
keyStorePassword	Password for the keystore
keyAlias	The alias of the public key used for SAML encryption
keyName	The name of the public key used for SAML encryption

Custom property	Value
keyPassword	The password for the key name

17. Add the external STS signing certificate to the truststore. This step is required if the SAML assertions are signed by the STS and the **signatureRequired** custom property is not specified, or has a value of **true**. This truststore is configured for the service provider.
    a. Set the custom property **trustStoreType** to match the keystore type. Supported keystore types include: **jks**, **jceks**, and **pkcs12**.
    b. Set the custom property **trustStorePath** to the keystore file location. For example, `*app_server_root*/etc/ws-security/samples/dsig-issuer.jceks`. The file dsig_issuer.jceks is not provided when WebSphere Application Server is installed, so you must create the file.
    c. Set the custom property **trustStorePassword** to the encoded value of the store password. The password is stored as a custom property and is encoded by the administrative console.
    d. Optional: You can set the custom property **trustedAlias** to a value such as `samlissuer`. Do not set the trustedAlias property if the SAML token is signed by different signers, for example, if the STS delegates token requests to different token providers, and each provider signs with a certificate. If this custom property is not specified, the web services runtime environment uses the signing certificate password in the SAML assertions to validate the signature and then verifies the certificate against the configured truststore.
    e. Optional: You can set the custom property **trustAnySigner** to the value `true` to allow no signer certificate validation. The Trust Any certificate configuration setting is ignored for the purposes of SAML signature validation.
    f. Optional: You can set the custom property **signatureRequired** to `false`, which waives digital signature validation. However, a good security practice is to require SAML assertions to be signed and always require issuer digital signature validation.
    g. Optional: You can configure the recipient to validate either the issuer name or the certificate SubjectDN of the issuer in the SAML assertion, or you can validate both. Create a list of trusted issuer names, or a list of trusted certificate SubjectDNs, or both types of lists. If you create both issuer name and SubjectDN lists, both issuer name and SubjectDN are verified. If the received SAML issuer name or signer SubjectDN is not in the trusted list, SAML validation fails, and an exception is issued. This example shows how to create a list of trusted issuers and trusted SubjectDNs.

       For each trusted issuer name, use *trustedIssuer_n* where *n* is a positive integer. For each trusted SubjectDN, use *trustedSubjectDN_n* where *n* is a positive integer. If you create both types of lists, the integer *n* must match in both lists for the same SAML assertion. The integer *n* starts with 1, and increments by 1.

       In this example, you trust a SAML assertion with the issuer name `WebSphere/samlissuer`, regardless of the SubjectDN of the signer, so you add the following custom property:

```
<properties value="WebSphere/samlissuer" name="trustedIssuer_1"/>
```

       In addition, you trust a SAML assertion issued by `IBM/samlissuer`, when the SubjectDN of the signer is `ou=websphere,o=ibm,c=us`, so you add the following custom properties:

```
<properties value="IBM/samlissuer" name="trustedIssuer_2"/>
<properties value="ou=websphere,o=ibm,c=us" name="trustedSubjectDN_2"/>
```

       By default, WebSphere Application Server trusts all SAML issuers when you do not define a `trustedIssuer_*n*` value. Without knowing this default behavior, you might mistakenly accept SAML assertions that are issued by an authorized STS
    h. Optional: You can add a list of non-root certificate authority (CA) certificates that can be used to check the signature of the SAML token. To add non-root certificates, add a custom property named `X509PATH_*n*` where *n* is a non-negative integer as the value for the non-root certificates.
    i. Optional: You can add a list of certificate revocation lists (CRLs) that can be used to validate the signature of the SAML token. To add CRLs, add a custom property named `CRLPATH_*n*` where *n* is a non-negative integer as the value for the CRLs.

j. Click **Apply** and **Save**.

18. Optional: You can configure the caller binding to select a SAML token to represent the requester identity. The Web Services Security runtime environment uses the specified JAAS login configuration to acquire the user security name and group membership data from the user registry using the SAML token NameId or NameIdentifier as the user name.
    a. Click **Applications** > **Application types** > **WebSphere enterprise applications** > ***JaxWSServicesSamples*** > **Service provider policy sets and bindings** > **Saml HoK Symmetric Provider sample** > **WS-Security** > **Callers**.
    b. Click **New** to create the caller configuration
    c. Specify a **Name**, such as `caller`.
    d. Enter a value for the **Caller identity local part**. For example, `http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0`, which is the local part of the CustomToken element in the attached WS-Security policy.
    e. Click **Apply** and **Save**.
19. Restart the web services provider application so that the policy set attachment modifications can take effect.

Results

When you have completed the procedure, the JaxWSServicesSamples web services application is ready to use the SAML20 HoK Symmetric default policy set, the Saml HoK Symmetric Client sample, and the Saml HoK Symmetric Provider sample general bindings.

SAMLIssuerConfig.properties file:

When creating a new SAML token, you can specify configuration properties to control how the token is configured. The configuration properties are stored in a properties file containing name/value pairs. The properties describe provider-side information such as the issuer location, and the keystore and truststore file paths.

Starting with WebSphere Application Server version 8, you can also use the administrative console or the `setSAMLIssuerConfigInBinding` command task to specify a self-issued SAML token's configuration as custom properties in the requester's outbound configuration in the general bindings or in the application-specific bindings. You can also specify a self-issued SAML token's configuration as custom properties of `com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext` objects when programming to Web Services Security (WSS) Application Programming interfaces (APIs). Migrate self-issued SAML token configuration data from the `SAMLIssuerConfig.properties` file to the bindings. Refer to the “Managing self-issue SAML token configuration using wsadmin commands” section for additional information.

The `SAMLIssuerConfig.properties` file usage is deprecated in WebSphere Application Server version 8. Do not specify a `SAMLIssuerConfig.properties` file using a Java System property. The `com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory.newDefaultProviderConfig()` method returns a `com.ibm.wsspi.wssecurity.saml.config.ProviderConfig` object with empty contents when no `SAMLIssuerConfig.properties` file is specified, which is the recommended programming style. Use `ProviderConfig` setter methods to populate its contents.

File Location

A single configuration file, SAMLIssuerConfig.properties, containing the provider-side properties is created and stored on each server. On a WebSphere server, the file is located in the server-level repository, or in the cell-level repository. In an environment that is not based on WebSphere, the file location is defined by a Java system property. The name of this property is `com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath`.

For example, the location of the file at the server level on a WebSphere server is:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/nodes/$NODENAME/servers/$SERVERNAME/SAMLIssuerConfig.properties
```

The location of the file at the cell level on a WebSphere server is:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/sts/SAMLIssuerConfig.properties
```

SAML token properties

The following table describes the provider configuration properties.

Table 158. Properties to configure provider information for a new SAML token. Use these properties to control how the token is created.

Property name	Sample property value	Property description
com.ibm.wsspi.wssecurity.dsig.oldEnvelopedSignature	true	Use only if you are setting the com.ibm.wsspi.wssecurity.dsig.enableEnvelopedSignatureProperty JVM custom property to `true`. See the topic *Java Virtual Machine (JVM) custom properties* for a description of when you might want to use this JVM custom property.
IssuerURI	http://www.websphere.ibm.com/SAML/SelfIssuer	The URI of the issuer.
TimeToLiveMilliseconds	3600000	Amount of time before expiration of the token.
KeyStoreRef	*MyKeyStoreRef*	A reference to a managed keystore from security.xml.
KeyStorePath	*app_server_root*/etc/ws-security/samples/dsig-receiver.ks	The location of the keystore file. **Note:** You must modify this value from the default value to match the path location for your system.
KeyStoreType	JKS	The keystore type.
KeyStorePassword	*password*	The password of the keystore file (the password must be XOR encoded). For more information, read about encoding passwords in files.
KeyAlias	*soapprovider*	The alias of the key as defined in the keystore file.
KeyName	CN=SOAPProvider, OU=TRL, O=IBM, ST=Kanagawa, C=JP	The name of the key as defined in the keystore file.
KeyPassword	*password*	The password of the private key as defined in the keystore file (the password must be XOR encoded).
TrustStoreRef	*MyTrustStoreRef*	A reference to a managed keystore from security.xml.
TrustStorePath	*app_server_root*/etc/ws-security/samples/dsig-receiver.ks	The location of the truststore file. **Note:** You must modify this value from the default value to match the path location for your system.
TrustStoreType	JKS	The truststore type.
TrustStorePassword	*password*	The password of the truststore file.
AttributeProvider	*com.mycompany.SAML.AttributeProviderImpl*	Implementation class of attribute provider.
NameIDProvider	*com.mycompany.SAML.NameIDProviderImpl*	Implementation class of name ID provider.

Example

See the following example of a SAML token configuration properties file:

```
IssuerURI=http://www.websphere.ibm.com/SAML/SelfIssuer
TimeToLiveMilliseconds=3600000
KeyStorePath=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
KeyStoreType=JKS
KeyStorePassword={xor}LDotKTot
KeyAlias=soapprovider
KeyName=CN=SOAPProvider, OU=TRL, O=IBM, ST=Kanagawa, C=JP
```

```
KeyPassword={xor}LDotKTot
TrustStorePath=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks
TrustStoreType=JKS
TrustStorePassword={xor}LDotKTot
```

Configuring client and provider bindings for the SAML sender-vouches token:

You can configure the client and provider policy set attachments and bindings for the SAML sender-vouches token. A SAML sender-vouches token is a SAML token that uses the sender-vouches subject confirmation method. The sender-vouches confirmation method is used when a server needs to propagate the client identity or behavior of the client.

Before you begin

- Before you can use a SAML sender-vouches token, you must create one or more new server profiles, or add SAML configuration settings to an existing profile.

  Refer to the topic *Creating application server profiles* for more information about creating a server profile.

  Refer to the various topics that describe how to configure SAML for more information about how to add SAML configuration settings to an existing profile.
- Determine which type of security you want to use to protect the integrity of SOAP messages and SAML tokens so that a receiver can verify that the message contents and SAML tokens were not modified by unauthorized parties. You must use either message-level security or HTTPS transport.

  As stated in section 3.5.2.1 of the SAML Token Profile specification:

  "To satisfy the associated confirmation method processing of the receiver, the attesting entity MUST protect the vouched for SOAP message content such that the receiver can determine when it has been altered by another party. The attesting entity MUST also cause the vouched for statements (as necessary) and their binding to the message contents be protected such that unauthorized modification be detected."

  You can use either transport-level or message-level security to meet this SAML sender-vouches requirement:

  You must use either message-level security or HTTPS transport to protect the sender-vouches token.
  - To utilize HTTP transport-level security, configure the HTTPS transport.
  - To utilize message-level security, the SAML Token Profile Specification suggests that the attesting entity "sign the relevant message content and assertions".

  To sign the relevant message content and assertions, you must at least sign the SAML token (the assertion). The relevant content is dependent on your application. The specification recommends that:
  - The sender at least sign the SOAP Body and SAML Assertion together to meet the relevant message content requirement.
  - The consumer verify that the SAML token is signed with SOAP body when using SAML sender-vouches.

About this task

This procedure describes the steps you must complete to digitally sign a SAML token. It does not describe any of the SAML Token Profile OASIS standard requirements for SAML sender-vouches or SAML bearer tokens regarding message parts that must be signed.

The example provided in this procedure uses the sample web services application JaxWSServicesSamples.

The procedure for creating the sender-vouches policy set begins with creating a new SAML sender-vouches policy.

Procedure

1. Create the SAML sender-vouches policy, and configure the message parts.

   You must create a SAML sender-vouches policy set, based on the SAML bear policy before you can configure the client and provider bindings for the SAML sender-vouches token, After you create the policy set, you must attach the bindings to the JAX-WS client and provider applications. For more information about the bearer policy sets, see the topic *Configuring client and provider bindings for the SAML bearer token*.

   Several default SAML token application policy sets and several general client and provider binding samples are provided with the product. A policy set that is used for a SAML sender-vouches token is similar to one that is used for a SAML bearer token . The following procedure describes how to create a sender-vouches policy set based on a SAML bearer token policy set.

   Unless they are imported as a copy, the SAML20 Bearer WSSecurity default and SAML20 Bearer WSHTTPS default policies cannot be updated for use with SAML sender-vouches tokens. SAML20 Bearer WSSecurity default and SAML20 Bearer WSHTTPS default policies are not configured to sign the SAML token. To meet the requirement of SAML sender-vouches, the policy must be updated to sign the SAML token. Therefore, either the policy must be imported as a copy, or you must make a copy of the policy. The following procedure makes a copy of the policy.

   a. Import the required Policy Sets.

      The Before you begin section of the topic *Configuring client and provider bindings for the SAML bearer token* describes how to import the Username WSHTTPS default and the SAML Bearer policy of the desired type. For example, SAML20 Bearer WSSecurity default is used for SAML 2.0 sender-vouches tokens using HTTP.

   b. Make a copy of the desired imported SAML Bearer policy that you can edit.

      1) In the administrative console, click **Services** > **Policy sets** > **Application policy sets**.
      2) Select the imported SAML Bearer policy you want to copy.

         For example, you might select **SAML20 Bearer WSSecurity default**.
      3) Click Copy... .
      4) Specify the desired name in the **Name** field. For example, you might specify `SAML20 sender-vouches`.
      5) Click OK.

   c. Edit the new SAML sender-vouches policy to add digital signature of the SAML token.

      1) In the administrative console, click **Services** > **Policy sets** > **Application policy sets** .
      2) Select the policy you just created.

         Using the preceding example, you would select **SAML20 sender-vouches**.

   d. From the administrative console, edit the SAML policy set, then click **WS-Security > Main policy > Request message part protection**.

   e. Under **Integrity protection**, click **Add**.

   f. Enter a part name for **Name of part to be signed**; for example, `saml_part`.

   g. Under **Elements in Part**, click **Add**.

   h. Select **XPath Expression**.

   i. Add two XPath expressions.

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
and local-name()='Envelope']/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
and local-name()='Security']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
and local-name()='SecurityTokenReference']
```

```
/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
and local-name()='Envelope']/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
and local-name()='Security']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
and local-name()='SecurityTokenReference']
```

   j. Click **Apply** and **Save**.

k. If an application has never been started using this policy, no further action is required. Otherwise, either restart the application server or follow the instructions in the *Refreshing policy set configurations using wsadmin scripting* article, for the application server to reload the policy set.

2. Configure the trust client

   If you will be using general bindings to access the external STS, skip to Attach the policy set and bindings to the client application step.

   If you will be using application specific bindings to access the external STS, complete the following substeps.

   a. Temporarily Attach a policy set for the trust client to the web services client application so that bindings can be configured.

      Attaching a policy set for the trust client allows you to use the administrative console to create, and then modify the client binding document bindings. You only have to complete this action if an application-specific binding is used to access the external STS.

      1) In the administrative console, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings**.
      2) Select the web services client resource (JaxWSServicesSamples).
      3) Click **Attach Client Policy Set**.
      4) Select the policy set **Username WSHTTPS default**.

   b. Create the trust client binding.

      1) Select the web services client resource again (JaxWSServicesSamples).
      2) Click **Assign Binding**.
      3) Click **New Application Specific Binding** to create an application-specific binding.
      4) Specify a binding configuration name for the new application-specific binding. In this example, the binding name is SamlTCSample.

   c. Add the SSL transport policy type to the binding,

      Click **Add > SSL transport** and then click **OK**.

   d. Add the WS-Security policy type to the binding, then modify the authentication settings for the trust client.

      1) If the he WS-Security policy type is not already in the SamlTCSample binding definition, click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings > SamlTCSample**.
      2) Click **Add > WS-Security > Authentication and protection > request:uname_token**.
      3) Click **Apply**.
      4) Select **Callback handler**
      5) Specify a user name and password for the web services client to authenticate to the external STS.
      6) Click **OK**, and then click **Save**.

   e. After the binding settings are saved, return to the **Service client policy sets and bindings** panel, and detach the policy set and bindings.

      1) Click either **Service client policy sets and bindings** in the navigation for this page, or **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings**.
      2) Select the web services client resource (JaxWSServicesSamples), and then click **Detach client policy set**.

      The application-specific binding configuration you just created is not deleted from the file system when the policy set is detached. Therefore, you can still use the application-specific binding you created to access the STS with the trust client.

3. Attach the SAML sender-vouches policy set and create new application specific bindings for the client application

   You must use application-specific custom bindings instead of general bindings for sender-vouches. Therefore, if you configure sender-vouches policy sets and bindings from attached bearer token policy sets and bindings, you must ensure that the assigned bindings are application-specific bindings.

   a. Attach the desired SAML policy set to the web services client application.
      1) Click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service client policy sets and bindings**.
      2) Select the web services client resource (**JaxWSServicesSamples**).
      3) Click **Attach Client Policy Set**.
      4) Select the SAML policy that you created.

         For example, you might select **SAML20 sender-vouches**.
   b. Create new application specific bindings for the client.
      1) Select the web services client resource again (**JaxWSServicesSamples**).
      2) Click **Assign Binding**.
      3) Select **New Application Specific Binding....**
      4) Specify a binding configuration name for the new application-specific binding.

         In this example, the binding name is SamlSenderVouchesClient.
      5) Click **Add** > **WS-Security**.
4. Edit the SAML token generator in application specific client bindings.
   a. Click **Authentication and protection**.
   b. Under Authentication tokens, click either **request:SAMLToken20Bearer** or **request:SAMLToken11Bearer**.
   c. Click **Apply**.
   d. Click **Callback handler**.
   e. Add the following custom properties.
      - `confirmationMethod=sender-vouches`
      - `keyType=http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer`
      - `stsURI=`*SecurityTokenService_address*

        For example, you might specify `https://example.com/Trust/13/UsernameMixed` for *SecurityTokenService_address*.
      - `wstrustClientPolicy=Username WSHTTPS default`.
      - `wstrustClientBinding=`*value*

        The value you specify for wstrustClientBinding must match the name of the application specific binding of the trust client that you created in the previous steps. For example, if in the previous steps, you created an application specific binding named SamlTCSample, you must specify SamlTCSample as the value for the wstrustClientBinding property.
      - `wstrustClientSoapVersion=`*value*

        Specify a value of `1.1` for this property if you want to use SOAP Version 1.1.

        Specify a value of `1.2` for this property if you want to use SOAP Version 1.2.
   f. Click **OK**.
   g. Click **WS-Security** in the navigation for this page.
5. Configure general digital signature in the client bindings.
   a. Configure a Certificate Store.
      1) Click **Keys and Certificates**.
      2) Under Certificate store, click **New Inbound...** .
      3) Specify `name=`*`clientCertStore`*.

4) Specify `Intermediate X.509 certificate=${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`.
5) Click **OK**.

b. Configure a Trust Anchor.
1) Under Trust anchor, click **New...**
2) Specify `name=`*`clientTrustAnchor`*.
3) Click **External Keystore** .
4) Specify `Full path=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks`.
5) Specify `Password=`*`client`*.
6) Click **OK**.
7) Click **WS-Security** in the navigation for this page.

c. Configure the Signature Generator.
1) Click **Authentication and protection > AsymmetricBindingInitiatorSignatureToken0** (signature generator), and then click **Apply**.
2) Click **Callback handler**
3) Specify `Keystore=custom`.
4) Click **Custom keystore configuration**, and then specify

```
Full path==${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks
Keystore password=client
Name=client
Alias=soaprequester
Password=client
```

5) Click **OK**, **OK**, and **OK**.

d. Configure the Signature Consumer.
1) Click **AsymmetricBindingRecipientSignatureToken0** (signature consumer), and then click **Apply**.
2) Click **Callback handler**.
3) Under Certificates, click the **Certificate store** radial button, and specify:

```
Certificate store=clientCertStore
Trusted anchor store=clientTrustAnchor
```

4) Click **OK**, and **OK**.

e. Configure the request Signing Information.
1) Click **request:app_signparts**, and specify `Name=`*`clientReqSignInfo`*.
2) Under Signing key information, click **New** , and then specify:

```
Name=clientReqSignKeyInfo
Type=Security Token reference
Token generator or consumer name=AsymmetricBindingInitiatorSignatureToken0
```

3) Click **Ok**, and then click **Apply**.
4) Under Message part reference, select **request:app_signparts** .
5) Click **Edit**.
6) Under Transform algorithms, click **New**
7) Specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
8) Click **OK**, **OK**, and **OK**.

f. Configure the response Signing Information.
1) Click **response:app_signparts**, and specify `Name=`*`clientRespSignInfo`*.
2) Click **Apply**.
3) Under Signing key information, click **New** , and then specify:

```
Name=clientRspSignKeyInfo
```

`Token generator or consumer name=AsymmetricBindingRecipientSignatureToken0`

4) Click **Ok**.
5) Under Signing key information, click **clientRspSignKeyinfo** , and then click **Add**.
6) Under Message part reference, select **response:app_signparts** .
7) Click **Edit**.
8) Under Transform algorithms, click **New**
9) Specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
10) Click **OK**, **OK**, and **OK**.

6. Configure digital signature for the SAML token in the client bindings.
   a. Modify the currently configured outbound Signed message part bindings to include the new SAML part that you created.

      Under **Request message signature and encryption protection**, select the part reference whose status is set to `Configured`. This part reference will most likely be **request:app_signparts**.
      1) From the **Available** list under Message part reference, select the name of the part to be signed, as created in step 1; for example, `saml_part`.
      2) Click **Add**, and then click **Apply**.
      3) In the **Assigned** list under Message part reference, highlight the name of the part you added; for example, `saml_part`.
      4) Click **Edit**.
      5) For the **Transform algorithms** setting, click **New**.
      6) Select **http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform**.
      7) Click **OK**, click **OK**, and then click **OK** one more time.
   b. Update the SAML token consumer with the custom property to indicate digital signature with Security Token Reference

      Under Authentication tokens, select and edit the SAML token you want to sign.
      1) Under Custom property, click **New**.
      2) Enter `com.ibm.ws.wssecurity.createSTR` as the custom property name.
      3) Enter `true` as the value of the custom property.
      4) Click **Apply**, and then click **Save**.
   c. Restart the application.
7. Attach the SAML sender-vouches policy set, and create new application specific bindings for the provider application.
   a. Attach the desired SAML policy set to the web services client application.
      1) Click **Applications > Application types > WebSphere enterprise applications > JaxWSServicesSamples > Service provider policy sets and bindings**.
      2) Select the web services client resource (**JaxWSServicesSamples**).
      3) Click **Attach Policy Set**.
      4) Select the SAML policy that you created.

         For example, you might select **SAML20 sender-vouches**.
   b. Create new application specific bindings for the provider.
      1) Select the web services client resource again (**JaxWSServicesSamples**).
      2) Click **Assign Binding**.
      3) Select **New Application Specific Binding....**
      4) Specify a binding configuration name for the new application-specific binding.

         In this example, the binding name is SamlSenderVouchesProvider.

5) Click **Add > WS-Security**.

8. Edit the SAML token consumer in application specific provider bindings
   a. Click **Authentication and protection**.
   b. Under Authentication tokens, click either **request:SAMLToken20Bearer** or **request:SAMLToken11Bearer**.
   c. Click **Apply**.
   d. Click **Callback handler**.
   e. Add the following custom properties.
      - `confirmationMethod=sender-vouches`
      - `keyType=http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer`
      - `signatureRequired=true`
   f. Optional: Set the trustAnySigner custom property to `true` if you want to allow no signer certificate validation.

      The Trust Any certificate configuration setting is ignored for the purposes of SAML signature validation. This property is only valid if the signatureRequired custom property is set to `true`, which is the default value for that property.
   g. Complete the following actions if assertions are signed by the STS, the signatureRequired custom property is set to the default value of `true`, and the trustAnySigner custom property is set to the default value of `false`.
      - Add a certificate to the truststore for the provider that allows for the external STS signing certificate to pass the trust validation, such as the STS signing certificate itself or its root CA certificate.
      - Set the trustStorePath custom property to a value that matches the trust store file name. This value can be fully-qualified or use keywords such as `${USER_INSTALL_ROOT}`.
      - Set the trustStoreType custom property to a value that matches the key store type. Supported keys store types include: jks, jceks, and pkcs12.
      - Set the trustStorePassword custom property to a value that matches the truststore password. The password is stored as a custom property and is encoded by the administrative console.
      - **Optional:** Set the trustedAlias custom property to a value such as samlissuer. If this property is specified, the X.509 certificate represented by the alias is the only STS certificate that is trusted for SAML signature verification. If this custom property is not specified, the web services runtime environment uses the signing certificate inside the SAML assertions to validate the SAML signature and then verifies the certificate against the configured truststore.
   h. Optional: Configure the recipient to validate either the issuer name or the certificate SubjectDN of the issuer in the SAML assertion, or both.

      You can create a list of trusted issuer names, or a list of trusted certificate SubjectDNs, or you can create both types of lists. If you create both issuer name and SubjectDN lists, both issuer name and SubjectDN are verified. If the received SAML issuer name or signer SubjectDN is not in the trusted lists, SAML validation fails, and an exception is issued.

      The following example shows how to create a list of trusted issuers and trusted SubjectDNs. For each trusted issuer name, use trustedIssuer_*n* where *n* is a positive integer. For each trusted SubjectDN, use trustedSubjectDN_*n* where *n* is a positive integer. If you create both types of lists, the integer *n* must match in both lists for the same SAML assertion. The integer *n* starts with 1, and increments by 1.

      In this example, you trust a SAML assertion with the issuer name WebSphere/samlissuer, regardless of the SubjectDN of the signer, so you add the following custom property:

      ```
      <properties value="WebSphere/samlissuer" name="trustedIssuer_1"/>
      ```

      In addition, you trust a SAML assertion issued by IBM/samlissuer, when the SubjectDN of the signer is ou=websphere,o=ibm,c=us, so you add the following custom properties:

```
<properties value="IBM/samlissuer" name="trustedIssuer_2"/>
<properties value="ou=websphere,o=ibm,c=us" name="trustedSubjectDN_2"/>
```

   i. Click **APPLY**.
   j. Click **WS-Security** in the navigation for this page.

9. Configure general digital signature in the provider bindings.
   a. Configure a Certificate Store.
      1) Click **Keys and Certificates**.
      2) Under Certificate store, click **New Inbound...**.
      3) Specify:
         `Name=`*`providerCertStore`*
         `Intermediate X.509 certificate=${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`
      4) Click **OK**.
   b. Configure a Trust Anchor.
      1) Under Trust anchor, click **New...**
      2) Specify, `Name=`*`providerTrustAnchor`*.
      3) Click **External Keystore**, and specify:
         `Full path=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks`
         `Password=`*`server`*
      4) Click **OK**, and then click **WS-Security** in the navigation for this page.
   c. Configure the Signature Generator.
      1) Click **Authentication and protection > AsymmetricBindingRecipientSignatureToken0** (signature generator), and then click**Apply**.
      2) Click **Callback handler**
      3) Specify `Keystore=custom`.
      4) Click **Custom keystore configuration**, and then specify
         `Full path=${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks`
         `Keystore password=`*`server`*
         `Name=`*`server`*
         `Alias=`*`soapprovider`*
         `Password=`*`server`*
      5) Click **OK**, **OK**, and **OK**.
   d. Configure the Signature Consumer.
      1) Click **AsymmetricBindingInitiatorSignatureToken0** (signature consumer), and then click **Apply**.
      2) Click **Callback handler**.
      3) Under Certificates, click the Certificate store radial button, and specify:
         `Certificate store=`*`providerCertStore`*
         `Trusted anchor store=`*`providerTrustAnchor`*
      4) Click **OK**.
      5) Click **Authentication and protection** in the navigation for this page.
   e. Configure the request Signing Information.
      1) Click **request:app_signparts**, and specify `Name=`*`reqSignInf`*.
      2) Click **Apply**.
      3) Under Signing key information, click **New** , and then specify:
         `Name=`*`reqSignKeyInfo`*
         `Token generator or consumer name=AsymmetricBindingInitiatorSignatureToken0`
      4) Click **Ok**.

5) Under Signing key information, click **reqSignKeyinfo**, and then click **Add**.
6) Under Message part reference, click **request:app_signparts**.
7) Click **Edit**.
8) Under Transform algorithms, click **New**, and then specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
9) Click **OK**, **OK**, and **OK**.

f. Configure the response Signing Information.

1) Click **response:app_signparts**, and specify `Name=`*`rspSignInfo`*.
2) Click **Apply**.
3) Under Signing key information, click **New** , and then specify:
```
Name=rspSignKeyInfo
Type=Security Token reference
Token generator or consumer name=AsymmetricBindingRecipientSignatureToken0
```
4) Click **Ok**, and then click **Apply**.
5) Under Message part reference, select **response:app_signparts** .
6) Click **Edit**.
7) Under Transform algorithms, click **New**.
8) Specify `URL=http://www.w3.org/2001/10/xml-exc-c14n#`.
9) Click **OK**, **OK**, and **OK**.

10. Configure digital signature for the SAML token in the provider bindings.

a. Click **WS-Security** in the navigation for this page, and then click **Authentication and protection**.

b. Modify the currently configured inbound Signed message part bindings to include the new SAML part that you created.

Under **Request message signature and encryption protection**, select the part reference whose status is set to `Configured`. This part reference will most likely be **request:app_signparts**.

1) From the **Available** list under Message part reference, select the name of the part to be signed, as created in step 1; for example, `saml_part`.
2) Click **Add**, and then click **Apply**.
3) In the **Assigned** list under Message part reference, highlight the name of the part you added; for example, `saml_part`.
4) Click **Edit**.
5) For the **Transform algorithms** setting, click **New**.
6) Select **http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform**.
7) Click **OK**, click **OK**, and then click **OK** one more time.
8) Click **Save**.

11. Optional: You can configure the caller binding to select a SAML token to represent the requester identity. The Web Services Security runtime environment uses the specified JAAS login configuration to acquire the user security name and group membership data from the user registry using the SAML token NameId or NameIdentifier as the user name.

a. Click **WebSphere enterprise applications** > ***JaxWSServicesSamples*** > **Service provider policy sets and bindings** > **Saml Bearer Provider sample** > **WS-Security** > **Callers**.

b. Click **New** to create the caller configuration

c. Specify a **Name**, such as `caller`.

d. Enter a value for the **Caller identity local part**.

For SAML 1.1 tokens enter:`http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1`

For SAML 2.0 tokens enter:`http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0`

e. Click **Apply** and **Save**.

12. Restart the web services provider application so that the policy set attachment modifications can take effect.

Results

The JaxWSServicesSamples web services application is ready to use the new SAML sender-vouches policy set, SAML sender-vouches application specific client binding, and SAML sender-vouches application-specific provider binding.

Managing self-issue SAML token configuration using wsadmin commands:

The `SAMLIssuerConfig.properties` file usage is deprecated in WebSphere Application Server Version 8. You can use the `listSAMLIssuerConfig` and `updateSAMLIssuerConfig` wsadmin command tasks to read and modify the `SAMLIssuerConfig.properties` cell level and server level configuration files. Starting with WebSphere Application Server Version 8, you should use the administrative console or the `setSAMLIssuerConfigInBinding` command task to specify a self-issued SAML token's configuration as custom properties in the requester's outbound configuration in the general bindings or in the application-specific bindings. Do not use server level and cell level `SAMLIssuerConfig.properties` file.

Before you begin

The product provides an alternate way to specify a self-issued SAML token configuration in policy set bindings. Migrate self-issued SAML token configuration data from the `SAMLIssuerConfig.properties` file to the bindings. Specifying configuration data for creating self-issued SAML tokens in general bindings or application-specific bindings provides management flexibility to specify the configuration at a finer grained scope, in addition to the cell level and the server level. For example you can configure a specific SAML token issuer for a particular web service application, for an arbitrary group of applications, or for a web service application in a security domain.

Note: Self-issued SAML token configuration data that is defined in the bindings takes precedence over data that is defined in the server level or the cell level `SAMLIssuerConfig.properties` file, in that order. When a self-issued SAML token configuration data is defined in an attached policy set bindings, the Web services security runtime environment will neglect the `SAMLIssuerConfig.properties` files, both at the server level and at the cell level. So it is important that when you migrate from the `SAMLIssuerConfig.properties` file to the bindings, you must migrate all the required properties.

About this task

Two command tasks are available to manage the `SAMLIssuerConfig.properties` file-based SAML issuer configuration. This file can be located at the cell level and the server level. These two tasks are:

- `listSAMLIssuerConfig`
- `updateSAMLIssuerConfig`

Procedure

1. Run the wsadmin command task in the interactive mode. The following Jython script illustrates how to run the wsadmin command task in the interactive mode.

```
AdminTask.listSAMLIssuerConfig('[-interactive]')
```

To select the server level SAML issuer configuration, the *serverName* and *nodeName* parameters are required. If these parameters are missing, then the command task lists the cell level SAML issuer configuration.

2. Use the `listSAMLIssuerConfig` command task to display the server level SAML issuer configuration.

```
AdminTask.listSAMLIssuerConfig('[-nodeName Node01 -serverName server1]')
```

   You need the “monitor” or above administrative role privilege to execute the `listSAMLIssuerConfig` command.

3. Use the updateSAMLIssuerConfig command task to update the server level or cell level SAML issuer configuration.

```
AdminTask.updateSAMLIssuerConfig('[-IssuerURI My_Issuer
                                  -TimeToLiveMilliseconds 3600000
                                  -KeyStoreRef "name=myKeyStore managementScope=(cell):Node01Cell:(node):Node01"
                                  -KeyAlias samlissuer
                                  -KeyName "CN=SAMLIssuer, O=Acme, C=US" -KeyPassword *****
                                  -TrustStoreRef "name=myKeyStore managementScope=(cell):Node01Cell:(node):Node01 "]')
```

   If the *serverName* and *nodeName* parameters are not specified, then the task updates the cell level SAML issuer configuration.

   You need the “administrator” administrative role privilege to execute the `updateSAMLIssuerConfig` command.

Results

You have created command scripts to automate the process of updating the cell level or the server level `SAMLIssuerConfig.properties` files, or you have created self-issued SAML token configuration data as custom properties in the requester's outbound configuration in the general bindings or in the application-specific bindings.

Example

The following example illustrates how to add or modify self-issued SAML token configuration data in the application-specific bindings:

```
AdminTask.setSAMLIssuerConfigInBinding('[-bindingName SAMLTestAppClientBinding
-bindingLocation [ [application JaxWSServicesSamples] [attachmentId 1904] ]
-com.ibm.wsspi.wssecurity.saml.config.issuer.IssuerURI My_Issuer
-com.ibm.wsspi.wssecurity.saml.config.issuer.TimeToLiveMilliseconds 3600000
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyStoreRef "name=myKeyStore managementScope=(cell):Node01Cell:(node):Node01 "
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyAlias samlissuer
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyName "CN=SAMLIssuer, O=Acme,C=US"
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyPassword *****
-com.ibm.wsspi.wssecurity.saml.config.issuer.TrustStoreRef "name=myKeyStore managementScope=(cell):Node01Cell:(node):Node01 "]')
```

The following example illustrates how to modify the general bindings:

```
AdminTask.setSAMLIssuerConfigInBinding('[-bindingName "Saml Bearer Client sample"
-bindingScope domain -bindingLocation  -domainName global
-com.ibm.wsspi.wssecurity.saml.config.issuer.IssuerURI My_Issuer
-com.ibm.wsspi.wssecurity.saml.config.issuer.TimeToLiveMilliseconds 3600000
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyStorePath "profile_root/etc/ws-security/saml/saml-issuer.jceks
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyStoreType jceks
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyStorePassword *****
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyAlias samlissuer
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyName "CN=SAMLIssuer, O=Acme, C=US"
-com.ibm.wsspi.wssecurity.saml.config.issuer.KeyPassword *****
-com.ibm.wsspi.wssecurity.saml.config.issuer.TrustStorePath "profile_root/profiles/<server_name>/etc/ws-security/saml/saml-issuer.jceks
-com.ibm.wsspi.wssecurity.saml.config.issuer.TrustStoreType jceks
-com.ibm.wsspi.wssecurity.saml.config.issuer.TrustStorePassword *****]')
```

When specifying the application bindings, **bindingLocation** is a required parameter and can be supplied as a properties object. The property names are `application` and `attachmentId`. When specifying the general bindings, **bindingLocation**, which can be null or have empty properties, is required. Additionally, **bindingScope** is required if the scope is not global. Use the **bindingName** parameter to identify the binding location. For more information about **bindingLocation**, **bindingScope**, and **domainName**, refer to the **setBinding** or **getBinding** command tasks documentation.

To remove SAML issuer configuration custom properties from the bindings, use the administrative console or the **setBinding** command task.

Configuring default Web Services Security bindings:

WebSphere Application Server provides support for a set of default Web Services Security bindings for applications. A set of bindings is a named object that is associated with a specific policy set and service resource attached to the policy set.

About this task

Bindings contain environment and platform specific information, such as the following types of information:

- Keys used for signature and encryption
- Keystore information
- Authentication information
- Persistent information

In WebSphere Application Server Version 7.0 and later, there are two types of bindings, application specific bindings and general bindings. Typically, bindings are specific to the application or the platform, and they are not shared.

General bindings can be configured to be used across a range of policy sets and can be reused across applications and for trust service attachments. Though general bindings are highly reusable, they are not able to provide configuration for advanced policy requirements, such as multiple signatures. There are two types of general bindings: general provider policy set bindings and general client policy set bindings. The general bindings that are shipped with WebSphere Application Server are initially set as the default bindings, but you can choose a different binding as the default, or change the level of binding that should be used as the default, for example, from cell level binding to server level binding. Default bindings are used when no application specific binding or trust service binding has been assigned to a policy set attachment. For more information, see the topic General JAX-WS default bindings for Web Services Security. For a description of the general sample bindings that are included with WebSphere Application Server, and used with the JAX-WS programming model, read the topic General sample bindings for JAX-WS applications.

To create general bindings:

Procedure

1. Log in to the administrative console and navigate to the general provider policy set and bindings panel, or the general client policy set and bindings panel
   - Click **Services > Policy sets > General provider policy set bindings**.
   - Click **Services > Policy sets > General client policy set bindings**.
2. Click **New**.

Results

Policy set bindings contain platform-specific information, like keystore, authentication information or persistent information, required by a policy set attachment. Each policy set attachment to a service provider or service client must have exactly one binding. When you create a policy set attachment, the general default bindings are used initially. When general bindings are used in association with a policy set attachment, the cell-level general bindings are applied at run time. If application server level bindings exist, the server-level general bindings override the cell-level definition. General bindings specify configuration for both service client and service provider attachments and the general bindings are not tailored to a specific policy set or application. When you define server-level general bindings, the binding begins in a completely unconfigured state. You must add the policy, and then fully configure the bindings for each added policy.

An application specific binding is a named binding that you create. Application specific bindings enable you to provide platform-specific configuration information for specific policy set attachments. When you create an application specific binding, the available binding configuration options are tailored to the definitions in

the attached policy set. You can reuse application specific bindings for multiple service resources within an application. For example, if you create a trust service specific binding, that binding can be reused only for trust service attachments. When you create an application specific binding for a policy set attachment, the binding begins in a completely unconfigured state. For each policy, such as WS-Security or HTTP Transport, where you want to override the general binding, you must add the policy, and then fully configure the bindings for each added policy.

Important: Only use the sample default bindings in a testing environment. Do not use sample default bindings in a production environment. Default bindings contain sample key files that must be customized before use in a production environment.

See the topic Defining and managing service client or provider bindings for more information about bindings.

General JAX-WS default bindings for Web Services Security:

General bindings are used as the default bindings at the cell level or server level, or for multiple domains, at the domain level. The general bindings that are included with WebSphere Application Server are initially set as the default bindings. However, you can choose a different binding as the default, or change the level of binding that is used as the default, for example, from cell-level binding to server-level binding.

Policy set bindings contain platform-specific information, such as keystore, authentication information or persistent information, required by a policy set attachment. In WebSphere Application Server Version 7.0 and later, there are two types of bindings: application-specific bindings, and general bindings. Both types of bindings are supported for WS-Security policy sets. General bindings can be used as default bindings, and can also be shared across multiple applications and for trust service attachments. There are two types of general bindings: one for service providers and one for service clients. You can define multiple general bindings for the provider and also for the client. However, only one general provider binding and one general client binding can be designated as the default.

Default bindings are used when no application-specific binding or trust service binding has been assigned to a policy set attachment. You can choose the general provider and general client bindings, which are used as the default bindings for the cell. These are the global security settings. Likewise, you can choose the general provider and general client bindings, which are used as the default bindings for a server. For specific information about selecting bindings, see the topic Defining and managing policy set bindings.

In an environment with multiple security domains, you can also choose the general provider and general client bindings, which are used as the default bindings for a domain. If you do not choose a binding to be the default for a server, the default bindings for the domain in which the server resides are used. If you do not choose a binding to be the default for a domain, the default bindings for the cell (global security) are used. You must choose default provider and default client bindings for the cell.

The general bindings that are included with WebSphere Application Server are initially set as the cell default bindings. You cannot delete a binding that has been selected as the default binding for server, a domain, or the cell. Before you delete a binding that is selected as the default, you must select a different default binding, or specify that the defaults for the cell (global security) should be used.

The following default bindings are shipped with the product:

- Provider sample
- Client sample
- Version 6.1 default policy set bindings

  The Version 6.1 bindings are used only if a WebSphere Application Server Version 6.1 Feature Pack for Web Services application is installed within the WebSphere Application Server Version 7.0 and later environment. For more information on these bindings, see the topic Version 6.1 default policy set bindings.

Important: Do not use the provider and client sample bindings that are included with WebSphere Application Server in their current state in a production environment. You must modify these bindings to meet your security needs before using them in a production environment by making a copy of the bindings and then modifying the copy. For example, change the key and keystore settings to ensure security, and modify the binding settings to match your environment.

For a detailed description of the general sample bindings, see the topic General sample bindings for JAX-WS applications.

To define and manage general bindings, in the administrative console click **Services > Policy sets > General provider policy set bindings** or **Services > Policy sets > General client policy set bindings**. To manage bindings for the cell or the domain, click **Services > Policy sets > Default policy set bindings**. The general service provider and client bindings have independent settings that you can customize to meet the needs of your environment. To learn more about general bindings, read the topic Defining and managing policy set bindings.

In addition to choosing default bindings for the cell (global security), you can also choose the general provider and general client bindings that you want to use as the default bindings for a server. When are using the JAX-WS programming model and want to specify the server default bindings, log on to the administrative console and click **Servers > Server Types > WebSphere application servers >** ***server_name***. In the Security section of the console page, click **Default policy set bindings**.

Administering message-level security for JAX-RPC web services

The Java™ API for XML-based RPC (JAX-RPC) specification enables you to develop SOAP-based interoperable and portable web services and web service clients. JAX-RPC simplifies development of web services by shielding you from the underlying complexity of SOAP communication, and enables clients to access a web service as if the web service was a local object mapped into the client's address space.

Securing messages using JAX-RPC at the request and response generators:

You can secure messages with tokens and encryption to protect message integrity, authenticity, and confidentiality.

About this task

To secure messages, you can:

- Configure generator signing to protect message integrity
- Configure encryption to protect message confidentiality at the server or cell level and at the application level
- Configure tokens to protect message authenticity at the server or cell level and at the application level

Procedure

- To configure generator signing to protect message integrity, see the steps outlined in “Configuring generator signing using JAX-RPC to protect message integrity” on page 822.
- To configure encryption to protect message confidentiality at the application level, see the steps outlined in “Configuring encryption using JAX-RPC to protect message confidentiality at the application level” on page 888.
- To configure encryption to protect message confidentiality at the server or cell level, see the steps outlined in “Configuring encryption using JAX-RPC to protect message confidentiality at the server or cell level” on page 912.
- To configure tokens to protect message authenticity at the application level, see the steps outlined in “Configuring token generators using JAX-RPC to protect message authenticity at the application level” on page 857.

- To configure tokens to protect message authenticity at the server or cell level, see the steps outlined in “Configuring token generators using JAX-RPC to protect message authenticity at the server or cell level” on page 916.

Results

By completing the steps in the previous tasks, you have secured messages using tokens and encryption to protect message integrity, authenticity, and confidentiality.

Securing messages using JAX-RPC at the request and response consumers:

You can secure messages at the request and response consumer level to protect message confidentiality and security.

About this task

To secure messages, you can:
- Configure signing to protect message confidentiality
- Configure encryption to protect message confidentiality at the server or cell level and at the application level
- Configure tokens to protect message authenticity at the server or cell level and at the application level

Procedure
- To configure consumer signing to protect message confidentiality, see the steps outlined in “Configuring consumer signing using JAX-RPC to protect message integrity” on page 838
- To configure encryption to protect message confidentiality at the application level, see the steps outlined in “Configuring encryption to protect message confidentiality at the application level” on page 900.
- To configure encryption to protect message confidentiality at the server or cell level, see the steps outlined in “Configuring encryption to protect message confidentiality at the server or cell level” on page 914.
- To configure tokens at the application level to protect message authenticity, see the steps outlined in “Configuring token consumers using JAX-RPC to protect message authenticity at the application level” on page 877.
- To configure tokens at the server or cell level to protect message authenticity, see the steps outlined in “Configuring token consumers using JAX-RPC to protect message authenticity at the server or cell level” on page 929.

Results

By completing the steps in the previous tasks, you have secured messages at the request and response consumer level.

Configuring message-level security for JAX-RPC at the application level:

Modify the application-level configurations in the administrative console.

Configuring generator signing using JAX-RPC to protect message integrity:

You can configure the generator key and signing information at the server or cell and application level to protect message integrity.

About this task

To protect message integrity, you can:

- Configure the signing information for the client-side request generator and the server-side response generator bindings at the server or cell level and at the application level
- Configure the key information for the request generator (client side) and the response generator (server side) bindings on the server or cell level and at the application level

Procedure

- To configure the signing information for the client-side request generator and the server-side response generator bindings at the server or cell, see the steps outlined in "Configuring the signing information using JAX-RPC for the generator binding on the server or cell level" on page 903.
- To configure the signing information for the client-side request generator and the server-side response generator bindings at the application level, see the steps outlined in "Configuring the signing information using JAX-RPC for the generator binding on the application level"
- To configure the key information for the request generator (client side) and the response generator (server side) bindings at the application level, see the steps outlined in "Configuring the key information using JAX-RPC for the generator binding on the application level" on page 844.
- To configure the key information for the generator binding on the server or cell level, see the steps outlined in "Configuring the key information for the generator binding using JAX-RPC on the server or cell level" on page 908.

Results

By completing the steps in these tasks, you have configured generator signing to protect the integrity of messages.

Configuring the signing information using JAX-RPC for the generator binding on the application level:

You can configure the signing information for the client-side request generator and the server-side response generator bindings at the application level.

Before you begin

Note: For WebSphere Application Server version 6.x or earlier only, in the server-side extensions file (`ibm-webservices-ext.xmi`) and the client-side deployment descriptor extensions file (`ibm-webservicesclient-ext.xmi`), you must specify which parts of the message are signed. Also, you must configure the key information that is referenced by the key information references on the signing information panel within the administrative console.

About this task

This task explains the required steps to configure the signing information for the client-side request generator and the server-side response generator bindings at the application level. WebSphere Application Server uses the signing information for the default generator to sign parts of the message including the body, time stamp, and user name token. The Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment. Complete the following steps to configure the signing information for the generator sections of the bindings files on the application level:

Procedure

1. Locate the signing information configuration panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security Properties, you can access the signing information for the request generator and the response generator bindings.

- For the request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
- For the response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.

d. Under Required properties, click **Signing information**.

e. Click **New** to create a signing information configuration, select the box next to the configuration and click **Delete** to delete an existing configuration, or click the name of an existing signing information configuration to edit its settings. If you are creating a new configuration, enter a name in the Signing information name field. For example, you might specify `gen_signinfo`.

2. Select a signature method algorithm from the Signature method field. The algorithm that is specified for the generator, which is either the request generator or the response generator configuration, must match the algorithm that is specified for the consumer, which is either the request consumer or response consumer configuration. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - http://www.w3.org/2000/09/xmldsig#hmac-sha1
   - http://www.w3.org/2000/09/xmldsig#dsa-sha1

   Restriction: Do not use this algorithm if you want the configured application to be compliant with the Basic Security Profile (BSP).

   Any ds:SignatureMethod/@Algorithm element in a SIGNATURE based on a symmetric key must have a value of http://www.w3.org/2000/09/xmldsig#rsa-sha1 or http://www.w3.org/2000/09/xmldsig#hmac-sha1.
3. Select a canonicalization method from the **Canonicalization method** field. The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/10/xml-exc-c14n#
   - http://www.w3.org/2001/10/xml-exc-c14n#WithComments
   - http://www.w3.org/TR/2001/REC-xml-c14n-20010315
   - http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments
4. Select a key information signature type from the Key information signature type field. WebSphere Application Server supports the following signature types:

   None Specifies that the <KeyInfo> element is not signed.

   Keyinfo
   Specifies that the entire <KeyInfo> element is signed.

   Keyinfochildelements
   Specifies that the child elements of the <KeyInfo> element are signed.

   The key information signature type for the generator must match the signature type for the consumer. You might encounter the following situations:
   - If you do not specify one of the previous signature types, WebSphere Application Server uses `keyinfo`, by default.
   - If you select Keyinfo or Keyinfochildelements and you select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm in a subsequent step, WebSphere Application Server also signs the referenced token.
5. Select a signing key information reference from the Signing key information field. This selection is a reference to the signing key that the Application Server uses to generate digital signatures.
6. Click **OK** and **Save** to save the configuration.
7. Click the name of the new signing information configuration. This configuration is the one that you specified in a previous step.

8. Specify the part reference, digest algorithm, and transform algorithm. The part reference specifies which parts of the message to digitally sign.
   a. Under Additional properties, click **Part references** > **New** to create a new part reference, click **Part references** > **Delete** to delete an existing part reference, or click a part name to edit an existing part reference.
   b. Specify a unique part name for this part reference. For example, you might specify `reqint`.
   c. Select a part reference from the Part reference field.

      The part reference refers to the message part that is digitally signed. The part attribute refers to the name of the <Integrity> element in the deployment descriptor when the <PartReference> element is specified for the signature. You can specify multiple <PartReference> elements within the <SigningInfo> element. The <PartReference> element has two child elements when it is specified for the signature: <DigestTransform> and <Transform>.
   d. Select a digest method algorithm from the menu. The digest method algorithm specified within the <DigestMethod> element is used in the <SigningInfo> element.

      WebSphere Application Server supports the following algorithms:
      - http://www.w3.org/2000/09/xmldsig#sha1
      - http://www.w3.org/2001/04/xmlenc#sha256
      - http://www.w3.org/2001/04/xmlenc#sha512
   e. Click **OK** to save the configuration.
   f. Click the name of the new part reference configuration. This configuration is the one that you specified in a previous step.
   g. Under Additional Properties, click **Transforms** > **New** to create a new transform, click **Transforms** > **Delete** to delete a transform, or click a transform name to edit an existing transform. If you create a new transform configuration, specify a unique name. For example, you might specify `reqint_body_transform1`.
   h. Select a transform algorithm from the menu. The transform algorithm is that is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following algorithms:
      - http://www.w3.org/2001/10/xml-exc-c14n#
      - http://www.w3.org/TR/1999/REC-xpath-19991116

        Restriction: Do not use this transform algorithm if you want your configured application to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.
      - http://www.w3.org/2002/06/xmldsig-filter2
      - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
      - http://www.w3.org/2002/07/decrypt#XML
      - http://www.w3.org/2000/09/xmldsig#enveloped-signature

      The transform algorithm that you select for the generator must match the transform algorithm that you select for the consumer.

      Important: If both of the following conditions are true, WebSphere Application Server signs the referenced token:
      - You previously selected the Keyinfo or the Keyinfochildelements option from the Key information signature type field on the signing information panel.
      - You select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm.
9. Click **Apply**.

10. Optional: Determine whether to disable the Inclusive namespace prefix list. The Exclusive XML Canonicalization Version 1.0 specification recommends that you include all of the namespace declarations that correspond to the namespace prefix in the canonicalization form. For security reasons, WebSphere Application Server, by default, includes the prefix in the digital signature for Web Services Security. However, some implementations of Web Services Security cannot handle this prefix list. WebSphere Application Server can handle digitally signed messages that either contain or do not contain the prefix list. If you experience a signature validation failure when a signed Simple Object Access Protocol (SOAP) message is sent and you are using another vendor in your environment, check with your service provider for a possible fix to their implementation before you disable this property. To disable this property, complete the following steps:
    a. Under Additional properties, click **Properties** > **New**.
    b. In the Property name field, enter the `com.ibm.wsspi.wssecurity.dsig.inclusiveNamespaces` property.
    c. In the Property value field, enter the `false` value.
    d. Click **OK**.

    You can set this property for both the request generator and the response generator configurations.
11. Click **Save** at the top of the panel to save your configuration.

Results

After completing these steps, the signing information is configured for the generator on the application level.

What to do next

You must specify a similar signing information configuration for the consumer.

Signing information collection:

Use this page to view a list of signing parameters. Signing information is used to sign and validate parts of a message including the body, time stamp, and user name token. You can also use these parameters for X.509 validation when the authentication method is IDAssertion and the ID type is X509Certificate in the server-level configuration. In such cases, you must fill in the certificate path fields only.

To view this administrative console page on the cell level for signing information, complete the following steps:
1. Click **Security > JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information**.
3. Click **New** to create a signing parameter. Click **Delete** to delete a signing parameter.

To view this administrative console page on the server level for signing information, complete the following steps:
1. Click **Servers > Server Types > WebSphere application Servers > *server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

    Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information**.
4. Click **New** to create a signing parameter. Click **Delete** to delete a signing parameter.

To view this administrative console page on the application level for signing information, complete the following steps:

1. Click **Applications > Application Types > WebSphere enterprise applications >** ***application_name***.
2. Under Modules, click **Manage modules >** ***URI_name***.
3. Under Web Services Security Properties, you can access the signing information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
   - For Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
4. Under Required properties, click **Signing information**.
5. Under Additional properties, you can use this panel to configure the following bindings:
   - For the Request receiver binding, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit**.
   - For the Response receiver binding, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit**.
6. Under Additional properties, click **Signing information**.
7. Click **New** to create a signing parameter. Click **Delete** to delete a signing parameter.

Signing information name:

Specifies the unique name that is assigned to the signing configuration.

Signature method:

Specifies the signature method algorithm that is chosen for the signing configuration.

Canonicalization method:

Specifies the canonicalization method algorithm that is chosen for the signing configuration.

Signing information configuration settings:

Use this page to configure new signing parameters.

The specifications that are listed on this page for the signature method, digest method, and canonicalization method are located in the World Wide Web Consortium (W3C) document entitled, *XML Signature Syntax and Specification: W3C Recommendation 12 Feb 2002*.

To view this administrative console page on the cell level for signing information, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information**.
3. Click **New** to create a signing parameter or click the name of an existing configuration to modify its settings.

To view this administrative console page on the server level for signing information, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information**.
4. Click **New** to create a signing parameter or click the name of an existing configuration to modify its settings.

To view this administrative console page on the application level for signing information, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access the signing information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
   - For Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
4. Under Required properties, click **Signing information**.
5. Under Additional properties, you can access the signing information for the following bindings:
   - For the Request receiver binding, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit**.
   - For the Response receiver binding, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit**.
6. Under Additional properties, click **Signing information**.
7. Click **New** to create a signing parameter or click the name of an existing configuration to modify its settings.

Signing information name:

Specifies the name that is assigned to the signing configuration.

Signature method:

Specifies the algorithm Uniform Resource Identifiers (URI) of the signature method.

The following pre-configured algorithms are supported:

- http://www.w3.org/2000/09/xmldsig#rsa-sha1
- http://www.w3.org/2000/09/xmldsig#dsa-sha1

  Do not use this algorithm if you want the configured application to be compliant with the Basic Security Profile (BSP). Any ds:SignatureMethod/@Algorithm element in a signature based on a symmetric key must have a value of http://www.w3.org/2000/09/xmldsig#rsa-sha1 or http://www.w3.org/2000/09/xmldsig#hmac-sha1.

- http://www.w3.org/2000/09/xmldsig#hmac-sha1

For Version 6.0.*x* applications, you can specify additional signature methods on the Algorithm URI panel. To access the Algorithm URI panel, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Algorithm mappings** > ***algorithm_factory_engine_class_name*** > **Algorithm URI** > **New**.

When you specify the Algorithm URI, you also must specify an algorithm type. To have the algorithm display as a selection in the Signature method field on the Signing information panel, you must select **Signature** as the algorithm type.

This field is available for Version 6.*x* and later applications.

Digest method:

Specifies the algorithm URI of the digest method.

The http://www.w3.org/2000/09/xmldsig#sha1 algorithm is supported.

Canonicalization method:

Specifies the algorithm URI of the canonicalization method.

The following pre-configured algorithms are supported:
- http://www.w3.org/2001/10/xml-exc-c14n#
- http://www.w3.org/2001/10/xml-exc-c14n#WithComments
- http://www.w3.org/TR/2001/REC-xml-c14n-20010315
- http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments

This field is for Version 6.*x* and later applications.

Key information signature type:

Specifies how to sign a KeyInfo element if `dsigkey` or `enckey` is specified for the signing part in the deployment descriptor.

This product supports the following keywords:

keyinfo (default)
: Specifies that the entire KeyInfo element is signed.

keyinfochildelements
: Specifies that the child elements of the KeyInfo element is signed.

If you do not specify a keyword, the application server uses the KeyInfo value, by default.

The Key information signature type field is available for the token consumer binding.

For Version 6.0.*x* applications, this field is also available for the default consumer, request consumer, and response consumer bindings.

Signing key information:

Specifies a reference to the key information that the application server uses to generate the digital signature.

You can specify only one signing key for the default generator, request generator, and response generator bindings on the cell level and the server level. However, you can specify multiple signing keys for the default consumer, request consumer, and response consumer bindings. The signing keys for the default consumer, request consumer, and response consumer bindings are specified using the Key Information references link under Additional properties on the Signing information panel.

On the application level, you can specify only one signing key for the request generator and the response generator. You can specify multiple signing keys for the request consumer and response generator. The signing keys for the request consumer and the response consumer are specified using the Key information references link under Additional properties.

You can specify a signing key configuration for the following bindings on the following levels:

Table 159. Signing key binding information. The key is used for digital signature of messages.

Binding name	Server level, cell level, or application level	Path
Default generator binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under JAX-RPC Default Generator Bindings, click **Key information**.
Default consumer binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under JAX-RPC Default Consumer Bindings, click **Key information**.
Default generator binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under JAX-RPC Default Generator Bindings, click **Key information**.
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under JAX-RPC Default Consumer Bindings, click **Key information**.

Certificate path:

Specifies the settings for the certificate path validation. When you select **Trust any**, this validation is skipped and all incoming certificates are trusted.

The certificate path options are available in token consumer attributes.

Trust anchor

The application server searches for trust anchor configurations on the application and server levels and lists the configurations in this menu.

In a WebSphere Application Server, Network Deployment environment, the application server also searches the cell level for trust anchor configurations.

You can specify trust anchors as an additional property for the response receiver binding and the request receiver binding.

You can specify a trust anchor configuration for the following bindings on the following levels:

Table 160. Trust anchor binding information. The trust anchor is used for signing messages.

Binding name	Server level, cell level, or application level	Path
Default generator binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Trust anchors**.
Default consumer binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Trust anchors**.
Default generator binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Trust anchors > New**.
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Trust anchors** > **New**.
Response receiver	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. 4. Under the Response receiver binding, click **Edit**. 5. Under Additional properties, click **Trust anchors** > **New**.
Request receiver	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. 4. Under the Request receiver binding, click **Edit**. 5. Under Additional properties, click **Trust anchors** > **New**.

For an explanation of the fields on the trust anchor panel, see the help topic Trust anchor configuration settings.

Certificate store

The application server searches for certificate store configurations on the application and server levels and lists the configurations in this menu.

In a WebSphere Application Server, Network Deployment environment, the application server also searches the cell level for certificate store configurations.

You can specify a certificate store configuration for the following bindings on the following levels:

Table 161. Certificate configurations for bindings. The certificate store is used for signing messages.

Binding name	**Server level, cell level, or application level**	**Path**
Default generator binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Collection certificate store** > **New**.
Default consumer binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Collection certificate store** > **New**.
Default generator binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Collection certificate store** > **New**.
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Collection certificate store** > **New**.
Response receiver	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. 4. Under the Response receiver binding, click **Edit**. 5. Under Additional properties, click **Collection certificate store** > **New**.
Request receiver	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. 4. Under the Request receiver binding, click **Edit**. 5. Under Additional properties, click **Collection certificate store** > **New**.

For an explanation of the fields on the collection certificate store panel, see the help topic Collection certificate store configuration settings.

Part reference collection:

Use this page to view the message part references for signature and encryption that are defined in the deployment descriptors.

To view this administrative console page on the cell level for signing information, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.

2. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information** > ***signing_information_name***.
3. Under Additional properties, click **Part references**.

To view this administrative console page on the server level for signing information, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information** > ***signing_information_name***.
4. Under Additional properties, click **Part references**.

To view this administrative console page on the application level for signing information, complete the following steps. Part references are available through the administrative console using Version 6.*x* and later applications only.

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access the signing information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sending) binding, click **Edit custom**.
   - For Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
4. Under Required properties, click **Signing information** > ***signing_information_name***.
5. Under Additional properties, click **Part references**.

Part name:

Specifies the name that is assigned to the part reference configuration.

Part reference name:

Specifies the name of the signed part that is defined in the deployment descriptor.

The Part reference name field is specified in the application binding configuration only.

Digest method algorithm:

Specifies the algorithm Uniform Resource Identifier (URI) of the digest method that is used for the signed part that is specified by the part reference.

Part reference configuration settings:

Use this page to specify a reference to the message parts for signature and encryption that are defined in the deployment descriptors.

To view this administrative console page on the cell level for signing information, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information** > ***signing_information_name***.
3. Under Additional properties, click **Part references**.
4. Click **New** to create a part reference or click the name of an existing configuration to modify its settings.

To view this administrative console page on the server level for signing information, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information** > ***signing_information_name***.
4. Under Additional properties, click **Part references**.
5. Click **New** to create a part reference or click the name of an existing configuration to modify its settings.

To view this administrative console page on the application level for signing information, complete the following steps.

Note: Part references are available through the administrative console using Version 6.*x* applications only.

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** .
2. Click **Manage modules** > ***URI_name*** .
3. Under Web Services Security Properties, you can access the signing information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sending) binding, click **Edit custom**.
   - For Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
4. Under Required properties, click **Signing information** > ***signing_information_name***.
5. Under Additional properties, click **Part references**.
6. Click **New** to create a part reference or click the name of an existing configuration to modify its settings.

You must specify a part name and select a part reference before specifying additional properties. Before specifying the digest method properties that are accessible under Additional properties, specify a digest method algorithm on this panel. If you specify `none` and click **Digest method**, an error message is displayed.

Part name:

Specifies the name that is assigned to the part reference configuration.

Part reference name:

Specifies the name of the <integrity> or <requiredIntegrity> element for the signed part of the message or it specifies the name of the <confidentiality> or <requiredConfidentiality> element for the encrypted part of the message in the deployment descriptor.

The part names that are defined in the deployment descriptor are listed as options in this field. This field is displayed for the binding configuration on the application level only.

Digest method algorithm:

Specifies the algorithm Uniform Resource Identifier (URI) of the digest method that is used for the signed part that is specified by the part reference.

This product provides the following predefined algorithm URIs:
- `http://www.w3.org/2000/09/xmldsig#sha1`
- `http://www.w3.org/2001/04/xmlenc#sha256`
- `http://www.w3.org/2001/04/xmlenc#sha512`

If you want to specify a custom algorithm, you must configure the custom algorithm in the Algorithm URI panel before setting the digest method algorithm.

To access the Algorithm URI panel, complete the following steps for the cell level:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Algorithm mappings** > ***algorithm_factory_engine_class_name*** > **Algorithm URI** > **New**.

The specified algorithms are listed as options for this field.

To access the Algorithm URI panel, complete the following steps for the server level:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Algorithm mappings** > ***algorithm_factory_engine_class_name*** > **Algorithm URI** > **New**.

The specified algorithms are listed as options for this field.

When you specify the Algorithm URI, you also must specify an algorithm type. To have the algorithm display as a selection in the Digest method algorithm field on the Part reference panel, you must select **Digest value calculation (Message digest)** as the algorithm type.

Transforms collection:

Use this page to view the transform algorithm that is used for processing the Web Services Security message.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

To view this administrative console page for the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.

2. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information** > ***signing_information_name***.
3. Under Additional properties, click **Part references** > ***part_name***.
4. Under Additional properties, click **Transforms**.

To view this administrative console page for the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information** > ***signing_information_name***.
4. Under Additional properties, click **Part references** > ***part_name***.
5. Under Additional properties, click **Transforms**.

To view this administrative console page for the application level, complete the following steps.

Note: This option is available for Version 6 and later applications only.
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage Modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access the transforms information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click `Edit custom`.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
4. Under Required properties, click **Signing information** > ***signing_information_name***.
5. Under Additional properties, click **Part references** > ***part_name*****Transforms** .

Transform name:

Specifies the name that is assigned to the transform algorithm.

Transform algorithm:

Specifies the algorithm Uniform Resource Identifier (URI) of the transform algorithm.

Transforms configuration settings:

Use this page to specify the transform algorithm that is used for processing the Web Services Security message.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

To view this administrative console page for the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.

2. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information** > ***signing_information_name***.
3. Under Additional properties, click **Part references** > ***part_name***.
4. Under Additional properties, click **Transforms**.
5. Click **New** to create a transform configuration or click the name of an existing configuration to modify its settings.

To view this administrative console page for the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Signing information** > ***signing_information_name***.
4. Under Additional properties, click **Part references** > ***part_name***.
5. Under Additional properties, click **Transforms**.
6. Click **New** to create a transform configuration or click the name of an existing configuration to modify its settings.

To view this administrative console page for the application level, complete the following steps. This option is available for Version 6.*x* applications only.

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access the transforms information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
4. Under Required properties, click **Signing information** > ***signing_information_name***.
5. Under Additional properties, click **Part references** > ***part_name*** > **Transforms**.
6. Click **New** to create a transform configuration or click the name of an existing configuration to modify its settings.

You must specify a transform name and select a transform algorithm before specifying additional properties.

Transform name:

Specifies the name that is assigned to the transform algorithm.

Transform algorithm:

Specifies the algorithm Uniform Resource Identifier (URI) of the transform algorithm.

This product supports the following algorithms:

http://www.w3.org/2001/10/xml-exc-c14n#
: This algorithm specifies the World Wide Web Consortium (W3C) Exclusive Canonicalization recommendation.

http://www.w3.org/TR/1999/REC-xpath-19991116
: This algorithm specifies the W3C XML path language recommendation. If you specify this algorithm, you must specify the property name and value by clicking **Properties**, which is displayed under Additional properties. For example, you might specify the following information:

Property
: `com.ibm.wsspi.wssecurity.dsig.XPathExpression`

Value `not(ancestor-or-self::*[namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and local-name()='Signature'])`

Note: Do not use this transform algorithm if you want your configured application to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.

http://www.w3.org/2002/06/xmldsig-filter2
: This algorithm specifies the XML-Signature XPath Filter Version 2.0 proposed recommendation.

When you use this algorithm, you must specify a set of properties. You can use multiple property sets for the XPath Filter Version 2. Therefore, it is recommended that your property names end with the number of the property set, which is denoted by an asterisk in the following examples:

- To specify an XPath expression for the XPath filter2, you might use:

```
name com.ibm.wsspi.wssecurity.dsig.XPath2Expression_*
```

- To specify a filter type for each XPath, you might use:

```
name com.ibm.wsspi.wssecurity.dsig.XPath2Filter_*
```

  Following this expression, you can have a value, `[intersect]`, `[subtract]`, or `[union]`.

- To specify the processing order for each XPath, you might use:

```
name com.ibm.wsspi.wssecurity.dsig.XPath2Order_*
```

  Following this expression, indicate the processing order of the XPath.

The following is a list of complete examples:

```
com.ibm.wsspi.wssecurity.dsig.XPath2Expression_2 = [XPath expression#1]
com.ibm.wsspi.wssecurity.dsig.XPath2Filter_1 = [intersect]
com.ibm.wsspi.wssecurity.dsig.XPath2Order_1 = [1]
com.ibm.wsspi.wssecurity.dsig.XPath2Expression_2 = [XPath expression#2]
com.ibm.wsspi.wssecurity.dsig.XPath2Filter_2 = [subtract]
com.ibm.wsspi.wssecurity.dsig.XPath2Order_2 = [2]
```

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
: This algorithm specifies the enhancements to SOAP messaging that provide message integrity and confidentiality.

http://www.w3.org/2002/07/decrypt#XML
: This algorithm specifies the W3C decryption transform for XML Signature recommendation.

http://www.w3.org/2000/09/xmldsig#enveloped-signature
: This algorithm specifies the W3C recommendation for XML digital signatures.

Configuring consumer signing using JAX-RPC to protect message integrity:

You can configure protect message integrity by configuring signing and key information at the server or cell and application level.

Before you begin

About this task

To protect message integrity, you can:

- Configure the signing information for the consumer binding on the application level or at the server or cell level
- Configure the key information for the consumer binding on the application level or at the server or cell level

Procedure

- To configure the signing information for the consumer binding on the application level, see the steps outlined in "Configuring the signing information using JAX-RPC for the consumer binding on the application level"
- To configure the signing information for the consumer binding on the server or cell level, see the steps outlined in "Configuring the signing information using JAX-RPC for the consumer binding on the server or cell level" on page 905
- To configure the key information for the consumer binding on the application level, see the steps outlined in "Configuring the key information for the consumer binding on the application level" on page 855
- To configure the key information for the consumer binding on the server or cell level, see the steps outlined in "Configuring the key information for the consumer binding using JAX-RPC on the server or cell level" on page 911

Results

By completing the steps in these tasks, you have configured the consumer signing to protect the integrity of messages.

Configuring the signing information using JAX-RPC for the consumer binding on the application level:

You can configure the signing information for the server-side request consumer and the client-side response consumer bindings at the application level.

Before you begin

Note: For WebSphere Application Server version 6.x or earlier only, in the server-side extensions file and the client-side deployment descriptor extensions file, you must specify which parts of the message are signed.

About this task

Configure the key information that is referenced by the key information references on the signing information panel within the administrative console. WebSphere Application Server uses the signing information on the consumer side to verify the integrity of the received SOAP message by validating that the message parts are signed. Complete the following steps to configure the signing information for the server-side request consumer and client-side response consumer sections of the bindings files on the application level.

Procedure

1. Access the administrative console.

   To access the administrative console, enter `http://localhost:`*`port_number`*`/ibm/console` in your web browser unless you have changed the port number.

2. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
3. Under Manage modules, click ***URI_name***.
4. Under Web Services Security Properties you can access the signing information for the request generator and response generator bindings.
   - To configure the request consumer signing information, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - To configure the response consumer signing information, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
5. Under Required properties, click **Signing information**.
6. Click **New** to create a signing information configuration, click **Delete** to delete an existing configuration, or click the name of an existing signing information configuration to edit its settings. If you are creating a new configuration, enter a name in the Signing information name field.
7. Select a signature method algorithm from the Signature method field. The signature method is the algorithm that is used to convert the canonicalized <SignedInfo> element in the binding file into the <SignatureValue> element. The algorithm that is specified for the consumer, which is either the request consumer or the response consumer configuration, must match the algorithm specified for the generator, which is either the request generator or response generator configuration. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - http://www.w3.org/2000/09/xmldsig#hmac-sha1
   - http://www.w3.org/2000/09/xmldsig#dsa-sha1

     Do not use this algorithm if you want the configured application to be compliant with the Basic Security Profile (BSP). Any ds:SignatureMethod/@Algorithm element in a signature based on a symmetric key must have a value of http://www.w3.org/2000/09/xmldsig#rsa-sha1 or http://www.w3.org/2000/09/xmldsig#hmac-sha1.
8. Select a canonicalization method from the Canonicalization method field. The canonicalization method algorithm is used to canonicalize the <SignedInfo> element before it is incorporated as part of the digital signature operation. The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/10/xml-exc-c14n#
   - http://www.w3.org/2001/10/xml-exc-c14n#WithComments
   - http://www.w3.org/TR/2001/REC-xml-c14n-20010315
   - http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments
9. Select a key information signature type from the Key information signature type field. The key information signature type specifies how the <KeyInfo> element in the SOAP message is digitally signed. WebSphere Application Server supports the following signature types:

   None Specifies that the key is not signed.

   Keyinfo
   Specifies that the entire KeyInfo element is signed.

   Keyinfochildelements
   Specifies that the child elements of the KeyInfo element are signed.

   If you do not specify one of the previous signature types, WebSphere Application Server uses keyinfo, by default. The key information signature type for the consumer must match the signature type for the generator.
10. Under Additional properties, click **Key information references**.
    a. Click **New** to create a key information reference or click the name of an existing entry to edit its configuration. The Key information references panel is displayed.

b. Enter a name in the Name field.
c. Select a key information reference in the Key information reference field. This reference is the key information configuration name that specifies the key information that is used by this signing information configuration.

11. Return to the Signing information panel. Under Additional properties, click **Part references**. On the Part references panel, you can specify references to the message parts that are defined in the deployment descriptor extensions file.
    a. Click **New** to create a new Part reference or click the name of an existing part reference to edit its configuration. The Part reference panel is displayed.
    b. Enter a name in the Part name field. This name is the name of the required integrity configuration in the deployment descriptor extensions file and specifies the message parts that must be digitally signed.
    c. Select a digest method algorithm from the Digest method algorithm field.

       WebSphere Application Server supports the following pre-configured algorithms:
       - http://www.w3.org/2000/09/xmldsig#sha1
       - http://www.w3.org/2001/04/xmlenc#sha256
       - http://www.w3.org/2001/04/xmlenc#sha512

       If you want to specify a custom algorithm, you must configure the custom algorithm in the Algorithm URI panel before setting the digest method algorithm.
12. Under Additional properties, click **Transforms**.
    a. Click **New** to create a new transform or click the name of an existing transform to edit its configuration.
    b. Enter a name in the Transform name field.
    c. Select a transform algorithm from the Transform algorithm field. WebSphere Application Server supports the following pre-configured algorithms:
       - http://www.w3.org/2001/10/xml-exc-c14n#
       - http://www.w3.org/TR/1999/REC-xpath-19991116

         Do not use this transform algorithm if you want your configured application to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.
       - http://www.w3.org/2002/06/xmldsig-filter2
       - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
       - http://www.w3.org/2002/07/decrypt#XML
       - http://www.w3.org/2000/09/xmldsig#enveloped-signature

       The transform algorithm that you select for the consumer must match the transform algorithm that you select for the generator. For each part reference in the signing information, specify both a digest method algorithm and a transform algorithm.
13. Click **OK**.
14. Click **Save** at the top of the panel to save your configuration.

Results

After completing these steps, you have configured the signing information for the consumer.

What to do next

You must specify a similar signing information configuration for the generator.

Key information references collection:

Use this page to view the key information references that are needed for encryption or signing.

To view this administrative console page on the cell level, complete the following steps. On the cell level, you can configure the key information references for the default consumer bindings only.

1. Click **Security > JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Consumer Bindings, click either of the following links:
   - Click **Encryption information >** ***encryption_information_name***.
   - Click **Signing information >** ***signing_information_name***.
3. Under Additional properties, click **Key information reference**.

To view this administrative console page on the server level, complete the following steps. On the server level, you can configure the key information references for the default consumer bindings only.

1. Click **Servers > Server Types > WebSphere application servers >** *server_name*.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Consumer Bindings, click either of the following links:
   - Click **Encryption information >** ***encryption_information_name***.
   - Click **Signing information >** ***signing_information_name***.
4. Under Additional properties, click **Key information reference**.

To view this administrative console page on the application level, complete the following steps. On the application level, you can configure the key information reference for the consumer bindings only.

1. Click **Applications > Application Types > WebSphere enterprise applications***application_name*.
2. Under Modules, click **Manage modules >** ***URI_name***.
3. Under Web Services Security properties, you can access the signing information for the following bindings:
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Encryption information**. Click **New** to create a new encryption configuration or click the name of a configuration to modify its settings.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Encryption information**. Click **New** to create a new encryption configuration or click the name of a configuration to modify its settings.

Name:

Specifies the name of the Key information reference.

Key information reference:

Specifies a reference to the message parts that are signed or encrypted.

The value of this field is the name of the <requiredIntegrity> or the <requiredConfidentiality> element in the deployment descriptor.

Key information reference configuration settings:

Use this page to specify a reference to the message parts for signature and encryption that is defined in the deployment descriptors.

To view this administrative console page on the cell level for the key information references, complete the following steps. On the cell level, you can configure the key information references for the default consumer bindings only.

1. Click **Security > JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Consumer Bindings, click either of the following links:
   - Click **Encryption information >** ***encryption_information_name***.
   - Click **Signing information >** ***signing_information_name***.
3. Under Additional properties, click **Key information references**.

To view this administrative console page on the server level for the key information references, complete the following steps. On the server level, you can configure the key information references for the default consumer bindings only.

1. Click **Servers > Server Types > WebSphere application servers >** *server_name*.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Consumer Bindings, click either of the following links:
   - Click **Encryption information >** ***encryption_information_name***.
   - Click **Signing information >** ***signing_information_name***.
4. Under Additional properties, click **Key information references**.

To view this administrative console page on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules >** ***URI_name***.
3. Under Web Services Security Properties, you can access the key information references for the following bindings:
   - For the Response consumer (sender) binding, click **Web services: Client security bindings**. Under Response consumer (sender) binding, click **Edit custom**. Under Required properties, click **Encryption information >** ***encryption_information_name***. Under Additional properties, click **Key information references**.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Encryption information >** ***encryption_information_name***. Under Additional properties, click **Key information references**.

Name:

Specifies the name of the key information reference.

Key information reference:

Specifies a reference to the message parts that are signed or encrypted.

The value of this field is the name of the <requiredIntegrity> or the <requiredConfidentiality> element in the deployment descriptor. You can specify a signing key configuration for the following bindings:

Table 162. Key information reference binding configurations. The key is used for signing or encrypting message parts.

Binding name	Server level, Cell level, or application level	Path
Default consumer binding	Cell level	1. Click **Security > JAX-WS and JAX-RPC security runtime**. 2. Under JAX-RPC Default Consumer Bindings, click **Key information**.
Default consumer binding	Server level	1. Click **Servers > Server Types > WebSphere application servers >** *server_name*. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under JAX-RPC Default Consumer Bindings, click **Key information**.
Response consumer (receiver) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** . 2. Under Modules, click **Manage modules > *URI_name***. 3. Under Web Services Security Properties, click **Web services: Client security bindings**. 4. Under Response consumer (receiver) binding, click **Edit custom**. 5. Under Required properties, click **Key information**.
Request consumer (receiver) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules > *URI_name***. 3. Under Web Services Security Properties, click **Web services: Server security bindings**. 4. Under Request consumer (receiver) binding, click **Edit custom**. 5. Under Required properties, click **Key information**.

Configuring the key information using JAX-RPC for the generator binding on the application level:

The key information is used to specify the configuration needed to generate the key for digital signature and encryption. The signing information and the encryption information configurations can share the key information, so they are both defined at the same level.

Before you begin

Before you begin this task, configure the key locators and the token consumers that are referenced by the Key locator reference and Token reference fields within the key information panel.

About this task

This task provides the steps needed for configuring the key information for the request generator (client side) and the response generator (server side) bindings at the application level.

Complete the following information to configure the key information for the generator binding on the application level:

Procedure

1. Locate the key information configuration panel in the administrative console.

a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
b. Under Manage modules, click ***URI_name***.
c. Under Web Services Security Properties you can access the key information for the request generator and response generator bindings.
   - For the request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
   - For the response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
d. Under Required properties, click **Key information**.
e. Click **New** to create a key information configuration, select the box next to an existing configuration and click **Delete** to delete the configuration, or click the name of an existing signing information configuration to edit its settings. If you are creating a new configuration, enter a name in the Key information name field. For example, you might specify `gen_signkeyinfo`.

2. Select a key information type from the Key information type field. The key information type specifies how to reference the security tokens. WebSphere Application Server supports the following key information types:

Key identifier
The security token is referenced using an opaque value that uniquely identifies the token. The algorithm that is used for generating the <KeyIdentifier> element value depends upon the token type. For example, a hash of the important elements of the security token is used for generating the <KeyIdentifier> element value. The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   <wsse:SecurityTokenReference>
      <wsse:KeyIdentifier ValueType="wsse:X509v3">/62wXO...
      </wsse:KeyIdentifier>
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Key name
The security token is referenced using a name that matches an identity assertion within the token. It is recommended that you do not use this key type as it might result in multiple security tokens that match the specified name. The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo>
   <ds:KeyName>CN=Group1</ds:KeyName>
</ds:KeyInfo>
```

Security token reference
The security token is directly referenced using Universal Resource Identifiers (URIs). The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo>
   <wsse:SecurityTokenReference>
      <wsse:Reference URI="#mytoken" />
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Embedded token
The security token is directly embedded within the <SecurityTokenReference> element. The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo>
   <wsse:SecurityTokenReference>
      <wsse:Embedded wsu:Id="tok1" />
         ...
      </wsse:Embedded>
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

X509 issuer name and issuer serial

The security token is referenced by an issuer name and an issuer serial number of an X.509 certificate. The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo>
   <wsse:SecurityTokenReference>
     <ds:X509Data>
        <ds:X509IssuerSerial>
           <ds:X509IssuerName>CN=Jones, O=IBM, C=US
           </ds:X509IssuerName>
           <ds:X509SerialNumber>1040152879
            </ds:X509SerialNumber>
        </ds:X509IssuerSerial>
     </ds:X509Data>
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Each type of key information is described in the Web Services Security: SOAP Message Security 1.0 (WS-Security 2004) OASIS standard, which is located at: http://www.oasis-open.org/home/index.php under Web Services Security.

3. Select a key locator reference from the Key locator reference field. This reference specifies a key locator that WebSphere Application Server uses to locate the keys that are used for digital signature and encryption. Before you can select a key locator, you must have configured a key locator. For more information on configuring a key locator, see the following articles:
   - "Configuring the key locator using JAX-RPC for the generator binding on the application level" on page 943
   - "Configuring the key locator using JAX-RPC for the consumer binding on the application level" on page 950
4. Click **Get keys** to view a list of key name references. After you click **Get keys**, the key names that are defined in the <sig_klocator> element are shown in the key name reference menu. If you change the key locator reference, you must click **Get keys** again to display the list of key names associated with the new key locator.
5. Select a key name reference from the Key name reference field. This reference specifies the name of a key that is used for generating a digital signature and for encryption. The list of key names provided comes from the key locator specified with the key locator reference.
6. Select a token reference from the Token reference field. This token reference specifies the name of token generator that is used for processing the security token. However, WebSphere Application Server requires this field only when you select Security token reference or Embedded token in the Key information type field. Before specifying a token reference, you must configure a token generator. For more information on configuring a token generator, see "Configuring token generators using JAX-RPC to protect message authenticity at the application level" on page 857.
7. Optional: If you select Key identifier as the key information type on this panel, you must specify an encoding method, calculation method, value type namespace URI, and a value type local name.
   a. Select an encoding method from the Encoding method field. The encoding method specifies the encoding format for the key identifier. WebSphere Application Server supports the following encoding methods:
      - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary
      - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#HexBinary

   b. Select a calculation method from the Calculation method field. WebSphere Application Server supports the following calculation methods:
      - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#ITSHA1
      - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#IT60SHA1

   c. Specify a value type namespace Uniform Resource Identifier (URI) in the Namespace URI field. In this field, specify the namespace URI of the value type for a security token that is referenced by

the key identifier. When you specify the X.509 certificate token, you do not need to specify this option. If you want to specify another token, you must specify the URI of the qualified name (QName) for value type.

d. Specify a value type local name. This name is the local name of the value type for a security token that is referenced by the key identifier. When this local name is used in conjunction with the corresponding namespace URI, the information is called the value type qualified name or QName.

   When you specify the X.509 certificate token, it is recommended that you use the predefined local names. When you specify the predefined local names, you do not need to specify the namespace URI of the value type. However, if you do not use one of the predefined local names, you must specify both the uniform resource identifier (URI) and the local name. WebSphere Application Server provides the following predefined local names:

   X.509 certificate token
   http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3

   X.509 certificates in a PKIPath
   http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1

   A list of X509 certificates and CRLs in a PKCS#7
   http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

   LTPA Lightweight Third-Party Authentication token. When you specify a value type local name of LTPA, you must also specify a namespace URI of http://www.ibm.com/websphere/appserver/tokentype/5.0.2.

   LTPA_PROPAGATION
   Lightweight Third-Party Authentication propagation token. When you specify a value type local name of LTPA_PROPAGATION, you must also specify a namespace URI of http://www.ibm.com/websphere/appserver/tokentype.

8. Click **OK** and then click **Save** to save the configuration.

Results

You have configured the key information for the generator binding at the application level

What to do next

You must specify a similar key information configuration for the consumer.

Key information collection:

Use this page to view the configurations that are currently available for generating or consuming the key for XML digital signatures and XML encryption.

To view this administrative console page on the cell level for the key information references, complete the following steps:

1. Click **Security > JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Generator Bindings or the JAX-RPC Default Consumer Bindings, click **Key information**.

To view this administrative console page on the server level for the key information references, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** *server_name*.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

3. Under JAX-RPC Default Generator Bindings or the JAX-RPC Default Consumer Bindings, click **Key information**.

To view this administrative console page on the application level for the key information references, complete the following steps.

Note: This option is available on the application level for Version 6 and later applications.

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules >** ***URI_name***.
3. Under Web Services Security Properties, you can access the signing information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
4. Under Required properties, click **Key information**.

Key information name:

Specifies the name that is given for the key configuration.

Key information class name:

Specifies the class name that is used for the key information type.

Key information type:

Specifies the type of mechanism used to reference the security token. The type corresponds to the class name that is specified in the Key information class name field.

Key information configuration settings:

Use this page to specify the related configuration need to specify the key for XML digital signature or XML encryption.

To view this administrative console page on the cell level for the key information references, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Generator Bindings or the JAX-RPC Default Consumer Bindings, click **Key information**.

To view this administrative console page on the server level for the key information references, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

3. Under JAX-RPC Default Generator Bindings or the JAX-RPC Default Consumer Bindings, click **Key information**.
4. Click **New** to create a new configuration or click the configuration name to modify its contents.

To view this administrative console page on the application level for the key information references, complete the following steps.

Note: This option is available on the application level for Version 6.*x* applications.

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Additional properties, you can access the signing information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
4. Under Required properties, click **Key information**.
5. Click **New** to create a new configuration or click the configuration name to modify its contents.

Before clicking **Properties** under Additional properties, you must enter a value in the **Key information name** field and select an option for the Key information type and Key locator reference options.

Key information name:

Specifies a name for the key information configuration.

Key information type:

Specifies the type of key information. The key information type specifies how to reference security tokens.

This product supports the following types of key information. Each type of key information is described in Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)

Table 163. Key information types. These types of key information are supported by the product.

Type	Description
Key identifier	The security token is referenced using an opaque value that uniquely identifies the token.
Key name	The security token is referenced using a name that matches an identity assertion within the token.
Security token reference	With this type, the security token is directly referenced.
Embedded token	With this type, the security token reference is embedded.
X509 issuer name and issuer serial	With this type, the security token is referenced by an issuer and serial number of an X.509 certificate

The X.509 issuer name and issuer serial is described in Web Services Security: X.509 Certificate Token Profile Version 1.0. The other types are described in Web Services Security: SOAP Message Security 1.0 (WS-Security 2004).

If you select **Key identifier** for the key information type, you can specify values in the following fields on this panel:

- Encoding method
- Calculation method
- Value type namespace URI
- Value type local name

Key locator reference:

Specifies the reference that is used to retrieve the key for digital signature and encryption.

Before specifying a key locator reference, you must configure a key locator. You can specify a signing key configuration for the following bindings:

Table 164. Signing key binding configurations. The key is used during digital signature and encryption.

Binding name	Server level, cell level, or application level	Path
Default generator binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Key locators**. 3. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Default consumer binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Key locators**. 3. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Default generator binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Key locators**. 4. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Request sender binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. Under Request sender binding, click **Edit**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Response receiver binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. Under Response receiver binding, click **Edit**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.

Table 164. Signing key binding configurations (continued). The key is used during digital signature and encryption.

Binding name	Server level, cell level, or application level	Path
Request receiver binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. Under Request receiver binding, click **Edit**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Response sender binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. Under Response sender binding, click **Edit**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Request generator (sender) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Response consumer (receiver) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Request consumer (receiver) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Response generator (sender) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.

Key name reference:

Specifies the name of the key that is used for generating digital signature and encryption.

This field is displayed for the default generator and is also displayed for the request generator and response generator for Version 6.*x* applications.

This field is displayed for the default generator and is also displayed for the request generator and response generator.

Table 165. Key name reference binding configurations. The key is used during digital signature and encryption.

Binding name	Server level, cell level, or application level	Path
Default generator binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime** 2. . 3. Under Additional properties, click **Key locators**. 4. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Default generator binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Key locators**. 4. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Request generator (sender) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.
Response generator (sender) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**. 4. Under Additional properties, click **Key locators**. 5. Click **New** to create a new key locator or click the name of a configured key locator to modify its configuration.

Token reference:

Specifies the name of a token generator or token consumer that is used for processing a security token.

The application server requires this field only when you specify Security token reference or Embedded token in the **Key information type** field. The **Token reference** field is also required when you specify a key identifier type for the consumer. Before specifying a token reference, you must configure a token generator or token consumer. You can specify a token configuration for the following bindings on the following levels:

Table 166. Token reference binding configurations. The reference information is used for a security token reference or an embedded token.

Binding name	Server level, cell level, or application level	Path
Default generator binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under JAX-RPC Default Generator Bindings, click **Token generators**. 3. Click **New** to create a new token generator or click the name of a configured token generator to modify its configuration.

Table 166. Token reference binding configurations (continued). The reference information is used for a security token reference or an embedded token.

Binding name	Server level, cell level, or application level	Path
Default consumer binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under JAX-RPC Default Consumer Bindings, click **Token consumers**. 3. Click **New** to create a new token consumer or click the name of a configured token consumer to modify its configuration.
Default generator binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under JAX-RPC Default Generator Bindings, click **Token generator**. 4. Click **New** to create a new token generator or click the name of a configured token generator to modify its configuration.
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under JAX-RPC Default Consumer Bindings, click **Token consumer**. 4. Click **New** to create a new token consumer or click the name of a configured token consumer to modify its configuration.
Request generator (sender) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**. 4. Under Additional properties, click **Token generators**. 5. Click **New** to create a new token generator or click the name of a configured token generator to modify its configuration.
Response consumer (receiver) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**. 4. Under Required properties, click **Token consumers**. 5. Click **New** to create a new token consumer or click the name of a configured token consumer to modify its configuration.
Request consumer (receiver) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**. 4. Under Required properties, click **Token consumers**. 5. Click **New** to create a new token consumer or click the name of a configured token consumer to modify its configuration.

Table 166. Token reference binding configurations (continued). The reference information is used for a security token reference or an embedded token.

Binding name	Server level, cell level, or application level	Path
Response generator (sender) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**. 4. Under Additional properties, click **Token generators**. 5. Click **New** to create a new token generator or click the name of a configured token generator to modify its configuration.

Encoding method:

Specifies the encoding method that indicates the encoding format for the key identifier.

This field is valid when you specify Key identifier in the Key information type field. This product supports the following encoding methods:
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#HexBinary

This field is available for the default generator binding only.

Calculation method:

This field is valid when you specify Key identifier in the **Key information type** field. This product supports the following calculation methods:
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#ITSHA1
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#IT60SHA1

This field is available for the generator binding only.

Value type namespace URI:

Specifies the namespace Uniform Resource Identifier (URI) of the value type for a security token that is referenced by the key identifier.

This field is valid when you specify Key identifier in the **Key information type** field. When you specify the X.509 certificate token, you do not need to specify this option. If you want to specify another token, specify the URI of `QName` for value type.

This product provides the following predefined value type URIs for the Lightweight Third Party Authentication (LTPA) token:
- http://www.ibm.com/websphere/appserver/tokentype
- http://www.ibm.com/websphere/appserver/tokentype/5.0.2

This field is available for the generator binding only.

Value type local name:

Specifies the local name of the value type for a security token that is referenced by the key identifier.

When this local name is used with the corresponding namespace URI, the information is called the *value type qualified name* or *QName*.

This field is valid when you specify Key identifier in the **Key information type** field. When you specify the X.509 certificate token, it is recommended that you use the predefined local names. When you specify the predefined local names, you do not need to specify the URI of the value type. This product provides the following predefined local names:

X.509 certificate token
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3

X.509 certificates in a PKIPath
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1

A list of X509 certificates and CRLs in a PKCS#7
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

Lightweight Third Party Authentication (LTPA)

LTPA_PROPAGATION

Attention: For LTPA, the value type local name is `LTPA`. If you enter `LTPA` for the local name, you must specify the `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` URI value in the **Value type URI** field as well. For LTPA token propagation, the value type local name is `LTPA_PROPAGATION`. If you enter `LTPA_PROPAGATION` for the local name, you must specify the `http://www.ibm.com/websphere/appserver/tokentype` URI value in the **Value type URI** field as well. For the other predefined value types (User name token, X509 certificate token, X509 certificates in a PKIPath, and a list of X509 certificates and CRLs in a PKCS#7), the value for the **Value type local name** field begins with `http://`. For example, if you are specifying the user name token for the value type, enter `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken` in the **Value type local name** field and then you do not need to enter a value in the value type URI field.

When you specify a custom value type for custom tokens, you can specify the local name and the URI of the quality name (QName) of the value type. For example, you might specify `Custom` for the local name and `http://www.ibm.com/custom` for the URI.

This field is also available for the generator binding only.

Configuring the key information for the consumer binding on the application level:

You can configure the key information for the request consumer (server side) and the response consumer (client side) bindings at the application level.

Before you begin

Configure the key locators and the token consumers that are referenced by the Key locator reference and the Token reference fields within the key information panel.

About this task

This task provides the steps that are needed for configuring the key information for the request consumer (server side) and the response consumer (client side) bindings at the application level. The key information on the consumer side is used for specifying the information about the key, which is used for validating the digital signature in the received message or for decrypting the encrypted parts of the message. Complete the following steps to configure the key information for consumer binding on the application level.

Procedure

1. Locate the key information configuration panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.

c. Under Web Services Security Properties, you can access the key information for the request consumer and response consumer bindings.
   - For the request consumer (receiver) binding, click **Web services: Server security bindings**. Under request consumer (receiver) binding, click **Edit custom**.
   - For the response consumer (receiver) binding, click **Web services: Client security bindings**. Under response consumer (receiver) binding, click **Edit custom**.

d. Under Required properties, click **Key information**.

e. Click one of the following to work with key information configuration:

   New To create a key information configuration. Enter a name in the Key information name field. For example, you might specify `con_signkeyinfo`.

   Delete To delete a configuration (selected in the box next to that configuration).

   an existing key information configuration
   To edit the settings of a key information configuration.

2. Select a key information type from the Key information type field. The key information types specify different mechanisms for referencing security tokens using the <wsse:SecurityTokenReference> element within the <ds:KeyInfo> element. WebSphere Application Server supports the following key information types:

Key identifier
The security token is referenced using an opaque value that uniquely identifies the token. The algorithm that is used for generating the <KeyIdentifier> element value depends upon the token type. For example, you can use the identifier for the public keys that are defined in the Internet Engineering Task Force (IETF) Request for Comment (RFC) 3280. The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   <wsse:SecurityTokenReference>
      <wsse:KeyIdentifier ValueType="http://docs.oasis-open.org/wss/2004/01
      /oasis-200401-wss-x509-token-profile-1.0#X509v3SubjectKeyIdentifier">
      /62wXO...
      </wsse:KeyIdentifier>
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Key name
The security token is referenced using a name that matches an identity assertion within the token. It is recommended that you do not use this key type as it might result in multiple security tokens that match the specified name. The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo>
   <ds:KeyName>CN=Group1</ds:KeyName>
</ds:KeyInfo>
```

In general, use a key name when you use a Key-Hashing Message Authentication Code (HMAC) digital signature algorithm, such as http://www.w3.org/2000/09/xmldsig#hmac-sha1.

Security token reference
The security token is directly referenced using Universal Resource Identifiers (URIs). The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo>
   <wsse:SecurityTokenReference>
      <wsse:Reference URI='#SomeCert'
       ValueType="http://docs.oasis-open.org/wss/2004/01/
          oasis-200401-wss-x509-token-profile-1.0#X509v3" />
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Attention: As stated in the Web Services Interoperability Organization (WS-I) Basic Security Profile Version 1 draft and shown in the previous example, the wsse:Reference element in a SECURE_ENVELOPE must have a ValueType attribute.

Embedded token

The security token is directly embedded within the <SecurityTokenReference> element. The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo>
   <wsse:SecurityTokenReference>
      <wsse:Embedded wsu:Id="tok1" />
         ...
      </wsse:Embedded>
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

X509 issuer name and issuer serial

The security token is referenced by an issuer name and an issuer serial number of an X.509 certificate. The following <KeyInfo> element is generated in the SOAP message for this key information type:

```
<ds:KeyInfo>
   <wsse:SecurityTokenReference>
     <ds:X509Data>
        <ds:X509IssuerSerial>
           <ds:X509IssuerName>CN=Jones, O=IBM, C=US</ds:X509IssuerName>
           <ds:X509SerialNumber>1040152879</ds:X509SerialNumber>
        </ds:X509IssuerSerial>
     </ds:X509Data>
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Each type of key information is described in the Web Services Security: SOAP Message Security 1.0 (WS-Security 2004) OASIS standard, which is located at: http://www.oasis-open.org/home/index.php under Web Services Security.

3. Select a key locator reference from the Key locator reference field. The value of this field is a reference to a key locator that WebSphere Application Server uses to locate the keys that are used for digital signature and encryption. Before you can select a key locator, you must configure a key locator. For more information on configuring a key locator, see “Configuring the key locator using JAX-RPC for the consumer binding on the application level” on page 950.
4. Select a token reference from the Token reference field. The token reference specifies a reference to a token consumer that is used for processing the security token in the message. However, WebSphere Application Server requires this field only when you select Security token reference or Embedded token in the Key information type field. Before specifying a token reference, you must configure a token consumer. For more information on configuring a token consumer, see “Configuring token consumers using JAX-RPC to protect message authenticity at the application level” on page 877.

   Select **(none)** if a token consumer is not required for this key information configuration.
5. Click **OK** and **Save** to save this configuration.

Results

You have configured the key information for the request or response (or both) consumer binding at the application level.

What to do next

If you have not configured the key information for the generator binding, you must specify a similar key information configuration for the generator. After you configure the key information for both the consumer and the generator, configure the signing information or encryption information, which references the key information that is specified in this key information task.

Configuring token generators using JAX-RPC to protect message authenticity at the application level:

When you specify the token generators at the application level, the information is used on the generator side to generate the security token.

Before you begin

You need to understand that the keystore/alias information that you provide for the generator, and the keystore/alias information that you provide for the consumer are used for different purposes. The main difference applies to the Alias for an X.509 callback handler.

When used in association with an encryption generator, the alias supplied for the generator is used to retrieve the public key to encrypt the message. A password is not required. The alias that is entered on a callback handler associated with an encryption generator must be accessible without a password. This means that the alias must not have private key information associated with it in the keystore. When used in association with a signature generator, the alias supplied for the generator is used retrieve the private key to sign the message. A password is required.

About this task

Complete the following steps to configure the token generator on the application level:

Procedure

1. Locate the token generator panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security Properties you can access the token generators for the following bindings:
      - For the request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
      - For the response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   d. Under Additional properties, click **Token generators**.
   e. Click **New** to create a token generator configuration, select an existing configuration. Click **Delete** to delete an existing configuration, or click the name of an existing token generator configuration to edit its settings. If you are creating a new configuration, enter a unique name in the **Token generator name** field. For example, you might specify `gen_signtgen`.
2. Specify a class name in the **Token generator class name** field. The token generator class must implement the com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent interface. The token generator class name for the request generator and the response generator must be similar to the token consumer class name for the request consumer and the response consumer. For example, if your application requires a username token consumer, you can specify the com.ibm.wsspi.wssecurity.token.UsernameTokenConsumer class name on the token consumer panel for the application level and the com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator class name in this field.
3. Optional: Select a part reference in the **Part reference** field. The part reference indicates the name of the security token that is defined in the deployment descriptor.

   Important: On the application level, if you do not specify a security token in your deployment descriptor, the **Part reference** field is not displayed. If you define a security token called `user_tgen` in your deployment descriptor, `user_tgen` is displayed as an option in the **Part reference** field. You can specify a security token in the deployment descriptor when you assemble your application using an assembly tool.
4. Select either **None** or **Dedicated signing information** for the certificate path. Select **None** when the token generator does not use the PKCS#7 token type. When the token generator uses the PKCS#7 token type and you want to package certificate revocation lists (CRLs) in the security token, select

Dedicated signing information and select a certificate store. To configure a collection certificate store and certificate revocation lists for the generator bindings on the application level, complete the following steps:

a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
b. Under Related Items, click **EJB Modules** or **Web Modules** > ***URI_name***.
c. Under Additional Properties you can access the collection certificate store configuration for the following bindings:
   - For the request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
   - For the response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
d. Under Additional properties, click **Collection certificate store**.

also see the information about configuring a collection certificate store.

5. Optional: Select the **Add nonce** option. This option indicates whether a nonce is included in the user name token for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens. The **Add nonce** option is valid only when the generated token type is a user name token and is available only for the request generator binding.

   If you select the **Add nonce** option, you can specify the following properties under Additional properties. These properties are used by the request consumer.

Table 167. Additional nonce properties. Use the nonce properties to include nonce in the user name token.

Property name	Default value	Explanation
com.ibm.ws.wssecurity.config.token. BasicAuth.Nonce.cacheTimeout	600 seconds	Specifies the timeout value, in seconds, for the nonce value that is cached on the server.
com.ibm.ws.wssecurity.config.token. BasicAuth.Nonce.clockSkew	0 seconds	Specifies the time, in seconds, before the nonce time stamp expires.
com.ibm.ws.wssecurity.config.token. BasicAuth.Nonce.maxAge	300 seconds	Specifies the clock skew value, in seconds, to consider when WebSphere Application Server checks the timeliness of the message.

   On the cell and server levels, you can specify these additional properties for a nonce on the Default bindings for Web Services Security panel within the administrative console.
   - For the cell level, click **Security** > **Web services**.
   - For the server level, click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. Under Security, click **JAX-WS and JAX-RPC security runtime**.

     Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

6. Optional: Select the **Add timestamp** option. This option indicates whether to insert a time stamp into the user name token. The **Add timestamp** option is valid only when the generated token type is a user name token and is available only for the request generator binding.
7. Specify the value type local name in the **Local name** field. For a user name token and an X.509 certificate security token, WebSphere Application Server provides predefined local names for the value type. When you specify any of the following local names, you do not need to specify a value type URI:

   http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
   This local name specifies a user name token.

   http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3
   This local name specifies an X.509 certificate token.

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1
This local name specifies X.509 certificates in a public key infrastructure (PKI) path.

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7
This local name specifies a list of X.509 certificates and certificate revocation lists in a PKCS#7 format.

For an LTPA token, you can use `LTPA` for the value type local name and `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` for the value type Uniform Resource Identifier (URI). For LTPA token propagation, you can use `LTPA_PROPAGATION` for the value type local name and `http://www.ibm.com/websphere/appserver/tokentype` for the value type URI.

8. Optional: Specify the value type URI in the **URI** field. This entry specifies the namespace URI of the value type for the generated token.
9. Click **OK** and **Save** to save the configuration.
10. Click the name of your token generator configuration.
11. Under Additional properties, click **Callback handler**.
12. Specify the settings for the callback handler.
    a. Specify a class name in the **Callback handler class name** field. This class name is the name of the callback handler implementation class that is used to plug-in a security token framework. The specified callback handler class must implement the javax.security.auth.callback.CallbackHandler interface and must provide a constructor using the following syntax:

```
MyCallbackHandler(String username, char[] password, java.util.Map properties)
```

Where:

username
Specifies the user name that is passed into the configuration.

password
Specifies the password that is passed into the configuration.

properties
Specifies the other configuration properties that are passed into the configuration.

This constructor is required if the callback handler needs a user name and a password. However, if the callback handler does not need a user name and a password, such as X509CallbackHandler, use a constructor with the following syntax:

```
MyCallbackHandler(java.util.Map properties)
```

WebSphere Application Server provides the following default callback handler implementations:

com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
This callback handler uses a login prompt to gather the user name and password information. However, if you specify the user name and password on this panel, a prompt is not displayed and WebSphere Application Server returns the user name and password to the token generator. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only. If you use this implementation, you must provide a basic authentication user ID and password on this panel.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This callback handler does not issue a prompt and returns the user name and password if it is specified on this panel. You can use this callback handler when the web service is acting as a client. If you use this implementation, you must provide a basic authentication user ID and password on this panel.

com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
This callback handler uses a standard-in prompt to gather the user name and password. However, if the user name and password is specified on this panel, WebSphere Application Server does not issue a prompt, but returns the user name and password to the token generator. Use this implementation for a Java Platform, Enterprise Edition (Java

EE) application client only. If you use this implementation, you must provide a basic authentication user ID and password on this panel.

com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler
This callback handler is used to obtain the Lightweight Third Party Authentication (LTPA) security token from the Run As invocation Subject. This token is inserted in the Web Services Security header within the SOAP message as a binary security token. However, if the user name and password are specified on this panel, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the Run As Subject. Use this callback handler only when the web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client. If you use this implementation, you must provide a basic authentication user ID and password on this panel.

com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler
This callback handler is used to create the X.509 certificate that is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore and a key definition is required for this callback handler. If you use this implementation, you must provide a key store password, path, and type on this panel.

com.ibm.wsspi.wssecurity.auth.callback.PKCS7CallbackHandler
This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web Services Security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You can specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. If you use this implementation, you must provide a key store password, path, and type on this panel.

com.ibm.wsspi.wssecurity.auth.callback.PkiPathCallbackHandler
This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used. If you use this implementation, you must provide a key store password, path, and type on this panel.

The callback handler implementation obtains the required security token and passes it to the token generator. The token generator inserts the security token in the Web Services Security header within the SOAP message. Also, the token generator is a plug-in point for the pluggable security token framework. Service providers can provide their own implementation, but the implementation must use the com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent interface.

b. Optional: Select the **Use identity assertion** option. Select this option if you have identity assertion defined in the IBM extended deployment descriptor. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For example, WebSphere Application Server sends only the user name of the original caller for a username token generator. For an X.509 token generator, the application server sends the original signer certification only.

c. Optional: Select the **Use RunAs identity** option. Select this option if you have identity assertion defined in the IBM extended deployment descriptor and you want to use the Run As identity instead of the initial caller identity for identity assertion in a downstream call. This option is valid only if you have configured Username TokenGenerator as a token generator.

d. Optional: Specify the basic authentication user ID in the **Basic authentication user ID** field. This entry specifies the user name that is passed to the constructors of the callback handler implementation. The basic authentication user name and password are used if you specified one of the following default callback handler implementations in the **Callback handler class name** field:

- com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler

- com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler

e. Optional: Specify the basic authentication password in the **Basic authentication password** field. This entry specifies the password that is passed to the constructors of the callback handler implementation.

f. Optional: Specify the key store password in the **Key store password** field. This entry specifies the password used to access the key store file. The key store and its configuration are used if you select on of the following default callback handler implementations that are provided by WebSphere Application Server:

com.ibm.wsspi.wssecurity.auth.callback.PKCS7CallbackHandler
The keystore is used to build the X.509 certificate with the certificate path.

com.ibm.wsspi.wssecurity.auth.callback.PkiPathCallbackHandler
The keystore is used to build the X.509 certificate with the certificate path.

com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler
The keystore is used to retrieve the X.509 certificate.

g. Optional: Specify the key store path in the **Path** field. It is recommended that you use the `${USER_INSTALL_ROOT}` in the path name as this variable expands to the WebSphere Application Server path on your machine. To change the path used by this variable, click **Environment** > **WebSphere variables**, and click **USER_INSTALL_ROOT**. This field is required when you use the `com.ibm.wsspi.wssecurity.auth.callback.PKCS7CallbackHandler`, `com.ibm.wsspi.wssecurity.auth.callback.PkiPathCallbackHandler`, or `com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler` callback handler implementations.

h. Optional: Select the key store type in the **Type** field. This selection indicates the format used by the keystore file. You can select one of the following values for this field:

JKS Use this option if the keystore uses the Java Keystore (JKS) format.

JCEKS
Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in WebSphere Application Server. This option provides stronger protection for stored private keys by using Triple DES encryption.

JCERACFKS
Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

PKCS11KS (PKCS11)
Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

PKCS12KS (PKCS12)
Use this option if your keystore uses the PKCS#12 file format.

13. Click **OK** and then click **Save** to save the configuration.
14. Click the name of your token generator configuration.
15. Under Additional properties, click **Callback handler** > **Keys**.
16. Specify the key name, key alias, and the key password.

a. Click **New** to create a key configuration, click **Delete** to delete an existing configuration, or click the name of an existing key configuration to edit its settings. If you are creating a new configuration, enter a unique name in the **Key name** field. For digital signatures, the key name is used by the request generator or response generator signing information to determine which key

is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption. The key name must be a fully qualified, distinguished name. For example, `CN=Bob,O=IBM,C=US`.

   b. Specify the key alias in the **Key alias** field. The key alias is used by the key locator to find the key within the keystore file.
   c. Specify the key password in the **Key password** field. This password is needed to access the key object within the keystore file.

17. Click **OK** and **Save** to save the configuration.

Results

You have configured the token generator for the application level.

What to do next

You must specify a similar token consumer configuration for the application level.

Request generator (sender) binding configuration settings:

Use this page to specify the binding configuration for the request generator.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** .
2. Under Modules, click **Manage modules**.
3. Click the Uniform Resource Identifier (URI).
4. Under Web Services Security Properties, click **Web services: Client security bindings**.
5. Under Request generator (sender) binding, click **Edit custom**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

The security constraints or bindings are defined using the application assembly process before the application is installed.

An assembly tool is not available on the z/OS platform.

If the security constraints are defined in the application, you must either define the corresponding binding information or select the Use defaults option on this panel and use the default binding information for the cell or server level. The default binding provided by this product is a sample. Do not use this sample in a production environment without modifying the configuration. The security constraints define what is signed or encrypted in the Web Services Security message. The bindings define how to enforce the requirements.

Digital signature security constraint (integrity)

The following table shows the required and optional binding information when the digital signature security constraint (integrity) is defined in the deployment descriptor.

Table 168. Binding information for digital signature security constraints. The binding information is used for digitally signing messages.

Information type	Required or optional
Signing information	Required
Key information	Required

Table 168. Binding information for digital signature security constraints (continued). The binding information is used for digitally signing messages.

Information type	Required or optional
Key locators	Optional
Collection certificate store	Optional
Token generator	Optional
Properties	Optional

You can use the key locators and the collection certificate store that are defined at either the cell or the server-level.

Encryption constraint (confidentiality)

The following table shows the required and optional binding information when the encryption constraint (confidentiality) is defined in the deployment descriptor.

Table 169. Binding information for encryption constraints. The binding information is used for encrypting messages.

Information type	Required or optional
Encryption information	Required
Key information	Required
Key locators	Optional
Collection certificate store	Optional
Token generator	Optional
Properties	Optional

You can use the key locators and the collection certificate store that are defined at either the cell or the server-level.

Security token constraint

The following table shows the required and optional binding information when the security token constraint is defined in the deployment descriptor.

Table 170. Binding information for security token constraints. The binding information is used for signing or encrypting messages.

Information type	Required or optional
Token generator	Required
Collection certificate store	Optional
Properties	Optional

You can use the collection certificate store that is defined at either the cell-level or the server-level.

Use defaults:

Select this option if you want to use the default binding information from either the cell or the server-level.

If you select this option, the application server checks for binding information on the server level. If the binding information does not exist on the server level, the application server checks the cell level.

Component:

Specifies the enterprise bean in an assembled EJB module.

Port:

Specifies the port in the web service that is defined during application assembly.

Web service:

Specifies the name of the web service that is defined during application assembly.

Response generator (sender) binding configuration settings:

Use this page to specify the binding configuration for the response generator or response sender.

To view this administrative console page, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules**.
3. Click the Uniform Resource Identifier (URI).
4. Under Web Services Security Properties, click **Web services: Server security bindings**.
5. Under Response generator (sender) binding, click **Edit custom**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

The security constraints or bindings are defined using the application assembly process before the application is installed.

An assembly tool is not available on the z/OS platform.

If the security constraints are defined in the application, you must either define the corresponding binding information or select the **Use defaults** option on this panel and use the default binding information for the cell level or the server-level. The default binding that is provided by this product is a sample. Do not use this sample in a production environment without modifying the configuration. The security constraints define what is signed or encrypted in the Web Services Security message. The bindings define how to enforce the requirements.

Digital signature security constraint (integrity)

The following table shows the required and optional binding information when the digital signature security constraint (integrity) is defined in the deployment descriptor.

Table 171. Binding information for digital signature constraints. The binding information is used for digitally signing messages.

Information type	Required or optional
Signing information	Required
Key information	Required
Key locators	Optional
Collection certificate store	Optional
Token generator	Optional
Properties	Optional

You can use the key locators and the collection certificate store that are defined at either the cell level or the server-level.

Encryption constraint (confidentiality)

The following table shows the required and optional binding information when the encryption constraint (confidentiality) is defined in the deployment descriptor.

Table 172. Binding information for encryption constraints. The binding information is used for encrypting messages.

Information type	Required or optional
Encryption information	Required
Key information	Required
Key locators	Optional
Collection certificate store	Optional
Token generator	Optional
Properties	Optional

You can use the key locators and the collection certificate store that are defined at either the cell level or the server-level.

Security token constraint

The following table shows the required and optional binding information when the security token constraint is defined in the deployment descriptor.

Table 173. Binding information for security token constraints. The binding information is used for signing and encrypting messages.

Information type	Required or optional
Token generator	Required
Collection certificate store	Optional
Properties	Optional

You can use the collection certificate store that is defined at either the cell level or the server-level.

Use defaults:

Select this option if you want to use the default binding information from the cell or server level.

If you select this option, the application server checks for binding information on the server level. If the binding information does not exist on the server level, the application server checks the cell level.

Port:

Specifies the port number in the web service that is defined during application assembly.

Web service:

Specifies the name of the web service that is defined during application assembly.

Callback handler configuration settings for JAX-RPC:

Use this page to specify how to acquire the security token that is inserted in the Web Services Security header for JAX-RPC within the SOAP message. The token acquisition is a pluggable framework that leverages the Java Authentication and Authorization Service (JAAS) `javax.security.auth.callback.CallbackHandler` interface for acquiring the security token.

gotcha: Before you specify values for the **Keystore** and **Key** properties on this page, you must understand that the keystore/alias information that you provide for the generator, and the

keystore/alias information that you provide for the consumer are used for different purposes. The main difference applies to the alias for an X.509 callback handler:

Generator
When used in association with an encryption generator, the alias supplied for the generator is used to retrieve the public key to encrypt the message. A password is not required. The alias that is entered on a callback handler associated with an encryption generator must be accessible without a password. This means that the alias must not have private key information associated with it in the keystore. When used in association with a signature generator, the alias supplied for the generator is used retrieve the private key to sign the message. A password is required.

Consumer
When used in association with an encryption consumer, the alias supplied for the consumer is used retrieve the private key to decrypt the message. A password is required.

When used in associated with a signature consumer, the alias supplied for the consumer is used strictly to retrieve the public key that is used to resolve an X.509 certificate that is not passed in the SOAP security header as a BinarySecurityToken. A password is not required.

The alias that is entered on a callback handler associated with an signature consumer must be accessible without a password. This means that the alias must not have private key information associated with it in the keystore.

When an X.509 certificate that is not passed in the SOAP security header as a BinarySecurityToken, a SecurityTokenReference will appear in the **`KeyInfo`** element within the **`Signature`** element in the SOAP security header that will be used to resolve the X.509 certificate. The methods that can be used are **`Key identifier`**, **`X.509 issuer name and issuer serial`**, and **`Thumbprint`**. The consumer will accept any of these three methods for resolving an X.509 certificate outside the message when a keystore/alias is configured for an X.509 token consumer associated with a signature consumer.

Because only one alias can be configured on the X.509 token consumer, the WS-Security run time can resolve only one certificate outside a message. For example, if the X.509 token consumer is configured for certificate A, if client A sends the keyIdentifier for certificate A, the certificate can be retrieved. However, if client B sends the keyIdentifier for certificate B, the certificate cannot be retrieved and the message will be rejected.

When an X.509 certificate is sent in the SOAP security header as a BinarySecurityToken, if there is a keystore/alias configured on the X.509 token consumer associated with a signature consumer, the certificate that is configured on the consumer will be compared against the one that is passed in the message. If they do not match, the message will be rejected. This behavior is different than JAX-RPC. The certificate associated with the alias configured on the X.509 token consumer is not used to evaluate trust on the inbound certificate. Only the trust store and cert stores are used for that purpose.

If you want the certificate configured on the X.509 token consumer associated with a signature consumer to be available for KeyInfo resolution, but not reject X.509 certificates that are passed in the message that do not match, you can add the following custom property to the X.509 token consumer callback handler:

```
com.ibm.wsspi.wssecurity.consumer.callbackHandlerKeystoreLimitsAccess=false
```

To view this administrative console page for the callback handler on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default generator bindings, click **Token generators** > ***token_generator_name***.
3. Under Additional properties, click **Callback handler**.

To view this administrative console page for the callback handler on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** .
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default generator bindings, click **Token generators >** ***token_generator_name***.
4. Under Additional properties, click **Callback handler**.

To view this administrative console page for the callback handler on the application level , complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage Modules*****URI_name***.
3. Under Web Services Security properties, you can access the callback handler information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**. Under Additional properties, click **Token generator**. Click **New** to create a new token generator configuration or click the name of an existing configuration to modify its settings. Under Additional properties, click **Callback handler**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**. Under Additional properties, click **Token generator**. Click **New** to create a new token generator configuration or click the name of an existing configuration to modify its settings. Under Additional properties, click **Callback handler**.

Callback handler class name:

Specifies the name of the callback handler implementation class that is used to plug in a security token framework.

The specified callback handler class must implement the `javax.security.auth.callback.CallbackHandler` class. The implementation of the JAAS `javax.security.auth.callback.CallbackHandler` interface must provide a constructor using the following syntax:

```
MyCallbackHandler(String username, char[] password,
    java.util.Map properties)
```

Where:

username
: Specifies the user name that is passed into the configuration.

password
: Specifies the password that is passed into the configuration.

properties
: Specifies the other configuration properties that are passed into the configuration.

The application server provides the following default callback handler implementations:

com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
: This callback handler uses a login prompt to gather user name and password information. However, if you specify the user name and password on this panel, a prompt is not displayed and the application server returns the user name and password to the token generator if it is specified on this panel. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This callback handler does not issue a prompt and returns the user name and password if it is specified on this panel. You can use this callback handler when the web service is acting as a client.

com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
This callback handler uses a standard-in prompt to gather the user name and password. However, if the user name and password is specified on this panel, the application server does not issue a prompt, but returns the user name and password to the token generator. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
This callback handler uses a standard-in prompt to gather the user name and password. However, if the user name and password is specified on this panel, the application server does not issue a prompt, but returns the user name and password to the token generator. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler
This callback handler is used to obtain the Lightweight Third Party Authentication (LTPA) security token from the RunAs invocation Subject. This token is inserted in the Web Services Security header within the SOAP message as a binary security token. However, if the user name and password are specified on this panel, the application server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the RunAs Subject. Use this callback handler only when the web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client.

com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler
This callback handler is used to create the X.509 certificate that is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore and a key definition is required for this callback handler.

com.ibm.wsspi.wssecurity.auth.callback.PKCS7CallbackHandler
This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web Services Security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You must specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format.

com.ibm.wsspi.wssecurity.auth.callback.PkiPathCallbackHandler
This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used.

The callback handler implementation obtains the required security token and passes it to the token generator. The token generator inserts the security token in the Web Services Security header within the SOAP message. Also, the token generator is the plug-in point for the pluggable security token framework. Service providers can provide their own implementation, but the implementation must use the `com.ibm.websphere.wssecurity.wssapi.token.SecurityToken` interface. The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side and to validate (authenticate) the security token on the consumer side, respectively.

Use identity assertion:

Select this option if you have identity assertion defined in the IBM extended deployment descriptor.

This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For example, the application server sends only the

user name of the original caller for a Username TokenGenerator. For an X.509 token generator, the application server sends the original signer certification only.

Use RunAs identity:

Select this option if you have identity assertion defined in the IBM extended deployment descriptor and you want to use the Run As identity instead of the initial caller identity for identity assertion for a downstream call.

This option is valid only if you have Username TokenGenerator configured as a token generator.

Basic authentication user ID:

Specifies the user name that is passed to the constructors of the callback handler implementation.

The basic authentication user name and password are used if you select one of the following default callback handler implementations provided by this product:

- `com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler`
- `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler`
- `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`
- `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`

These implementations are described in detail under the **Callback handler class name** field description in this article.

Basic authentication password:

Specifies the password that is passed to the constructor of the callback handler.

The keystore and its related configuration are used if you select one of the following default callback handler implementations provided by this product:

com.ibm.wsspi.wssecurity.auth.callback.PKCS7CallbackHandler
The keystore is used to build the X.509 certificate with the certificate path.

com.ibm.wsspi.wssecurity.auth.callback.PkiPathCallbackHandler
The keystore is used to build the X.509 certificate with the certificate path.

com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler
The keystore is used to retrieve the X.509 certificate.

Keystore: Select **None** if no keystore is needed for this configuration.

Select **Predefined keystore** to choose predefined keystores with keystore configuration name.

Select **User-defined keystore** to use user-defined keystores.

The following information needs to be specified:

Key store configuration name:

Specifies the name of the key store configuration defined in the keystore settings in secure communications.

Key store password:

Specifies the password that is used to access the keystore file.

Key store path:

Specifies the location of the keystore file.

Use ${*USER_INSTALL_ROOT*} in the path name because this variable expands to the product path on your machine. To change the path used by this variable, click **Environment** > **WebSphere variables** and click **USER_INSTALL_ROOT**.

Key store type:

Specifies the type of keystore file format

Choose one of the following values for this field:

JKS Use this option if the keystore uses the Java Keystore (JKS) format.

JCEKS
Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in the application server. This option provides stronger protection for stored private keys by using Triple DES encryption.

JCERACFKS
Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

PKCS11KS (PKCS11)
Use this option if your keystore file uses the PKCS#11 file format. Keystore files that use this format might contain Rivest Shamir Adleman (RSA) keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

PKCS12KS (PKCS12)
Use this option if your keystore file uses the PKCS#12 file format.

Key collection:

Use this page to view a list of logical names that is mapped to a key alias in the keystore file.

To view this administrative console panel for the key collection on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default generator bindings, click **Token Generators** > ***token_generator_name***.
3. Under Additional properties, click **Callback handler** > **Keys**.

Keys are also available from the JAX-WS and JAX-RPC security runtime panel by clicking **Key locators** > ***key_locator_name***. Under Additional properties, click **Keys**.

To view this administrative console page for the key locator collection on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** .
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default generator bindings, click **Token Generators** > ***token_generator_name*** .
4. Under Additional properties, click **Callback handler** > **Keys**.

Keys are also available from the JAX-WS and JAX-RPC security runtime panel by clicking **Key locators** > ***key_locator_name***. Under Additional properties, click **Keys**.

To use this administrative console page for the key locator collection on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access key locators for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Key locators**. Under Additional properties, click **Keys**.
4. Under Additional properties, you can access key locators for the following bindings:
   - For the Request sender, click **Web services: Client security bindings**. Under Request sender binding, click **Edit** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Request receiver, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Response sender, click **Web services: Server security bindings**. Under Response sender binding, click **Edit** > **Key locators**. Under Additional properties, click `Keys`.
   - For the Response receiver, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit** > **Key locators**. Under Additional properties, click **Keys**.

Key name:

Specifies the name of the key object that is found in the keystore file.

Key alias:

Specifies an alias for the key object.

The alias is used when the key locator searches for the key objects in the keystore file.

Key configuration settings:

Use this page to define the mapping of a logical name to a key alias in a keystore file.

To view this administrative console panel for the key collection on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default generator bindings, click **Token Generators** > ***token_generator_name***.
3. Under Additional properties, click **Callback handler** > **Keys**.
4. Specify a new key configuration by clicking **New** or by clicking the key configuration name to modify the settings.

Keys are also available from the JAX-WS and JAX-RPC security runtime panel by clicking **Key locators** > ***key_locator_name***. Under Additional properties, click **Keys** > **New** Specify a new key configuration by clicking New or by clicking the key configuration name to modify the settings..

To view this administrative console page for the key locator collection on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default generator bindings, click **Token Generators** > ***token_generator_name***.
4. Under Additional properties, click **Callback handler** > **Keys**.
5. Specify a new key configuration by clicking **New** or by clicking the key configuration name to modify the settings.

Keys are also available from the JAX-WS and JAX-RPC security runtime panel by clicking **Key locators** > ***key_locator_name***. Under Additional properties, click **Keys** > **New**. Specify a new key configuration by clicking New or by clicking the key configuration name to modify the settings.

To use this administrative console page for the key locator collection on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Additional properties, you can access key locators for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Key locators**. Under Additional properties, click **Keys**.
4. Under Web Services Security Properties, you can access key locators for the following bindings:
   - For the Request sender, click **Web services: Client security bindings**. Under Request sender binding, click **Edit** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Request receiver, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Response sender, click **Web services: Server security bindings**. Under Response sender binding, click **Edit** > **Key locators**. Under Additional properties, click **Keys**.
   - For the Response receiver, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit** > **Key locators**. Under Additional properties, click **Keys**.
5. Specify a new key configuration by clicking **New** or by clicking the key configuration name to modify the settings.

Key name:

Specifies the name of the key object. For digital signatures, the key name is used by the request sender or request generator signing information to determine which key is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption.

The key name must be a fully qualified, distinguished name. For example, `CN:Bob,O=IBM,C=US`.

Note: If you enter the distinguished name with spaces before or after commas and equal symbols, the application server normalizes the distinguished names automatically during run time by removing these extra spaces.

Key alias:

Specifies the alias for the key object, which is used by the key locator to find the key within the keystore file.

Key password:

Specifies the password that is needed to access the key object within the keystore file.

Web services: Client security bindings collection:

Use this page to view a list of application-level, client-side binding configurations for Web Services Security. These bindings are used when a web service is a client to another web service.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

To view this administrative console page, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, click **Web services: Client security bindings**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Component:

Specifies the enterprise bean in an assembled Enterprise JavaBeans (EJB) module.

Port:

Specifies the port that is used to send messages to a server and receive messages from a server.

Web service:

Specifies the name of the web service that is defined during application assembly.

Request generator (sender) binding:

Specifies the binding configuration that is used to send request messages to the request consumer.

Click **Edit custom** to configure the required and additional properties such as signing information, key information, token generators, key locators, and collection certificate stores.

The binding information for the request generator that is specified for the client must match the binding information for the request consumer that is specified for the server.

Response consumer (receiver) binding:

Specifies the binding configuration that is used to receive response messages from the response generator.

Click **Edit custom** to configure the required and additional properties such as signing information, key information, token consumers, key locators, collection certificate stores, and trust anchors.

The binding information for the response consumer that is specified for the client must match the binding information for the response generator that is specified for the server.

Request sender binding:

Specifies the binding configuration that is used to send request messages to the request receiver.

Click **Edit** to configure the additional properties for the request sender such as signing information, key information, encryption information, key locators, and the login binding.

The binding information for the request sender that is specified for the client must match the binding information for the request receiver that is specified for the server.

Response receiver binding:

Specifies the binding configuration that is used to receive response messages from the response sender.

Click **Edit** to configure the additional properties for the response receiver such as signing information, encryption information, trust anchors, collection certificate stores, and key locators.

The binding information for the response receiver that is specified for the client must match the binding information for the response sender that is specified for the server.

HTTP basic authentication:

Specifies the user name and password to use for this port with HTTP transport-level basic authentication. You can enable transport-level authentication security independently of message-level security.

Although the name of this field is HTTP basic authentication, you can use this field to specify the user name and password in conjunction with any transport method. This field is not specific to HTTP transport. For example, you can use this same field with Java Message Service (JMS).

Click **Edit** to configure the basic authentication ID and password for transport-level authentication.

HTTP SSL configuration:

Enables and configures transport-level Secure Sockets Layer (SSL) security for this port. You can enable transport-level SSL security independently of message-level security.

Click **Edit** to specify the settings for transport-level HTTP SSL configuration for this port.

Web services: Server security bindings collection:

Use this page to view a list of server-side binding configurations for Web Services Security.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.

2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, click **Web services: Server security bindings**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Port:

Specifies the port in which messages are received from the request generator.

Web service:

Specifies the name of the web service that is defined during application assembly.

Request consumer (receiver) binding:

Specifies the binding configuration that is used to receive request messages from the request generator (sender) binding.

Click **Edit custom** to configure the required and additional information such as signing information, key information, token consumers, key locators, intermediate certificates in the collection certificate store, and trust anchors.

The binding information for the request consumer that is specified for the server must match the binding information for the request generator that is specified for the client.

Response generator (sender) binding:

Specifies the binding configuration that is used to send response messages to the response consumer.

Click **Edit custom** to configure the required and additional information such as signing information, key information, token generators, key locators, and intermediate certificates in the collection certificate store.

The binding information for the response generator that is specified for the server must match the binding information for the response consumer that is specified for the client.

Request receiver binding:

Specifies the binding configuration that is used to receive request messages from the request sender binding.

Click **Edit** to configure additional properties for the request receiver such as signing information, encryption information, trust anchors, collection certificate stores, key locators, trusted ID evaluators, and login mappings.

The binding information for the request receiver that is specified for the server must match the binding information for the request sender that is specified for the client.

Response sender binding:

Specifies the binding configuration that is used to send response messages to the response receiver.

Click **Edit** to configure additional properties for the response sender such as signing information, encryption information, and key locators.

The binding information for the response sender that is specified for the server must match the binding information for the response receiver that is specified for the client.

Configuring token consumers using JAX-RPC to protect message authenticity at the application level:

You can specify the token consumer on the application level. The token consumer information is used on the consumer side to incorporate the security token.

Before you begin

You need to understand that the keystore/alias information that you provide for the generator, and the keystore/alias information that you provide for the consumer are used for different purposes. The main difference applies to the Alias for an X.509 callback handler.

When used in association with an encryption consumer, the alias supplied for the consumer is used to retrieve the private key to decrypt the message. A password is required. When associated with a signature consumer, the alias supplied for the consumer is used strictly to retrieve the public key that is used to resolve an X.509 certificate that is not passed in the SOAP security header as a BinarySecurityToken. A password is not required.

About this task

Complete the following steps to configure the token consumer on the application level.

Procedure

1. Locate the token consumer panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Modules, click **Manage modules** > ***URI_name***.
   c. Under Web Services Security Properties you can access the token consumer for the following bindings:
      - For the request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
      - For the response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   d. Under Required properties, click **Token consumer**.
   e. Click **New** to create a token consumer configuration, click **Delete** to delete an existing configuration, or click the name of an existing token consumer configuration to edit its settings. If you are creating a new configuration, enter a unique name in the **Token consumer name** field. For example, you might specify `con_signtcon`.
2. Specify a class name in the **Token consumer class name** field. The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to validate (authenticate) the security token on the consumer side.

   The token consumer class name for the request consumer and the response consumer must be similar to the token generator class name for the request generator and the response generator. For example, if your application requires a user name token consumer, you can specify the com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator class name on the Token generator panel for application level and the com.ibm.wsspi.wssecurity.token.UsernameTokenConsumer class name in this field.
3. Optional: Select a part reference in the **Part reference** field. The part reference indicates the name of the security token that is defined in the deployment descriptor. For example, if you receive a username token in your request message, you might want to reference the token in the username token consumer.

Important: On the application level, if you do not specify a security token in your deployment descriptor, the **Part reference** field is not displayed. If you define a security token called `user_tcon` in your deployment descriptor, `user_tcon` is displayed as an option in the **Part reference** field.

4. Optional: In the certificate path section of the panel, select a certificate store type and indicate the trust anchor and certificate store name, if necessary. These options and fields are necessary when you specify com.ibm.wsspi.wssecurity.token.X509TokenConsumer as the token consumer class name. The names of the trust anchor and the collection certificate store are created in the certificate path under your token consumer.

   Restriction: The `com.ibm.wsspi.wssecurity.token.TokenConsumingComponent` interface is not used with JAX-WS web services. If you are using JAX-RPC web services, this interface is still valid.

   You can select one of the following options:

   None If you select this option, the certificate path is not specified.

   Trust any
   If you select this option, any certificate is trusted. When the received token is consumed, the Application Server does not validate the certificate path.

   Dedicated signing information
   If you select this option, you can select a trust anchor and a certificate store configuration. When you select the trust anchor or the certificate store of a trusted certificate, you must configure the trust anchor and the certificate store before setting the certificate path.

   Trust anchor
   A trust anchor specifies a list of key store configurations that contain trusted root certificates. These configurations are used to validate the certificate path of incoming X.509-formatted security tokens. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trustworthiness of a certificate chain. You must create the keystore file using the key tool utility, which is located in the *install_dir*/java/jre/bin/keytool file.

   You can configure trust anchors for the application level by completing the following steps:

   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Related Items, click **EJB Modules** or **Web Modules** > ***URI_name***.
   c. Access the token consumer from the following bindings:
      - For the request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
      - For the response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   d. Under Additional properties, click **Trust anchors**.

   Collection certificate store
   A collection certificate store includes a list of untrusted, intermediary certificates and certificate revocation lists (CRLs). The collection certificate store is used to validate the certificate path of the incoming X.509-formatted security tokens. You can configure the collection certificate store for the application level by completing the following steps:

   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.

b. Under Related Items, click **EJB Modules** or **Web Modules** > ***URI_name***.

c. Access the token consumer from the following bindings:
   - For the request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
   - For the response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.

d. Under Additional properties, click **Collection certificate store**.

5. Optional: Specify a trusted ID evaluator. The trusted ID evaluator is used to determine whether to trust the received ID. You can select one of the following options:

None If you select this option, the trusted ID evaluator is not specified.

Existing evaluator definition
If you select this option, you can select one of the configured trusted ID evaluators. For example, you can select the **SampleTrustedIDEvaluator**, which is provided by WebSphere Application Server as an example.

Binding evaluator definition
If you select this option, you can configure a new trusted ID evaluator by specifying a trusted ID evaluator name and class name.

Trusted ID evaluator name
Specifies the name that is used by the application binding to refer to a trusted identity (ID) evaluator that is defined in the default bindings.

Trusted ID evaluator class name
Species the class name of the trusted ID evaluator. The specified trusted ID evaluator class name must implement the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface. The default TrustedIDEvaluator class is com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl. When you use this default TrustedIDEvaluator class, you must specify the name and value properties for the default trusted ID evaluator to create the trusted ID list for evaluation. To specify the name and value properties, complete the following steps:

a. Under Additional properties, click **Properties** > **New**.

b. Specify the trusted ID evaluator name in the **Property** field. You must specify the name in the form, `trustedId_n` where `_n` is an integer from 0 to n.

c. Specify the trusted ID in the **Value** field.

For example:

```
property name="trustedId_0", value="CN=Bob,O=ACME,C=US"
property name="trustedId_1, value="user1"
```

If the distinguished name (DN) is used, the space is removed for comparison. See the programming model information in the documentation for an explanation of how to implement the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface. For more information, see "Default implementations of the Web Services Security service provider programming interfaces" on page 263.

Note: Define the trusted ID evaluator on the server level instead of the application level. To define the trusted ID evaluator on the server level, complete the following steps:

a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.

b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

c. Under Additional properties, click **Trusted ID evaluators**.

d. Click **New** to define a new trusted ID evaluator.

The trusted ID evaluator configuration is available only for the token consumer on the server-side application level.

6. Optional: Select the **Verify nonce** option. This option indicates whether to verify a nonce in the user name token if it is specified for the token consumer. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens. The **Verify nonce** option is valid only when the incorporated token type is a user name token.
7. Optional: Select the **Verify timestamp** option. This option indicates whether to verify a time stamp in the user name token. The **Verify nonce** option is valid only when the incorporated token type is a user name token.
8. Specify the value type local name in the **Local name** field. This field specifies the local name of the value type for the consumed token. For a user name token and an X.509 certificate security token, WebSphere Application Server provides predefined local names for the value type.

Table 174. Uniform Resource Identifier (URI) and Local name combinations. The local name value indicates the type of consumed token.

URI	Local name	Description
A namespace URI is not applicable.	Specify `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3` as the local name value.	Specifies the name of an X.509 certificate token
A namespace URI is not applicable.	Specify `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1` as the local name value.	Specifies the name of the X.509 certificates in a PKI path
A namespace URI is not applicable.	Specify `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7` as the local name value.	Specifies a list of X509 certificates and certificate revocation lists (CRL) in a PKCS#7
Specify `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` as the URI value.	Specify `LTPA` as the local name value.	Specifies a binary security token that contains an embedded Lightweight Third Party Authentication (LTPA) token.

9. Optional: Specify the value type URI in the **URI** field. This entry specifies the namespace URI of the value type for the consumed token.

   Remember: If you specify the token consumer for a username token or an X.509 certificate security token, you do not need to specify a value type URI.

   If you want to specify another token, you must specify both the local name and the URI. For example, if you have an implementation of your own custom token, you can specify `CustomToken` in the **Local name** field and `http://www.ibm.com/custom`
10. Click **OK** and **Save** to save the configuration.
11. Click the name of your token consumer configuration.
12. Under Additional properties, click **JAAS configuration**. The Java Authentication and Authorization Service (JAAS) configuration specifies the name of the JAAS configuration that is defined in the JAAS login panel. The JAAS configuration specifies how the token logs in on the consumer side.
13. Select a JAAS configuration from the **JAAS configuration name** field. The field specifies the name of the JAAS system of application login configuration. You can specify additional JAAS system and application configurations by clicking **Global security**. Under Authentication, click **Java Authentication and Authorization Service** and click either **Application logins** > **New** or **System logins** > **New**. For more information on the JAAS configurations, see "JAAS configuration settings" on page 885. Do not remove the predefined system or application login configurations. However,

within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module. WebSphere Application Server provides the following predefined JAAS configurations:

ClientContainer
: This selection specifies the login configuration that is used by the client container applications. The configuration uses the CallbackHandler application programming interface (API) that is defined in the deployment descriptor for the client container. To modify this configuration, see the JAAS configuration panel for application logins.

WSLogin
: This selection specifies whether all of the applications can use the WSLogin configuration to perform authentication for the security run time. To modify this configuration, see the JAAS configuration panel for application logins.

DefaultPrincipalMapping
: This selection specifies the login configuration that is used by Java 2 Connectors (J2C) to map users to principals that are defined in the J2C authentication data entries. To modify this configuration, see the JAAS configuration panel for application logins.

system.LTPA_WEB
: This selection processes login requests that are used by the web container such as servlets and JavaServer Pages (JSPs) files. To modify this configuration, see the JAAS configuration panel for system logins.

system.RMI_OUTBOUND
: This selection processes RMI requests that are sent outbound to another server when the com.ibm.CSIOutboundPropagationEnabled property is `true`. This property is set in the CSIv2 authentication panel.

  To access the panel, click **Security** > **Global security**. Under Authentication, click **RMI/IIOP security** > **CSIv2 Outbound authentication**. To set the `com.ibm.CSIOutboundPropagationEnabled` property, select **Security attribute propagation**. To modify this JAAS login configuration, see the JAAS - System logins panel.

system.wssecurity.X509BST
: This selection verifies an X.509 binary security token (BST) by checking the validity of the certificate and the certificate path. To modify this configuration, see the JAAS configuration panel for system logins.

system.wssecurity.PKCS7
: This selection verifies an X.509 certificate within a PKCS7 object that might include a certificate chain, a certificate revocation list, or both. To modify this configuration, see the JAAS configuration panel for system logins.

system.wssecurity.PkiPath
: This section verifies an X.509 certificate with a public key infrastructure (PKI) path. To modify this configuration, see the JAAS configuration panel for system logins.

system.wssecurity.UsernameToken
: This selection verifies the basic authentication (user name and password) data. To modify this configuration, see the JAAS configuration panel for system logins.

system.wssecurity.IDAssertionUsernameToken
: This selection supports the use of identity assertion in Versions 6 and later applications to map a user name to a WebSphere Application Server credential principal. To modify this configuration, see the JAAS configuration panel for system logins.

system.WSS_INBOUND
: This selection specifies the login configuration for inbound or consumer requests for security token propagation using Web Services Security. To modify this configuration, see the JAAS configuration panel for system logins.

system.WSS_OUTBOUND
This selection specifies the login configuration for outbound or generator requests for security token propagation using Web Services Security. To modify this configuration, see the JAAS configuration panel for system logins.

None With this selection, you do not specify a JAAS login configuration.

14. Click **OK** and then click **Save** to save the configuration.

Results

You have configured the token consumer for the application level.

What to do next

You must specify a similar token generator configuration for the application level.

Request consumer (receiver) binding configuration settings:

Use this page to specify the binding configuration for the request consumer.

To view this administrative console page, complete the following steps:

1. Click **Applications > Application Types > WebSphere enterprise applications***application_name*.
2. Click **Manage modules**.
3. Click the Uniform Resource Identifier (URI).
4. Under Web Services Security Properties, click **Web services: Server security bindings**.
5. Under Request consumer (receiver) binding, click **Edit custom**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

The security constraints or bindings are defined using the application assembly process before the application is installed.

An assembly tool is not available on the z/OS platform.

If the security constraints are defined in the application, you must either define the corresponding binding information or select the **Use defaults** option on this panel and use the default binding information for the cell or server level. The default binding that is provided by this product is a sample. Do not use this sample in a production environment without modifying the configuration. The security constraints define what is signed or encrypted in the Web Services Security message. The bindings define how to enforce the requirements.

Digital signature security constraint (integrity)

The following table shows the required and optional binding information when the digital signature security constraint (integrity) is defined in the deployment descriptor.

Table 175. Binding information for digital signature security constraints. The binding information is used for signing or encrypting messages.

Information type	Required or optional
Signing information	Required
Key information	Required
Token consumer	Required
Key locators	Optional

Table 175. Binding information for digital signature security constraints (continued). The binding information is used for signing or encrypting messages.

Information type	Required or optional
Collection certificate store	Optional
Trust anchors	Optional
Properties	Optional

You can use the key locators, collection certificate stores, and trust anchors that are defined at either the server level or the cell level.

Encryption constraint (confidentiality)

The following table shows the required and optional binding information when the encryption constraint (confidentiality) is defined in the deployment descriptor.

Table 176. Binding information for encryption constraints. The binding information is used for signing or encrypting messages.

Information type	Required or optional
Encryption information	Required
Key information	Required
Token consumer	Required
Key locators	Optional
Collection certificate store	Optional
Trust anchors	Optional
Properties	Optional

You can use the key locators, collection certificate store, and trust anchors that are defined at either the server level or the cell level.

Security token constraint

The following table shows the required and optional binding information when the security token constraint is defined in the deployment descriptor.

Table 177. Binding information for security token constraints. The binding information is used for signing or encrypting messages.

Information type	Required or optional
Token consumer	Required
Collection certificate store	Optional
Trust anchors	Optional
Properties	Optional

You can use the collection certificate store and trust anchors that are defined at the server level or the cell level.

Use defaults:

Select this option if you want to use the default binding information from the server or cell level.

If you select this option, the application server checks for binding information on the server level. If the binding information does not exist on the server level, the application server checks the cell level.

Port:

Specifies the port in the web service that is defined during application assembly.

Web service:

Specifies the name of the web service that is defined during application assembly.

Response consumer (receiver) binding configuration settings:

Use this page to specify the binding configuration for the response consumer.

To view this administrative console page, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules**.
3. Click the Uniform Resource Identifier (URI).
4. Under Web Services Security Properties, click **Web services: Client security bindings**.
5. Under Response consumer (receiver) binding, click **Edit custom**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

The security constraints or bindings are defined using the application assembly process before the application is installed.

An assembly tool is not available on the z/OS platform.

If the security constraints are defined in the application, you must either define the corresponding binding information or select the **Use defaults** option on this panel and use the default binding information for the server or cell level. The default binding that is provided by this product is a sample. Do not use this sample in a production environment without modifying the configuration. The security constraints define what is signed or encrypted in the Web Services Security message. The bindings define how to enforce the requirements.

Digital signature security constraint (integrity)

The following table shows the required and optional binding information when the digital signature security constraint (integrity) is defined in the deployment descriptor.

Table 178. Binding information for digital signature security constraints. The binding information is used for validating digital signature.

Information type	Required or optional
Signing information	Required
Key information	Required
Token consumer	Optional
Key locators	Optional
Collection certificate store	Optional
Trust anchors	Optional
Properties	Optional

You can use the key locators, collection certificate stores, and trust anchors that are defined at either the server level or the cell level.

Encryption constraint (confidentiality)

The following table shows the required and optional binding information when the encryption constraint (confidentiality) is defined in the deployment descriptor.

Table 179. Binding information for encryption constraints. The binding information is used for decrypting messages.

Information type	Required or optional
Encryption information	Required
Key information	Required
Token consumer	Optional
Key locators	Optional
Collection certificate store	Optional
Trust anchors	Optional
Properties	Optional

You can use the key locators, collection certificate store, and trust anchors that are defined at the application level, server level, or the cell level.

Security token constraint

The following table shows the required and optional binding information when the security token constraint is defined in the deployment descriptor.

Table 180. Binding information for security token constraints. The binding information is used for digital signature verification and for decrypting messages.

Information type	Required or optional
Token consumer	Required
Collection certificate store	Optional
Trust anchors	Optional
Properties	Optional

You can use the collection certificate store and trust anchors that are defined at the application level, server level, or the cell level.

Use defaults:

Select this option if you want to use the default binding information from the cell or server level.

If you select this option, the application server checks for binding information on the server level. If the binding information does not exist on the server level, the application server checks the cell level.

Component:

Specifies the enterprise bean in an assembled Enterprise JavaBeans (EJB) module.

Port:

Specifies the port in the web service that is defined during application assembly.

Web service:

Specifies the name of the web service that is defined during application assembly.

JAAS configuration settings:

Use this page to specify the name of the Java Authentication and Authorization Service (JAAS) configuration that is defined in the JAAS login panel.

Complete the following steps to access this page on the cell level:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.

2. Under JAX-RPC Default Consumer Bindings, click **Token consumers** > ***token_consumer_name*** or click **New** to create a new token consumer.
3. Under Additional properties, click **JAAS configuration**.

Complete the following steps to access this page on the server level:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Consumer Bindings, click **Token consumers** > ***token_consumer_name*** or click **New** to create a new token consumer.
4. Under Additional properties, click **JAAS configuration**.

Complete the following steps to access this page on the application level:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access the JAAS configuration settings for the following bindings:
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Token consumers** > ***token_consumer_name*** or click **New** to create a new token consumer. Under Additional properties, click **JAAS configuration**.
   - For the Request consumer (receiver) binding, click **Web services: Server security binding**. Under Request consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Token consumers** > ***token_consumer_name*** or click **New** to create a new token consumer. Under Additional properties, click **JAAS configuration**.

   Important: If you create a new token consumer, you must click **Apply** before you can proceed to the JAAS configuration.

JAAS configuration name:

Specifies the name of the JAAS system or application login configuration.

Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which the application server loads each module.

Preconfigured system login configurations

The following predefined system login configurations are defined on the system logins panel, which is accessible by completing the following steps:
1. Click **Security** > **Global security**.
2. Expand Java Authentication and Authorization Service, click **System logins**.

system.wssecurity.IDAssertionUsernameToken
: Enables a Version 6.x application to use identity assertion to map a user name to an application server credential principal.

system.wssecurity.IDAssertion
: Enables an application to use identity assertion to map a user name to an application server credential principal.

system.wssecurity.Signature
: Enables an application to map a distinguished name (DN) in a signed certificate to an application server credential principal.

system.LTPA_WEB
: Processes login requests used by the web container such as servlets and JavaServer Pages (JSP) files.

system.WEB_INBOUND
: Handles logins for web application requests, which include servlets and JavaServer Pages (JSP) files.

system.RMI_INBOUND
: Handles logins for inbound Remote Method Invocation (RMI) requests.

system.DEFAULT
: Handles the logins for inbound requests that are made by internal authentications and most of the other protocols ecept web applications and RMI requests.

system.RMI_OUTBOUND
: Processes RMI requests that are sent outbound to another server when either the com.ibm.CSI.rmiOutboundLoginEnabled or the com.ibm.CSIOutboundPropagationEnabled properties are `true`. These properties are set in the Common Secure Interoperability Version 2 (CSIv2) authentication panel.

  To access the panel, click **Security** > **Global security**. Epand RMI/IIOP security, click **CSIv2 Outbound authentication**. To set the `com.ibm.CSI.rmiOutboundLoginEnabled` property, select the **Custom outbound mapping** option. To set the `com.ibm.CSIOutboundPropagationEnabled` property, select the **Security attribute propagation** option.

system.wssecurity.509BST
: Verifies an .509 binary security token (BST) by checking the validity of the certificate and the certificate path.

system.wssecurity.PKCS7
: Verifies an .509 certificate with a certificate revocation list in a Public Key Cryptography Standards #7 (PKCS7) object.

system.wssecurity.PkiPath
: Verifies an .509 certificate with a public key infrastructure (PKI) path.

system.wssecurity.UsernameToken
: Verifies basic authentication (user name and password).

Application login configurations

The following predefined application login configurations are defined on the Application logins panel, which is accessible by completing the following steps:

1. Click **Security** > **Global security**.
2. Expand Java Authentication and Authorization Service, click **Application logins**.

ClientContainer
: Specifies the login configuration that is used by the client container application. This application uses the CallbackHandler API that is defined in the deployment descriptor of the client container.

WSLogin
: Specifies whether all applications can use the WSLogin configuration to perform authentication for the application server security run time.

DefaultPrincipalMapping
: Specifies the login configuration that is used by Java 2 Connectors (J2C) to map users to principals that are defined in the J2C authentication data entries.

Configuring encryption using JAX-RPC to protect message confidentiality at the application level:

You can configure encryption information, used to specify how the generators (senders) encrypt outgoing messages, for the request generator (client side) and the response generator (server side) bindings at the application level.

Before you begin

Configure the key information that is referenced by the key information references in the encryption information panel.

About this task

This task provides the steps that are needed for configuring encryption information for the request generator (client side) and the response generator (server side) bindings at the application level. This encryption information is used to specify how the generators (senders) encrypt outgoing messages.

Complete the following steps to configure the encryption information for the request generator or response generator section of the bindings file on the application level:

Procedure

1. Locate the encryption information configuration panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security Properties, you can access the key information for the request generator and response generator bindings.
      - For the request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
      - For the response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   d. Under Required properties, click **Encryption information**.
   e. Click **New** to create an encryption information configuration. Click **Delete** to delete an existing configuration or click the name of an existing encryption information configuration to edit its settings. If you are creating a new configuration, enter a name in the **Encryption information name** field. For example, you might specify `gen_encinfo`.
2. Select a data encryption algorithm from the **Data encryption algorithm** field. The selection specifies the algorithm that is used to encrypt parts of the message. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/04/xmlenc#tripledes-cbc
   - http://www.w3.org/2001/04/xmlenc#aes128-cbc
   - http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - http://www.w3.org/2001/04/xmlenc#aes192-cbc

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

   Restriction: Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

The data encryption algorithm that you select for the generator side must match the data encryption method that you select for the consumer side.

3. Select a key encryption algorithm from the **Key encryption algorithm** field. This selection specifies the algorithm that is used to encrypt keys. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

     When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this one. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5.

     Restriction: This algorithm is not supported when the WebSphere Application Server is running in Federal Information Processing Standard (FIPS) mode.

     By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:
     - `http://www.w3.org/2001/04/xmlenc#sha256`
     - `http://www.w3.org/2001/04/xmlenc#sha512`

     By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

     Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.
   - http://www.w3.org/2001/04/xmlenc#rsa-1_5
   - http://www.w3.org/2001/04/xmlenc#kw-tripledes
   - http://www.w3.org/2001/04/xmlenc#kw-aes128
   - http://www.w3.org/2001/04/xmlenc#kw-aes256

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - http://www.w3.org/2001/04/xmlenc#kw-aes192

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

     Restriction: Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

   The key encryption algorithm that you select for the generator side must match the key encryption method that you select for the consumer side.

4. Select an encryption key information reference from the Encryption key information menu. This selection is a reference to the encryption key that is used to encrypt parts of the message. To configure the key information, see "Configuring the key information using JAX-RPC for the generator binding on the application level" on page 844.
5. Select a part reference from the **Part reference** field. This field specifies the name of the part reference for the generator binding element in the deployment descriptor.
6. Click **OK** and then click **Save** to save the configuration.

Results

The encryption information is configured for the generator binding at the application level.

What to do next

You must specify a similar encryption information configuration for the consumer.

Encryption information collection:

Use this page to specify the configuration for the encrypting and decrypting parameters. This configuration is used to encrypt and decrypt parts of the message, including the body and user name token.

To view the administrative console panel for the encryption information on the cell level, complete the following steps:
1. Click **Security > JAX-WS and JAX-RPC security runtime**.
2. Under either JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Encryption information**.

To view the administrative console panel for the encryption information on the server level, complete the following steps:
1. Click **Servers > Server Types > WebSphere application servers >** *server_name*.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under either JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Encryption information**.

To view this administrative console page for the collection certificate store on the application level, complete the following steps:
1. Click **Applications > Application Types > WebSphere enterprise applications***application_name*.
2. Under Modules, click **Manage modules >** ***URI_name***.
3. Under Web Services Security Properties, you can access encryption information for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**. Under Required properties, click **Encryption information**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Encryption information**.
   - For the Response generator, click **Web services: Server security bindings.** Under Response generator (sender) binding, click **Edit custom**. Under Required properties, click **Encryption information**.

- For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Encryption information**.

4. Under Additional properties, you can access encryption information for the following bindings:
   - For the Request receiver, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit**. Under Additional properties, click **Encryption information**.
   - For the Response receiver, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit**. Under Additional properties, click **Encryption information**.

Encryption information name:

Specifies the name of the encryption information.

Key locator reference:

Specifies the name of the key locator configuration that retrieves the key for XML digital signature and XML encryption.

Key encryption algorithm: Specifies the algorithm that is used to encrypt and decrypt keys.

Data encryption algorithm: Specifies the algorithm that is used to encrypt and decrypt data.

Encryption information configuration settings: Message parts:

Use this page to configure the encryption and decryption parameters. You can use these parameters to encrypt and decrypt various parts of the message, including the body and the token.

To view the administrative console panel for the encryption information on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under either JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Encryption information**.
3. Click **New** to create a new encryption configuration or click the name of an existing encryption configuration.

To view the administrative console panel for the encryption information on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under either JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Encryption information**.
4. Click **New** to create a new encryption configuration or click the name of an existing encryption configuration.

To view this administrative console page for the encryption information on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Module update** > ***module_name***.

3. Under Web Services Security Properties, you can access encryption information for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**. Under Required properties, click **Encryption information**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Encryption information**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**. Under Required properties, click **Encryption information**.
   - For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Encryption information**.
4. Click either **New** to create a new encryption configuration or click the name of an existing encryption configuration.

Note: Fix packs that include updates to the Software Development Kit (SDK) might overwrite unrestricted policy files. Back up unrestricted policy files before you apply a fix pack and reapply these files after the fix pack is applied.

Encryption information name:

Specifies the name for the encryption information.

Information	Value
Data type	String

Data encryption algorithm:

Specifies the algorithm Uniform Resource Identifier (URI) of the data encryption method.

The following algorithms are supported:
- http://www.w3.org/2001/04/xmlenc#tripledes-cbc
- http://www.w3.org/2001/04/xmlenc#aes128-cbc
- http://www.w3.org/2001/04/xmlenc#aes256-cbc. To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html. For more information, see “Encryption information configuration settings: Methods” on page 897.
- http://www.w3.org/2001/04/xmlenc#aes192-cbc. To use this algorithm, you must download the unrestricted JCE policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html. For more information, see the help topic Encryption information configuration settings: Methods.

  Restriction: Do not use the 192-bit data encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files. For more information, see the **Key encryption algorithm** field description.

Key locator reference:

Specifies the name of the key locator configuration that retrieves the key for XML digital signature and XML encryption.

The Key locator reference field is displayed for the request receiver and response receiver bindings.

You can configure these key locator reference options on the server level, the cell level, and the application level. The configurations that are listed in the field are a combination of the configurations on these three levels.

You can specify an encryption key configuration for the following bindings on the following levels:

Table 181. Encryption key binding configurations. Use these configurations to encrypt and decrypt various parts of a message.

Binding name	Server level, cell level, or application level	Path
Default generator binding	Cell level	1. Click **Security > JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Key locators**.
Default consumer bindings	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Key locators**.
Default generator binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Key locators**.
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Key locators**.
Request sender	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. Under Request sender binding, click **Edit**. 4. Under Additional properties, click **Key locators**.
Request receiver	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. Under Request receiver binding, click **Edit**. 4. Under Additional properties, click **Key locators**.
Response sender	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Server security bindings**. Under Response sender binding, click **Edit**. 4. Under Additional properties, click **Key locators**.

Table 181. Encryption key binding configurations (continued). Use these configurations to encrypt and decrypt various parts of a message.

Binding name	Server level, cell level, or application level	Path
Response receiver	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Click **Web services: Client security bindings**. Under Response receiver binding, click **Edit**. 4. Under Additional properties, click **Key locators**.

Key encryption algorithm:

Specifies the algorithm Uniform Resource Identifier (URI) of the key encryption method.

The following algorithms are provided by the application server:

- http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

  When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this one. This algorithm appears in the list of supported key transport algorithms when running with Software Development Kit (SDK) Version 1.5 or later.

  By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:

  - `http://www.w3.org/2001/04/xmlenc#sha256`
  - `http://www.w3.org/2001/04/xmlenc#sha512`

  By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

  Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

- http://www.w3.org/2001/04/xmlenc#rsa-1_5
- http://www.w3.org/2001/04/xmlenc#kw-tripledes
- http://www.w3.org/2001/04/xmlenc#kw-aes128
- http://www.w3.org/2001/04/xmlenc#kw-aes192

  Restriction: Do not use the 192-bit data encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

- http://www.w3.org/2001/04/xmlenc#kw-aes256

Application server platforms and IBM Developer Kit, Java Technology Edition Version 1.4.2

By default, the Java Cryptography Extension (JCE) ships with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

Before downloading these policy files, back up the existing policy files (`local_policy.jar` and `US_export_policy.jar` in the *WAS_HOME*`/java/lib/security/` directory) prior to overwriting them in case you want to restore the original files later.

Attention: Fix packs that include updates to the Software Development Kit (SDK) might overwrite unrestricted policy files. Back up unrestricted policy files before you apply a fix pack and reapply these files after the fix pack is applied.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

To download the policy files, complete one of the following sets of steps:

After following either of these sets of steps, two Java archive (JAR) files are placed in the Java virtual machine (JVM) `jre/lib/security/` directory.

Application server platform and IBM Developer Kit, Java Technology Edition Version 5

By default, the Java Cryptography Extension (JCE) ships with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files. Before downloading these policy files, back up the existing policy files (`local_policy.jar` and `US_export_policy.jar` in the *WAS_HOME*`/java/lib/security/` directory) prior to overwriting them in case you want to restore the original files later.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

To download the policy files, complete one of the following sets of steps:

- For application server platforms using IBM Developer Kit, Java Technology Edition Version 5, you can obtain unlimited jurisdiction policy files by completing the following steps:
  1. Go to the following website: IBM developer works: Security Information
  2. Click **Java 5**
  3. Click **IBM SDK Policy files**.

     The Unrestricted JCE Policy files for SDK 5 website is displayed.
  4. Enter your user ID and password or register with IBM to download the policy files. The policy files are downloaded onto your machine.

After following these sets of steps, two Java archive (JAR) files are placed in the Java virtual machine (JVM) `jre/lib/security/` directory.

Custom algorithms on the cell level

To specify custom algorithms on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Algorithm mappings**.
3. Click **New** to specify a new algorithm mapping or click the name of an existing configuration to modify its settings.
4. Under Additional properties, click **Algorithm URI**.
5. Click **New** to create a new algorithm URI. You must specify **Key encryption** in the **Algorithm type** field to have the configuration display in the **Key encryption algorithm** field on the Encryption information configuration settings panel.

Custom algorithms on the server level

To specify custom algorithms on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Algorithm mappings**.
4. Click **New** to specify a new algorithm mapping or click the name of an existing configuration to modify its settings.
5. Under Additional properties, click **Algorithm URI**.
6. Click **New** to create a new algorithm URI. You must specify **Key encryption** in the **Algorithm type** field to have the configuration display in the **Key encryption algorithm** field on the Encryption information configuration settings panel.

Encryption key information:

Specifies the name of the key information reference that is used for encryption. This reference is resolved to the actual key by the specified key locator and defined in the key information.

You must specify either one or no encryption key configurations for the request generator and response generator bindings.

For the response consumer and the request consumer bindings, you can configure multiple encryption key references. To create a new encryption key reference, under Additional properties, click **Key information references**.

You can specify an encryption key configuration for the following bindings on the following levels:

Table 182. Encryption key binding configurations. Use these configurations to encrypt and decrypt various parts of a message.

Binding name	Server level, cell level, or application level	Path
Default generator binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under JAX-RPC Default generator binding, click **Key information**.
Default consumer binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under JAX-RPC Default consumer binding, click **Key information**.
Default generator binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under JAX-RPC Default generator binding, click **Key information**.

Table 182. Encryption key binding configurations (continued). Use these configurations to encrypt and decrypt various parts of a message.

Binding name	Server level, cell level, or application level	Path
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under JAX-RPC Default consumer binding, click **Key information**.
Request generator (sender) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Under Web Services Security Properties, click **Web services: Client security bindings**. 4. Under Request generator (sender) binding, click **Edit custom**. 5. Under Required properties, click **Key information**.
Response generator (sender) binding	Application level	1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***. 2. Under Modules, click **Manage modules** > ***URI_name***. 3. Under Web Services Security Properties, click **Web services: Server security bindings**. 4. Under Response generator (sender) binding, click **Edit custom**. 5. Under Required properties, click **Key information**.

Part Reference:

Specifies the name of the <confidentiality> element for the generator binding or the <requiredConfidentiality> element for the consumer binding element in the deployment descriptor.

This field is available on the application level only.

Encryption information configuration settings: Methods:

Use this page to configure the encryption and decryption parameters for the signature method, digest method, and canonicalization method.

The specifications that are listed on this page for the signature method, digest method, and canonicalization method are located in the World Wide Web Consortium (W3C) document entitled, *XML Encryption Syntax and Processing: W3C Recommendation 10 Dec 2002*.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** and complete one of the following steps:
   - Click **Manage modules** > ***URI_file_name*** > **Web Services: Client Security Bindings**. Under Request sender binding, click **Edit.** Under Web Services Security Properties, click **Encryption Information**.
   - Under Modules, click **Manage modules** > ***URI_file_name*** > **Web Services: Server Security Bindings**. Under Response sender binding, click **Edit.** Under Web Services Security Properties, click **Encryption Information.**

2. Select **None** or **Dedicated encryption information**. The application server can have either one or no encryption configurations for the request sender and the response sender bindings. If you are not using encryption, select **None**. To configure encryption for either of these two bindings, select **Dedicated encryption information** and specify the configuration settings using the fields that are described in this topic.

Note: Fix packs that include updates to the Software Development Kit (SDK) might overwrite unrestricted policy files. Back up unrestricted policy files before you apply a fix pack and reapply these files after the fix pack is applied.

Encryption information name:

Specifies the name for the encryption information.

Key locator reference:

Specifies the name that is used to reference the key locator.

You can configure these key locator reference options on the cell level, the server level, and the application level. The configurations that are listed in the field are a combination of the configurations on these three levels.

To configure the key locators on the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Key locators**.

To configure the key locators on the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Key locators**.

To configure the key locators on the application level, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access the key locators for the following bindings:
   - For the Request sender, click **Web services: Client security bindings**. Under Request sender binding, click **Edit**. Under Additional properties, click **Key locators**.
   - For the Request receiver, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit**. Under Additional properties, click **Key locators**.
   - For the Response sender, click **Web services: Server security bindings**. Under Response sender binding, click **Edit**. Under Additional properties, click **Key locators**.
   - For the Response receiver, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit**. Under Additional properties, click **Key locators**.

Encryption key name:

Specifies the name of the encryption key that is resolved to the actual key by the specified key locator.

Information	Value
Data type	String

Key encryption algorithm:

Specifies the algorithm uniform resource identifier (URI) of the key encryption method.

The following algorithms are supported:

- http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

  When running with IBM Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this one. This algorithm appears in the list of supported key transport algorithms when running with JDK 1.5 or later.

  By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:
  - `http://www.w3.org/2001/04/xmlenc#sha256`
  - `http://www.w3.org/2001/04/xmlenc#sha512`

  By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

  Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.
- http://www.w3.org/2001/04/xmlenc#rsa-1_5.
- http://www.w3.org/2001/04/xmlenc#kw-tripledes.
- http://www.w3.org/2001/04/xmlenc#kw-aes128.
- http://www.w3.org/2001/04/xmlenc#kw-aes192. To use the 192-bit key encryption algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file.

  Restriction: Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
- http://www.w3.org/2001/04/xmlenc#kw-aes256. To use the 256-bit key encryption algorithm, you must download the unrestricted JCE policy file.

Note: If an InvalidKeyException error occurs and you are using the 129xxx or 256xxx encryption algorithm, the unrestricted policy files might not exist in your configuration.

Java Cryptography Extension

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

Note: Before downloading these policy files, back up the existing policy files (`local_policy.jar` and `US_export_policy.jar` in the *WAS_HOME*`/java/lib/security/` directory) prior to overwriting them in case you want to restore the original files later.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy

files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

Application server platforms and IBM Developer Kit, Java Technology Edition Version 1.4.2

To download the policy files, complete one of the following sets of steps:

- For application server platforms using IBM Developer Kit, Java Technology Edition Version 1.4.2, including the AIX®, Linux, and Windows platforms, complete the following steps to obtain unlimited jurisdiction policy files:
  1. Go to the following website: IBM developer kit: Security information
  2. Click **Java 1.4.2**
  3. Click **IBM SDK Policy files**.

     The Unrestricted JCE Policy files for SDK 1.4 website is displayed.
  4. Enter your user ID and password or register with IBM to download the policy files. The policy files are downloaded onto your machine.

After completing these steps, two Java archive (JAR) files are placed in the Java virtual machine (JVM) `jre/lib/security/` directory.

Data encryption algorithm:

Specifies the algorithm Uniform Resource Identifiers (URI) of the data encryption method.

The following algorithms are supported:

- http://www.w3.org/2001/04/xmlenc#tripledes-cbc
- http://www.w3.org/2001/04/xmlenc#aes128-cbc
- http://www.w3.org/2001/04/xmlenc#aes192-cbc

  Restriction: Do not use the 192-bit data encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
- http://www.w3.org/2001/04/xmlenc#aes256-cbc

By default, the JCE ships with restricted or limited strength ciphers. To use 192-bit and 256- bit AES encryption algorithms, you must apply unlimited jurisdiction policy files. For more information, see the Key encryption algorithm field description.

Configuring encryption to protect message confidentiality at the application level:

You can configure the encryption information for the request consumer (server side) and response consumer (client side) bindings at the application level.

Before you begin

Configure the key information that is referenced in the encryption information panel. For more information, see “Configuring the key information for the consumer binding on the application level” on page 855.

About this task

This task provides the steps that are needed for configuring the encryption information for the request consumer (server side) and response consumer (client side) bindings at the application level. The encryption information on the consumer side is used for decrypting the encrypted message parts in the incoming SOAP message.

Complete the following steps to configure the encryption information for the request consumer or response consumer section of the bindings file on the application level:

Procedure

1. Locate the Encryption information configuration panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security Properties you can access the encryption information for the request consumer and response consumer bindings.
      - For the request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
      - For the response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   d. Under Required properties, click **Encryption information**.
   e. Click **New** to create an encryption information configuration, click **Delete** to delete an existing configuration, or click the name of an existing encryption information configuration to edit its settings. If you are creating a new configuration, enter a name in the **Encryption information name** field. For example, you might specify `cons_encinfo`.
2. Select a data encryption algorithm from the **Data encryption algorithm** field. The data encryption algorithm is used for encrypting or decrypting parts of a SOAP message such as the SOAP body or the username token. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/04/xmlenc#tripledes-cbc
   - http://www.w3.org/2001/04/xmlenc#aes128-cbc
   - http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - http://www.w3.org/2001/04/xmlenc#aes192-cbc

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

     Restriction: Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

     Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

   The data encryption algorithm that you select for the consumer side must match the data encryption method that you select for the generator side.
3. Select a key encryption algorithm from the **Key encryption algorithm** field. The key encryption algorithm is used for encrypting the key that is used for encrypting the message parts within the SOAP message. Select **(none)** if the data encryption key, which is the key that is used for encrypting the message parts, is not encrypted. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this one. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5.

Restriction: This algorithm is not supported when the WebSphere Application Server is running in Federal Information Processing Standard (FIPS) mode.

- http://www.w3.org/2001/04/xmlenc#rsa-1_5
- http://www.w3.org/2001/04/xmlenc#kw-tripledes
- http://www.w3.org/2001/04/xmlenc#kw-aes128
- http://www.w3.org/2001/04/xmlenc#kw-aes256

  To use the http://www.w3.org/2001/04/xmlenc#aes256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
- http://www.w3.org/2001/04/xmlenc#kw-aes192

  To use the http://www.w3.org/2001/04/xmlenc#kw-aes192 algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

  Restriction: Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

The key encryption algorithm that you select for the consumer side must match the key encryption method that you select for the generator side.

4. Optional: Select a part reference in the **Part reference** field. The part reference specifies the name of the message part that is encrypted and is defined in the deployment descriptor. For example, you can encrypt the bodycontent message part in the deployment descriptor. The name of this Required Confidentiality part is `conf_con`. This message part is shown as an option in the **Part reference** field.
5. Under Additional properties, click **Key information references**.
6. Click **New** to create a key information configuration, click **Delete** to delete an existing configuration, or click the name of an existing key information configuration to edit its settings. If you are creating a new configuration, enter a name in the **Name** field. For example, you might specify `con_ekeyinfo`. This entry is the name of the <encryptionKeyInfo> element in the binding file.
7. Select a key information reference from the **Key information reference** field. This reference is the value of the keyinfoRef attribute of the <encryptionKeyInfo> element and it is the name of the <keyInfo> element that is referenced by this key information reference. Each key information reference entry generates an <encryptionKeyInfo> element under the <encryptionInfo> element in the binding configuration file. For example, if you enter `con_ekeyinfo` in the **Name** field and `dec_keyinfo` in the **Key information reference** field, the following <encryptionKeyInfo> element is generated in the binding file:

```
<encryptionKeyInfo xmi:id="EncryptionKeyInfo_1085092248843"
keyinfoRef="dec_keyinfo" name="con_ekeyinfo"/>
```

8. Click **OK** and then click **Save** to save the configuration.

Results

You have configured the encryption information for the consumer binding at the application level

What to do next

You must specify a similar encryption information configuration for the generator.

Configuring message-level security for JAX-RPC at the server or cell level:

Specify the server-level or cell-level configuration.

Configuring the signing information using JAX-RPC for the generator binding on the server or cell level:

You can configure the signing information for the client-side request generator and the server-side response generator bindings at the server or cell level.

Before you begin

Note: For WebSphere Application Server version 6.x or earlier only, in the server-side extensions file (`ibm-webservices-ext.xmi`) and the client-side deployment descriptor extensions file (`ibm-webservicesclient-ext.xmi`), you must specify which parts of the message are signed. Also, you need to configure the key information that is referenced by the key information references on the Signing information panel within the administrative console.

About this task

This task explains the steps that are needed for you to configure the signing information for the client-side request generator and the server-side response generator bindings at the server or cell level. WebSphere Application Server uses the signing information for the default generator to sign parts of the message that include the body, time stamp, and user name token if these bindings are not defined at the application level. The Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

You can configure the signing information for the generator binding on the server level and the cell level. In the following steps, use the first step to configure the signing information for the server level and use the second step to configure the signing information on the cell level:

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

      Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Default generator bindings, click **Signing information**.
4. Click **New** to create a signing information configuration, click **Delete** to delete an existing configuration, or click the name of an existing signing information configuration to edit the settings. If you are creating a new configuration, enter a unique name for the signing configuration in the Signing information name field. For example, you might specify `gen_signinfo`.

   Note: If you create more than one signing information configuration, the WS-Security runtime environment only honors the first configuration listed in the bindings file.
5. Select a signature method algorithm from the Signature method field. The algorithm that is specified for the default generator must match the algorithm that is specified for the default consumer. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - http://www.w3.org/2000/09/xmldsig#hmac-sha1
   - http://www.w3.org/2000/09/xmldsig#dsa-sha1

     Do not use this algorithm if you want the configured application to be compliant with the Basic Security Profile (BSP). Any ds:SignatureMethod/@Algorithm element in a SIGNATURE based on a symmetric key must have a value of http://www.w3.org/2000/09/xmldsig#rsa-sha1 or http://www.w3.org/2000/09/xmldsig#hmac-sha1.

6. Select a canonicalization method from the Canonicalization method field. The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer. WebSphere Application Server supports the following pre-configured canonical XML and exclusive XML canonicalization algorithms:
   - http://www.w3.org/2001/10/xml-exc-c14n#
   - http://www.w3.org/2001/10/xml-exc-c14n#WithComments
   - http://www.w3.org/TR/2001/REC-xml-c14n-20010315
   - http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments
7. Select a key information signature type from the **Key information signature type** field. The key information signature type determines how to digitally sign the key. WebSphere Application server supports the following signature types:

   None Specifies that the <KeyInfo> element is not signed.

   Keyinfo
   Specifies that the entire <KeyInfo> element is signed.

   Keyinfochildelements
   Specifies that the child elements of the <KeyInfo> element are signed.

   The key information signature type for the generator must match the signature type for the consumer. You might encounter the following situations:
   - If you do not specify one of the previous signature types, WebSphere Application Server uses `keyinfo`, by default.
   - If you select Keyinfo or Keyinfochildelements and you select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm in a subsequent step, WebSphere Application Server also signs the referenced token.
8. Select a signing key information reference from the Signing key information field. This selection is a reference to the signing key that the Application Server uses to generate digital signatures. In the binding files, this information is specified within the <signingKeyInfo> tag. The key that is used for signing is specified by the key information element, which is defined at the same level as the signing information. For more information, see "Configuring the key information for the generator binding using JAX-RPC on the server or cell level" on page 908.
9. Click **OK** to save the configuration.
10. Click the name of the new signing information configuration. This configuration is the one that you specified in the previous steps.
11. Specify the part reference, digest algorithm, and transform algorithm. The part reference specifies which parts of the message to digitally sign.
    a. Under Additional Properties, click **Part references** > **New** to create a new part reference, click **Part references** > **Delete** to delete an existing part reference, or click a part name to edit an existing part reference.
    b. Specify a unique part name for the message part that needs signing. This message part is specified on both the server side and the client side. You must specify an identical part name for both the server side and the client side. For example, you might specify `reqint` for both the generator and the consumer.

       Important: You do not need to specify a value for the Part reference in the default bindings like you specify on the application level because the part reference on the application level points to a particular part of the message that is signed. Because the default bindings for the server and cell levels are applicable to all of the services that are defined on a particular server, you cannot specify this value.
    c. Select a digest method algorithm in the **Digest method algorithm** field. The digest method algorithm that is specified in the binding files within the <DigestMethod> element is used in the <SigningInfo> element.

WebSphere Application Server supports the following algorithms:

- http://www.w3.org/2000/09/xmldsig#sha1
- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

d. Click **OK** and **Save** to save the configuration.

e. Click the name of the new part reference configuration. This configuration is the one that you specified in the previous steps.

f. Under Additional properties, click **Transforms** > **New** to create a new transform, click **Transforms** > **Delete** to delete a transform, or click a transform name to edit an existing transform. If you create a new transform configuration, specify a unique name. For example, you might specify `reqint_body_transform1`.

g. Select a transform algorithm from the menu. The transform algorithm is specified within the <Transform> element. This algorithm element specifies the transform algorithm for the digital signature. WebSphere Application Server supports the following algorithms:

- http://www.w3.org/2001/10/xml-exc-c14n#
- http://www.w3.org/TR/1999/REC-xpath-19991116

  Restriction: Do not use this transform algorithm if you want your configured application to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.
- http://www.w3.org/2002/06/xmldsig-filter2
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
- http://www.w3.org/2002/07/decrypt#XML
- http://www.w3.org/2000/09/xmldsig#enveloped-signature

The transform algorithm that you select for the generator must match the transform algorithm that you select for the consumer.

Important: If both of the following conditions are true, WebSphere Application Server signs the referenced token:

- You previously selected the **Keyinfo** or the **Keyinfochildelements** option from the Key information signature type field on the signing information panel.
- You select **http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform** as the transform algorithm.

12. Click **Apply**.
13. Click **Save** at the top of the panel to save your configuration.

Results

After completing these steps, you have configured the signing information for the generator on the server or cell level.

What to do next

You must specify a similar signing information configuration for the consumer.

Configuring the signing information using JAX-RPC for the consumer binding on the server or cell level:

You can configure the signing information for the client-side request generator and server-side response generator bindings at the server or cell level.

Before you begin

Note: For WebSphere Application Server version 6.x or earlier only, in the server-side extensions file (`ibm-webservices-ext.xmi`) and the client-side deployment descriptor extensions file (`ibm-webservicesclient-ext.xmi`), you must specify which parts of the message are signed. Also, you need to configure the key information that is referenced by the key information references on the signing information panel within the administrative console.

About this task

This task explains the steps that are needed for you to configure the signing information for the client-side request generator and server-side response generator bindings at the server or cell level. WebSphere Application Server uses the signing information for the default generator to sign parts of the message including the body, time stamp, and user name token, if these bindings are not defined at the application level. The Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

You can configure the signing information for the consumer binding on the server level and the cell level. In the following steps, use the first step to access the server-level default bindings and use the second step to access the cell-level bindings.

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

      Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Default consumer bindings, click **Signing information**.
4. Click **New** to create a signing information configuration, click **Delete** to delete an existing configuration, or click the name of an existing signing information configuration to edit the settings. If you are creating a new configuration, enter a unique name for the signing configuration in the Signing information name field. For example, you might specify `gen_signinfo`.

   Note: If you create more than one signing information configuration, the WS-Security runtime environment only honors the first configuration listed in the bindings file.
5. Select a signature method algorithm from the Signature method field. The algorithm that is specified for the default consumer must match the algorithm that is specified for the default generator. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - http://www.w3.org/2000/09/xmldsig#hmac-sha1
   - http://www.w3.org/2000/09/xmldsig#dsa-sha1

     Do not use this algorithm if you want the configured application to be compliant with the Basic Security Profile (BSP). Any ds:SignatureMethod/@Algorithm element in a SIGNATURE based on a symmetric key must have a value of http://www.w3.org/2000/09/xmldsig#rsa-sha1 or http://www.w3.org/2000/09/xmldsig#hmac-sha1.
6. Select a canonicalization method from the Canonicalization method field. The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer. WebSphere Application Server supports the following pre-configured canonical XML and exclusive XML canonicalization algorithms:
   - http://www.w3.org/2001/10/xml-exc-c14n#

- http://www.w3.org/2001/10/xml-exc-c14n#WithComments
- http://www.w3.org/TR/2001/REC-xml-c14n-20010315
- http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments

7. Select a key information signature type from the Key information signature type field. The key information signature type determines how to digitally sign the key. WebSphere Application Server supports the following signature types:

   None Specifies that the KeyInfo element is not signed.

   Keyinfo
   Specifies that the entire KeyInfo element is signed.

   Keyinfochildelements
   Specifies that the child elements of the KeyInfo element are signed.

   The key information signature type for the consumer must match the signature type for the generator. You might encounter the following situations:

   - If you do not specify one of the previous signature types, WebSphere Application Server uses keyinfo, by default.
   - If you select **Keyinfo** or **Keyinfochildelements** and you select **http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform** as the transform algorithm in a subsequent step, WebSphere Application Server also signs the referenced token.

8. Click **OK** to save the configuration.
9. Click the name of the new signing information configuration. This configuration is the one that you specified in the previous steps.
10. Specify the key information reference, part reference, digest algorithm, and transform algorithm.
    a. Under Additional properties, click **Key information references** > **New** to create a new reference, click **Key information references** > **Delete** to delete an existing reference, or click a reference name to edit an existing key information reference.
    b. Enter a name for the configuration in the Name field. For example, enter `con_skeyinfo`.
    c. Select a key information reference from the Key information reference field. The key Information reference points to the key that WebSphere Application Server uses for digital signing. In the binding files, the reference is specified within the <signingKeyInfo> element. The key that is used for signing is specified by the Key information element, which is defined at the same level as the signing information. For more information, see "Configuring the key information for the consumer binding on the application level" on page 855.
    d. Click **OK** and **Save** to save the configuration.
    e. Under Additional Properties, click **Part references** > **New** to create a new part reference, click **Part references** > **Delete** to delete an existing part reference, or click a part name to edit an existing part reference. The part reference specifies which parts of the message to digitally sign. The part attribute refers to the name of the `<RequiredIntegrity>` element in the deployment descriptor when `<PartReference>` is specified for the digital signature. WebSphere Application Server enables you to specify multiple `<PartReference>` elements for the `<SigningInfo>` element. The `<PartReference>` element has two child elements: `<DigestMethod>` and `<Transform>`.
    f. Specify a unique part name for this part reference. For example, you might specify `reqint`.

       Important: You do not need to specify a value for the Part Reference field like you specify on the application level because the part reference on the application level points to a particular part of the message that is signed. Because the default bindings for the server and cell levels are applicable to all of the services defined on a particular server, you cannot specify this value.

    g. Select a digest method algorithm in the **Digest method algorithm** field. The digest method algorithm specified within the `<DigestMethod>` element that is used in the `<SigningInfo>` element.

       WebSphere Application Server supports the following algorithms:

- http://www.w3.org/2000/09/xmldsig#sha1
- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

h. Click **OK** and **Save** to save the configuration.

i. Click the name of the new part reference configuration. This configuration is the one that you specified in the previous steps.

j. Under Additional properties, click **Transforms** > **New** to create a new transform, click **Transforms** > **Delete** to delete a transform, or click a transform name to edit an existing transform. If you create a new transform configuration, specify a unique name. For example, you might specify `reqint_body_transform1`.

k. Select a transform algorithm from the menu. The transform algorithm is specified within the <Transform> element. It specifies the transform algorithm for the signature. WebSphere Application Server supports the following algorithms:

- http://www.w3.org/2001/10/xml-exc-c14n#
- http://www.w3.org/TR/1999/REC-xpath-19991116

  Restriction: Do not use this transform algorithm if you want your configured application to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.
- http://www.w3.org/2002/06/xmldsig-filter2
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
- http://www.w3.org/2002/07/decrypt#XML
- http://www.w3.org/2000/09/xmldsig#enveloped-signature

The transform algorithm that you select for the consumer must match the transform algorithm that you select for the generator.

Important: If both of the following conditions are true, WebSphere Application Server signs the referenced token:

- You previously selected the Keyinfo or the Keyinfochildelements option from the Key information signature type field on the signing information panel.
- You select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm.

11. Click **OK**.
12. Click **Save** at the top of the panel to save your configuration.

Results

After completing these steps, you have configured the signing information for the consumer on the server or cell level.

What to do next

You must specify a similar signing information configuration for the generator.

Configuring the key information for the generator binding using JAX-RPC on the server or cell level:

Use the key information for the default generator to specify the key that is used by the signing or the encryption information configurations if these bindings are not defined at the application level.

About this task

The signing and encryption information configurations can share the same key information, which is why they are both defined on the same level. WebSphere Application Server provides default values for these bindings. However, an administrator must modify these values for a production environment.

You can configure the key information for the generator binding on the server level and the cell level. In the following steps, use the first step to configure the key information on the server level or use the second step to configure the key information on the cell level:

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

      Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Default generator bindings, click **Key information**.
4. Click **New** to create a key information configuration, click **Delete** to delete an existing configuration, or click the name of an existing key information configuration to edit the settings. If you are creating a new configuration, enter a unique name for the key configuration in the Key information name field. For example, you might specify `sig_keyinfo`.
5. Select a key information type from the Key information type field. WebSphere Application Server supports the following types of key information:

   Key identifier
   This key information type is used when two parties agree on how to create a key identifier. For example, a field of X.509 certificates can be used for the key identifier according to the X.509 profile.

   Key name
   This key information type is used when the sender and receiver agree on the name of the key.

   Security token reference
   This key information type is typically used when an X.509 certificate is used for digital signature.

   Embedded token
   This key information type is used to embed a security token in an embedded element.

   X509 issuer name and issuer serial
   This key information type specifies an X.509 certificate with its issuer name and serial number.

   Select **Security token reference** if you are using an X.509 certificate for the digital signature. In these steps, it is assumed that **Security token reference** is selected for this field.

   Important: This key information type must match the key information type that is specified for the consumer.
6. Select a key locator reference from the Key locator reference menu. In these steps, assume that the key locator reference is called `sig_klocator`. The key locator reference is the name of the key locator that is used to generate the key for digital signature. You must configure a key locator before you can select it in this field. For more information on configuring the key locator, see "Configuring the key locator using JAX-RPC on the server or cell level" on page 952.

7. Click **Get keys** to view a list of key name references. After you click **Get keys**, the key names that are defined in the <sig_klocator> element are shown in the key name reference menu. If you change the key locator reference, you must click **Get keys** again to display the list of key names that are associated with the new key locator.
8. Select a key name reference from the Key name reference menu. The key name reference specifies the name of the key that is used for generating the digital signature or for encryption. The Key name reference menu displays a list of key names that are defined for the selected key locator in the Key locator reference field. For example, select **signerkey**. It is assumed that signer key is a key name that is defined for the `sig_klocator` key locator.
9. Select a token reference from the Token reference field. The token reference refers to the name of a configured token generator. When a security token is required in the deployment descriptor, the token reference attribute is required. If you select **Security token reference** in the Key information type field, the token reference is required and you can specify an X.509 token generator. To specify an X.509 token generator, you must have an X.509 token generator configured. To configure an X.509 token generator, see "Configuring token generators using JAX-RPC to protect message authenticity at the server or cell level" on page 916. For the remaining steps, it is assumed that an X.509 token generator that is named gen_tcon is already configured.
10. Optional: Select an encoding method from the Encoding method field This field specifies the encoding format for the key identifier. The encoding method attribute is valid when you select **Key identifer** as the key information type. WebSphere Application Server supports the following encoding methods:
    - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary
    - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#HexBinary
11. Optional: Select a calculation method from the Calculation method field. The calculation method specifies the calculation algorithm that is used for the key identifier. This attribute is valid when you select **Key identifier** as the key information type. WebSphere Application Server supports the following calculation methods:
    - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#ITSHA1
    - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#IT60SHA1
12. Optional: Specify a Uniform Resource Identifier (URI) of the value type for a security token from the Namespace URI field. The namespace URI is referenced by the key identifier. This attribute is valid when you select **Key identifier** as the key information type. When you specify the X.509 certificate token, you do not need to specify the namespace URI. If another token is specified, you must specify the namespace URI. For example, you can specify `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` for the Lightweight Third Party Authentication (LTPA) token and `http://www.ibm.com/websphere/appserver/tokentype` for the LTPA_PROPAGATION token.
13. Optional: Specify the local name of the value type for a security token in the **Local name** field. The local name is referenced by the key identifier. This attribute is valid when you select **Key identifier** as the key information type. WebSphere Application Server supports the following local names:

    For an X.509 certificate token
    http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3

    For X.509 certificates in a PKIPath
    http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1

    For a list of X.509 certificates and CRLs in a PKCS#7
    http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

    For LTPA
    LTPA

    For LTPA_PROPAGATION
    LTPA_PROPAGATION
14. Click **OK** and **Save** to save the configuration.

Results

You have configured the key information for the generator binding at the server or cell level.

What to do next

You must specify a similar key information configuration for the consumer.

Configuring the key information for the consumer binding using JAX-RPC on the server or cell level:

The key information for the default consumer is used to specify the key for the signing or the encryption information configurations if these bindings are not defined at the application level.

About this task

The signing and encryption information configurations can share the same key information, which is why they are both defined on the same level. WebSphere Application Server provides default values for these bindings. However, an administrator must modify these values for a production environment.

You can configure the key information for the consumer binding on the server level and the cell level. In the following steps, use the first step to access the server-level default bindings and use the second step to access the cell-level bindings:

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Default consumer bindings, click **Key information**.
4. Click **New** to create a key information configuration, click **Delete** to delete an existing configuration, or click the name of an existing key information configuration to edit the settings. If you are creating a new configuration, enter a unique name for the key configuration in the Key information name field. For example, you might specify `con_signkeyinfo`.
5. Select a key information type from the Key information type field. WebSphere Application Server supports the following types of key information:

   Key identifier
   This key information type is used when two parties agree on how to create a key identifier. For example, a field of X.509 certificates can be used for the key identifier according to the X.509 profile.

   Key name
   This key information type is used when the sender and receiver agree on the name of the key.

   Security token reference
   This key information type is typically used when an X.509 certificate is used for digital signature.

   Embedded token
   This key information type is used to embed a security token in an embedded element.

   X509 issuer name and issuer serial
   This key information type specifies an X.509 certificate with its issuer name and serial number.

Select **Security token reference** if you are using an X.509 certificate for the digital signature. In these steps, it is assumed that **Security token reference** is selected for this field.

Important: This key information type must match the key information type that is specified for the generator.

6. Select a key locator reference from the Key locator reference menu. In these steps, assume that the key locator reference is called `sig_klocator`. You must configure a key locator before you can select it in this field. For more information on configuring the key locator, see "Configuring the key locator using JAX-RPC on the server or cell level" on page 952.
7. Select a token reference from the Token reference field. The token reference refers to the name of a configured token consumer. When a security token is required in the deployment descriptor, the token reference attribute is required. If you select **Security token reference** in the Key information type field, the token reference is required and you can specify an X.509 token consumer. To specify an X.509 token consumer, you must have an X.509 token consumer configured. To configure an X.509 token consumer, see "Configuring token consumers using JAX-RPC to protect message authenticity at the server or cell level" on page 929.
8. Click **OK** and **Save** to save the configuration.

Results

You have configured the key information for the consumer binding at the server or cell level.

What to do next

You must specify a similar key information configuration for the generator.

Configuring encryption using JAX-RPC to protect message confidentiality at the server or cell level:

You can configure the encryption information for the generator binding on the server or cell level.

About this task

The encryption information for the default generator specifies how to encrypt the information on the sender side if these bindings are not defined at the application level. WebSphere Application Server provides default values for the bindings. However, an administrator must modify the defaults for a production environment.

You can configure the encryption information for the generator binding on the server level and the cell level. In the following steps, use the first step to configure the encryption information for the server level and use the second step to configure the encryption information for the cell level:

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

      Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Default generator bindings, click **Encryption information**.
4. Click **New** to create an encryption information configuration, click **Delete** to delete an existing configuration, or click the name of an existing encryption information configuration to edit the settings. If you are creating a new configuration, enter a unique name for the encryption configuration in the Encryption information name field. For example, you might specify `gen_encinfo`.

Note: If you create more than one encryption information configuration, the WS-Security runtime environment only honors the first configuration listed in the bindings file.

5. Select a data encryption algorithm from the Data encryption algorithm field. This algorithm is used to encrypt the data. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/04/xmlenc#tripledes-cbc
   - http://www.w3.org/2001/04/xmlenc#aes128-cbc
   - http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - http://www.w3.org/2001/04/xmlenc#aes192-cbc

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

     Restriction: Do not use this algorithm, the 192-bit key encryption algorithm, if you want your configured application to be in compliance with the Basic Security Profile (BSP).

   Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

   The data encryption algorithm that you select for the generator side must match the data encryption algorithm that you select for the consumer side.
6. Select a key encryption algorithm from the Key encryption algorithm field. This algorithm is used to encrypt the key. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

     When running with JDK 1.4, the list of supported key transport algorithms will not include this one. This algorithm will appear in the list of supported key transport algorithms when running with JDK 1.5.

     Restriction: This algorithm is not supported when the WebSphere Application Server is running in Federal Information Processing Standard (FIPS) mode.

     By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:
     – `http://www.w3.org/2001/04/xmlenc#sha256`
     – `http://www.w3.org/2001/04/xmlenc#sha512`

     By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

     Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.
   - http://www.w3.org/2001/04/xmlenc#rsa-1_5
   - http://www.w3.org/2001/04/xmlenc#kw-tripledes

- http://www.w3.org/2001/04/xmlenc#kw-aes128
- http://www.w3.org/2001/04/xmlenc#kw-aes256

  To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
- http://www.w3.org/2001/04/xmlenc#kw-aes192

  To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

  Restriction: Do not use this algorithm, the 192-bit key encryption algorithm, if you want your configured application to be in compliance with the Basic Security Profile (BSP).

If you select **None**, the key is not encrypted.

The key encryption algorithm that you select for the generator side must match the key encryption algorithm that you select for the consumer side.

7. Select a encryption key configuration from the Encryption key information field. This attribute specifies the name of the key that is used to encrypt the message. To configure the key information, see “Configuring the key information for the generator binding using JAX-RPC on the server or cell level” on page 908.
8. Click **OK** and then click **Save** to save the configuration.

Results

You have configured the encryption information for the generator binding at the server or cell level.

What to do next

You must specify a similar encryption information configuration for the consumer.

Configuring encryption to protect message confidentiality at the server or cell level:

The encryption information for the default consumer specifies how to process the encryption information on the receiver side if these bindings are not defined at the application level. WebSphere Application Server provides default values for the bindings. However, an administrator must modify the defaults for a production environment.

About this task

You can configure the encryption information for the consumer binding on the server level and the cell level. In the following steps, use the first step to access the server-level default bindings and use the second step to access the cell-level bindings.

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Default consumer bindings, click **Encryption information**.
4. Click **New** to create an encryption information configuration, click **Delete** to delete an existing configuration, or click the name of an existing encryption information configuration to edit the settings.

If you are creating a new configuration, enter a unique name for the encryption configuration in the Encryption information name field. For example, you might specify `con_encinfo`.

Note: If you create more than one encryption information configuration, the WS-Security runtime environment only honors the first configuration listed in the bindings file.

5. Select a data encryption algorithm from the Data encryption algorithm field. This algorithm is used to encrypt the data. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/04/xmlenc#tripledes-cbc
   - http://www.w3.org/2001/04/xmlenc#aes128-cbc
   - http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - http://www.w3.org/2001/04/xmlenc#aes192-cbc

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

     Restriction: Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

     Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

   The data encryption algorithm that you select for the consumer side must match the data encryption algorithm that you select for the generator side.
6. Select a key encryption algorithm from the Key encryption algorithm field. This algorithm is used to encrypt the key. WebSphere Application Server supports the following pre-configured algorithms:
   - http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

     When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this one. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5.

     Restriction: This algorithm is not supported when the WebSphere Application Server is running in Federal Information Processing Standard (FIPS) mode.
   - http://www.w3.org/2001/04/xmlenc#rsa-1_5
   - http://www.w3.org/2001/04/xmlenc#kw-tripledes
   - http://www.w3.org/2001/04/xmlenc#kw-aes128
   - http://www.w3.org/2001/04/xmlenc#kw-aes256

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - http://www.w3.org/2001/04/xmlenc#kw-aes192

     To use this algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

     Restriction: Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

If you select **None**, the key is not encrypted.

The key encryption algorithm that you select for the consumer side must match the key encryption algorithm that you select for the generator side.

7. Under Additional properties, click **Key information references**.
8. Click **New** to create a key information configuration, click **Delete** to delete an existing configuration, or click the name of an existing key information configuration to edit the settings. If you are creating a new configuration, enter a unique name for the key information configuration in the name field. For example, you might specify `con_enckeyinfo`.
9. Select a key information reference from the Key information reference field. This selection refers to the name of the key information that is used for encryption. For more information, see "Configuring the key information for the consumer binding using JAX-RPC on the server or cell level" on page 911.
10. Click **OK** and **Save** to save the configuration.

Results

You have configured the encryption information for the consumer binding at the server or cell level.

What to do next

You must specify a similar encryption information configuration for the generator.

Configuring token generators using JAX-RPC to protect message authenticity at the server or cell level:

The token generator on the server or cell level is used to specify the information for the token generator if these bindings are not defined at the application level. The signing information and the encryption information can share the token generator information, which is why they are all defined at the same level.

Before you begin

You need to understand that the keystore/alias information that you provide for the generator, and the keystore/alias information that you provide for the consumer are used for different purposes. The main difference applies to the Alias for an X.509 callback handler.

When used in association with an encryption generator, the alias supplied for the generator is used to retrieve the public key to encrypt the message. A password is not required. The alias that is entered on a callback handler associated with an encryption generator must be accessible without a password. This means that the alias must not have private key information associated with it in the keystore. When used in association with a signature generator, the alias supplied for the generator is used retrieve the private key to sign the message. A password is required.

About this task

WebSphere Application Server provides default values for bindings. You must modify the defaults for a production environment.

You can configure the token generator on the server level and the cell level. In the following steps, use the first step to access the server-level default bindings and use the second step to access the cell-level bindings.

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Default generator bindings, click **Token generators**.
4. Click **New** to create a token generator configuration, click **Delete** to delete an existing configuration, or click the name of an existing token generator configuration to edit its settings. If you are creating a new configuration, enter a unique name for the token generator configuration in the **Token generator name** field. For example, you might specify `sig_tgen`. This field specifies the name of the token generator element.
5. Specify a class name in the **Token generator class name** field. The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side.

   Restriction: The com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent interface is not used with JAX-WS web services. If you are using JAX-RPC web services, this interface is still valid.

   The token generator class name must be similar to the token consumer class name. For example, if your application requires an X.509 certificate token consumer, you can specify the com.ibm.wsspi.wssecurity.token.X509TokenConsumer class name on the Token consumer panel and the com.ibm.wsspi.wssecurity.token.X509TokenGenerator class name in this field. WebSphere Application Server provides the following default token generator class implementations:

   com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator
   This implementation generates a username token.

   com.ibm.wsspi.wssecurity.token.X509TokenGenerator
   This implementation generates an X.509 certificate token.

   com.ibm.wsspi.wssecurity.token.LTPATokenGenerator
   This implementation generates a Lightweight Third Party Authentication (LTPA) token.
6. Select a certificate path option. The certificate path specifies the certificate revocation list (CRL), which is used for generating a security token that is wrapped in a PKCS#7 with a CRL. WebSphere Application Server provides the following certificate path options:

   None Select this option in case the CRL is not used for generating a security token. You must select this option when the token generator does not use the PKCS#7 token type.

   Dedicated signing information
   If the CRL is wrapped in a security token, select **Dedicated signing information** and select a collection certificate store name from the **Certificate store** field. The **Certificate store** field shows the names of collection certificate stores already defined.

   To define a collection certificate store on the cell level, see “Configuring the collection certificate on the server or cell level” on page 973.
7. Select the **Add nonce** option to include a nonce in the user name token for the token generator. Nonce is a unique cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens. The **Add nonce** option is available if you specify a user name token for the token generator.
8. Select the **Add timestamp** option to include a time stamp in the user name token for the token generator.
9. Specify a value type local name in the **Local name** field. This entry specifies the local name of the value type for a security token that is referenced by the key identifier. This attribute is valid when **Key identifier** is selected as Key information type. To specify the Key information type, see “Configuring the key information for the generator binding using JAX-RPC on the server or cell level” on page 908. WebSphere Application Server provides the following predefined X.509 certificate token configurations:

X.509 certificate token
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3

X.509 certificates in a PKIPath
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1

A list of X.509 certificates and CRLs in a PKCS#7
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

LTPA For LTPA, the value type local name is LTPA. If you enter `LTPA` for the local name, you must specify the http://www.ibm.com/websphere/appserver/tokentype/5.0.2 uniform resource identifier (URI) value in the **Value type URI** field as well.

LTPA version 2
For LTPA version 2, the value type local name is LTPAv2. If you enter LTPAv2 for the local name, you must specify the http://www.ibm.com/websphere/appserver/tokentype uniform resource identifier (URI) value in the **Value type URI** field as well.

LTPA_PROPAGATION
For LTPA token propagation, the value type local name is `LTPA_PROPAGATION`. If you enter `LTPA_PROPAGATION` for the local name, you must specify the `http://www.ibm.com/websphere/appserver/tokentype` URI value in the **Value type URI** field as well.

For example, when an X.509 certificate token is specified, you can use `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3` for the local name.

10. Specify the value type URI in the **URI** field. This entry specifies the namespace URI of the value type for a security token that is referenced by the key identifier. This attribute is valid when **Key identifier** is selected as Key information type on the Key information panel for the default generator. When the X.509 certificate token is specified, you do not need to specify the namespace URI. If another token is specified, you must specify the namespace URI of the value type.
11. Click **OK** and then **Save** to save the configuration.
12. Click the name of your token generator configuration.
13. Under Additional properties, click **Callback handler** to configure the callback handler properties. The callback handler specifies how to acquire the security token that is inserted in the Web Services Security header within the SOAP message. The token acquisition is a pluggable framework that leverages the Java Authentication and Authorization Service (JAAS) `javax.security.auth.callback.CallbackHandler` interface for acquiring the security token.
    a. Specify a callback handler class implementation in the **Callback handler class name** field. This attribute specifies the name of the Callback handler class implementation that is used to plug in a security token framework. The specified callback handler class must implement the `javax.security.auth.callback.CallbackHandler` class. WebSphere Application Server provides the following default callback handler implementations:

       com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
       This callback handler uses a login prompt to gather the user name and password information. However, if you specify the user name and password on this panel, a prompt is not displayed and WebSphere Application Server returns the user name and password to the token generator. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

       com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
       This callback handler does not issue a prompt and returns the user name and password if it is specified in the basic authentication section of this panel. You can use this callback handler when the web service is acting as a client.

       com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
       This callback handler uses a standard-in prompt to gather the user name and password. However, if the user name and password is specified in the basic authentication section of

this panel, WebSphere Application Server does not issue a prompt, but returns the user name and password to the token generator. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler
: This callback handler is used to obtain the Lightweight Third Party Authentication (LTPA) security token from the Run As invocation Subject. This token is inserted in the Web Services Security header within the SOAP message as a binary security token. However, if the user name and password are specified in the basic authentication section of this panel, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token. It obtains the security token this way rather than obtaining it from the Run As Subject. Use this callback handler only when the web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client.

com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler
: This callback handler is used to create the X.509 certificate that is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore file and a key definition are required for this callback handler.

com.ibm.wsspi.wssecurity.auth.callback.PKCS7CallbackHandler
: This callback handler is used to create X.509 certificates that are encoded with the PKCS#7 format. The certificate is inserted in the Web Services Security header in the SOAP message as a binary security token. A keystore file is required for this callback handler. You must specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. For more information on configuring the collection certificate store, see "Configuring the collection certificate on the server or cell level" on page 973.

com.ibm.wsspi.wssecurity.auth.callback.PkiPathCallbackHandler
: This callback handler is used to create X.509 certificates that are encoded with the PkiPath format. The certificate is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore file is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used.

For an X.509 certificate token, you might specify the `com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler` implementation.

b. Optional: Select the **Use identity assertion** option. Select this option if you have identity assertion that is defined in the IBM extended deployment descriptor. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For example, WebSphere Application Server sends only the user name of the original caller for a user name token generator. For an X.509 token generator, the application server sends the original signer certification only.

c. Optional: Select the **Use RunAs identity** option. Select this option if the following conditions are true:
   - You have identity assertion defined in the IBM extended deployment descriptor.
   - You want to use the Run As identity instead of the initial caller identity for identity assertion for a downstream call.

d. Optional: Specify a basic authentication user ID and password in the **User ID** and **Password** fields. This entry specifies the user name and password that is passed to the constructors of the callback handler implementation. The basic authentication user ID and password are used if you specify one of the following default callback handler implementations that are provided by WebSphere Application Server:
   - com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
   - com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler
   - com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler

- com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler

e. Optional: Specify a keystore password and path. The keystore and its related information are necessary when the key or certificate is used for generating a token. For example, the keystore information is required if you select one of the following default callback handler implementations that are provided by WebSphere Application Server:

- com.ibm.wsspi.wssecurity.auth.callback.PKCS7CallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.PkiPathCallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler

The keystore files contain public and private keys, root certificate authority (CA) certificates, intermediate CA certificates, and so on. Keys that are retrieved from the keystore file are used to sign and validate or encrypt and decrypt messages or message parts. To retrieve a key from a keystore file, you must specify the keystore password, the keystore path, and the keystore type.

14. Select a keystore type from the **Type** field. WebSphere Application Server provides the following options:

JKS
Use this option if you are not using Java Cryptography Extensions (JCE) and if your keystore file uses the Java Keystore (JKS) format.

JCEKS
Use this option if you are using Java Cryptography Extensions.

JCERACFKS
Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

PKCS11KS (PKCS11)
Use this format if your keystore file uses the PKCS#11 file format. Key store files using this format might contain RSA keys on cryptographic hardware or might encrypt the keys that use cryptographic hardware to ensure protection.

PKCS12KS (PKCS12)
Use this option if your keystore file uses the PKCS#12 file format.

15. Click **OK** and then **Save** to save the configuration.
16. Click the name of your token generator configuration.
17. Under Additional properties, click **Callback handler** > **Keys**.
18. Click **New** to create a key configuration, click **Delete** to delete an existing configuration, or click the name of an existing key configuration to edit its settings. If you are creating a new configuration, enter a unique name for the key configuration in the **Key name** field. This name refers to the name of the key object that is stored within the keystore file.
19. Specify an alias for the key object in the **Key alias** field. Use the alias when the key locator searches for the key objects in the keystore.
20. Specify the password that is associated with the key in the **Key password** field.
21. Click **OK** and **Save** to save the configuration.

Results

You have configured the token generators at the server or the cell level.

What to do next

You must specify a similar token consumer configuration.

Token generator collection:

Use this page to view the token generators. The information is used on the generator side only to generate the security token.

To view this administrative console page for the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Generator Bindings, click **Token generators**.

To view this administrative console page for the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Generator Bindings, click **Token generators**.

Token generator name:

Specifies the name of the token generator configuration.

For example, the default X509 token generator names are either `gen_enctgen` for encrypting or `gen_signtgen` for signing. Or a custom token generator name might be `sig_tgen` for signing.

Token generator class name:

Specifies the name of the token generator implementation class.

This class must implement the `com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent` interface.

Token generator class name:

Specifies the name of the token generator implementation class.

The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side.

Token generator configuration settings:

Use this page to specify the information for the token generator. The information is used at the generator side only to generate the security token.

To view this administrative console page for the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Generator Bindings, click **Token generators** > ***token_generator_name*** or click **New** to create a new token generator.

To view this administrative console page for the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Generator Bindings, click **Token generators** > ***token_generator_name*** or click **New** to create a new token generator.
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.

3. Under Additional properties, you can access the token generator information for the following bindings:
   - For the Request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
4. Click **New** to create a new token generator or click the name of an existing token generator name to specify its settings.

To view this administrative console page for the application level, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, click **Web services: Client security bindings**.
4. Under Request generator (sender) binding, click **Edit custom**.
5. Under Additional properties, click **Token generators** > **New**.

Before specifying additional properties, specify a value in the **Token generator name** and the **Token generator class name** fields.

Token generator name:

Specifies the name of the token generator configuration.

For example, the default X509 token generator names are either `gen_enctgen` for encrypting or `gen_signtgen` for signing. Or, a custom token generator name might be `sig_tgen` for signing.

Token generator class name:

Specifies the name of the token generator implementation class.

This class must implement the `com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent` interface.

Token generator class name:

Specifies the name of the token generator implementation class.

Certificate path:

Specifies the certificate revocation list (CRL) that is used for generating a security token wrapped in a PKCS#7 token type with CRL.

When the token generator is not for a PKCS#7 token type, you must select **None**. When the token generator is for the PKCS#7 token type and you want to package CRL in the security token, select **Dedicated signing information** and specify the CRL for the collection certificate store.

You can specify a certificate store configuration for the following bindings on the following levels:

Table 183. Certificate path binding settings. The certificate is used for signing messages.

Binding name	Server level, cell level, or application level	Path
Default generator bindings	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Collection certificate store**.

Table 183. Certificate path binding settings (continued). The certificate is used for signing messages.

Binding name	Server level, cell level, or application level	Path
Default generator bindings	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Collection certificate store**.

Using the collection certificate store, you can configure a related certificate revocation list by clicking **Certificate revocation list** under Additional properties.

Add nonce:

Indicates whether nonce is included in the user name token for the token generator. *Nonce* is a unique cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens.

On the application level, if you select the **Add nonce** option, you can specify the following properties under Additional properties:

Table 184. Additional nonce properties. Nonce is used to add additional security to a message.

Property name	Default value	Explanation
com.ibm.ws.wssecurity.config.token. BasicAuth.Nonce.cacheTimeout	600 seconds	Specifies the timeout value, in seconds, for the nonce value that is cached on the server.
com.ibm.ws.wssecurity.config.token. BasicAuth.Nonce.clockSkew	0 seconds	Specifies the time, in seconds, before the nonce time stamp expires.
com.ibm.ws.wssecurity.config.token. BasicAuth.Nonce.maxAge	300 seconds	Specifies the clock skew value, in seconds, to consider when the application server checks the timeliness of the message.

These properties are available on the administrative console at the cell and server level. However, on the application level, you can configure the properties under Additional properties.

This option is displayed on the cell, server, and application levels. This option is valid only when the generated token type is a user name token.

Add timestamp:

Specifies whether to insert the time stamp into the user name token.

This option is displayed on the cell, server, and application levels. This option is valid only when the generated token type is a user name token.

Value type local name:

Specifies the local name of the value type for the generated token.

For a user name token and an X.509 certificate security token, this product provides predefined value types. When you specify the following local names, you do not need to specify the Uniform Resource Identifier (URI) of value type.

Username token
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken

X509 certificate token
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3

X509 certificates in a PKIPath
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1

A list of X509 certificates and CRLs in a PKCS#7
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

Lightweight Third Party Authentication (LTPA)

LTPA_PROPAGATION

Important: For LTPA, the value type local name is `LTPA`. If you enter `LTPA` for the local name, you must specify the `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` URI value in the Value type URI field as well. For LTPA token propagation, the value type local name is `LTPA_PROPAGATION`. If you enter `LTPA_PROPAGATION` for the local name, you must specify the `http://www.ibm.com/websphere/appserver/tokentype` URI value in the Value type URI field as well. For the other predefined value types (Username token, X509 certificate token, X509 certificates in a PKIPath, and a list of X509 certificates and CRLs in a PKCS#7), the value for the local name field begins with `http://`. For example, if you are specifying the user name token for the value type, enter `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken` in the Value type local name field and then you do not need to enter a value in the Value type URI field.

When you specify a custom value type for custom tokens, you can specify the local name and the URI of the quality name (QName) of the value type. For example, you might specify `Custom` for the local name and `http://www.ibm.com/custom` for the URI.

Value type URI:

Specifies the namespace URI of the value type for the generated token.

When you specify the token generator for the user name token or the X.509 certificate security token, you do not need to specify this option. If you want to specify another token, specify the URI of the QName of the value type.

The application server provides the following predefined value type URIs:
- For the LTPA token: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
- For the LTPA token propagation: http://www.ibm.com/websphere/appserver/tokentype

Algorithm URI collection:

Use this page to view a list of uniform resource identifier (URI) algorithms for XML digital signature or XML encryption that are mapped to an algorithm factory engine class. With algorithm mappings, service providers can use other cryptographic algorithms for digest value calculation, digital signature signing and verification, data encryption and decryption, and key encryption and decryption.

To view this administrative console page on the cell level, complete the following steps:
1. Click **Security > JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Algorithm mappings**.
3. Click on an algorithm mapping name.
4. Under Additional properties, click **Algorithm URI**.

To view administrative console page on the server level, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Algorithm mappings**.
4. Click on an algorithm mapping name.
5. Under Additional properties, click **Algorithm URI**.

Algorithm URI:

Specifies the algorithm uniform resource identifier (URI) for the specified algorithm type.

Algorithm type:

Specifies the algorithm type.

Algorithm URI configuration settings:

Use this page to specify the algorithm uniform resource identifier (URI) and its usage type.

This product supports the following algorithm URI types:

Message digest
: Specifies the algorithm URI that is used for digest value calculation.

Signature
: Specifies the algorithm URI that is used for digital signature, including both signature and signing verification.

Data encryption
: Specifies the algorithm URI that is used for both encrypting and decrypting data.

Key encryption
: Specifies the algorithm URI that is used for encrypting and decrypting the encryption key.

If the URI is used for multiple usage types, then you must define a mapping of the URI to each usage type.

To view this administrative console page on the server level, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** *server_name*.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Algorithm mappings**.

   Note: The Algorithm mappings feature is not supported when the **Use the Federal Information Processing Standard (FIPS)** option has been selected on the SSL certificate and key management panel of the administrative console. When this option is selected, the **New** button in the Algorithm mappings panel is not available.
4. Click **New**.
5. Under Additional properties, click **Algorithm URI >** ***algorithm_URI_name***.

To view the administrative console page on the cell level:

1. Click **Security > JAX-WS and JAX-RPC security runtime**, or **Services > JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Algorithm mappings**.
3. Click **New**.
4. Under Additional properties, click **Algorithm URI >** ***algorithm_URI_name***.

Algorithm URI:

Specifies the algorithm uniform resource identifier (URI) for the specified algorithm type.

The algorithm URI that is defined on this page is available to the various binding configurations. For example, if you specify an algorithm URI and select **Signature** from the Algorithm type field, the URI displays in the Signature method field on the signing information panel.

Algorithm type:

Specifies the type of algorithm that is specified in the Algorithm URI field.

The following types of algorithms are supported by this product. The following list shows where configurations that are specified on this panel are displayed for a binding configuration:

Table 185. Algorithm types. The algorithm types in the table are supported by the product.

Algorithm type	Explanation	Location of the configuration
Signature	This algorithm type is used for digital signatures.	This configuration displays in the Signature method field on the Signing information panel. For information on how to access the Signing information panel, see the help topic Signing information configuration settings.
Digest value calculation (message digest)	This algorithm type is used for calculating the digest value.	This configuration displays in the Digest method algorithm field on the Part references panel. For information on how to access the Part references panel, see the help topic Part reference configuration settings.
Data encryption	This algorithm type is used for encrypting data.	This configuration displays in the Data encryption algorithm field on the Encryption information panel. For information on how to access the Encryption information panel, see the help topic Encryption information configuration settings: Message parts.
Key encryption	This algorithm type is used for encrypting the key that is used for data encryption.	This configuration displays in the Key encryption algorithm field on the Encryption information panel. For information on how to access the Encryption information panel, see the help topic Encryption information configuration settings: Message parts.

The actual implementation of the algorithm is done in the implementation class for the engine factory.

Algorithm mapping collection:

You can view a list of custom uniform resource identifier (URI) algorithms for digest value calculation, signature, key encryption, and data encryption. The application server maps these algorithms to an implementation of the algorithm factory engine interface. With algorithm mappings, service providers can extend the cryptographic algorithms for XML digital signature and XML encryption.

To view this administrative console page on the cell level, complete the following steps:

1. Click **Security > JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Algorithm mappings**.

To view this administrative console page on the server level, complete the following steps:

1. Click **Servers > Server Types > WebSphere application servers >** ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

3. Under Additional properties, click **Algorithm mappings**.

Algorithm factory engine class:

Specifies the custom class that implements the factory engine implementation class for the algorithm factory engine.

The implementation class for the factory engine implements the cryptographic functions of the defined uniform resource identifier (URI).

Note: The Algorithm mappings feature is not supported when the **Use the Federal Information Processing Standard (FIPS) algorithms** option has been selected on the Global security panel of the administrative console. When this option is selected, the **New** button in the Algorithm mappings panel is not available.

Algorithm mapping configuration settings:

Use this page to view a list of custom uniform resource identifier (URI) algorithms for digest value calculation, signature, key encryption, and data encryption. The application server maps these algorithms to an implementation of the algorithm factory engine interface. With algorithm mappings, service providers can extend the cryptographic algorithms for XML digital signature and XML encryption.

To view this administrative console page on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Algorithm mappings*algorithm_factory_engine_class_name***.

   Note: The Algorithm mappings feature is not supported when the **Use the Federal Information Processing Standard (FIPS)** option has been selected on the SSL certificate and key management panel of the administrative console. When this option is selected, the **New** button in the Algorithm mappings panel is not available.
4. Click **New**.

To view this administrative console page on the cell level:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**, or **Services** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Algorithm mappings** > ***algorithm_factory_engine_class_name***.
3. Click **New**.

Algorithm factory engine class:

Specifies the custom class that implements the factory engine interface.

To use this algorithm mapping feature, you must specify a custom algorithm class in the Algorithm factory engine class field for digital signature, data encryption, digest value calculation, and key encryption. The algorithm factory engine provides a plug-in point for service providers to provide their implementation for digest value calculation, digital signature, key encryption, and data encryption that is based on a specified algorithm uniform resource identifier (URI). By clicking **Algorithm URI** under Additional properties, you can specify the algorithm URI and its usage type. This product supports the following algorithm types:

Message digest
Specifies the algorithm URI that is used for digest value calculation.

Signature
Specifies the algorithm URI that is used for digital signatures including both signing and signature verification.

Data encryption
Specifies the algorithm URI that is used for both encrypting and decrypting data.

Key encryption
Specifies the algorithm URI that is used for both encrypting and decrypting the encryption key.

If the URI is used for multiple usage types, then you must define a mapping of the URI to each usage type. The actual implementation of the algorithm is provided by the custom class that implements the factory engine interface. For more information, refer to the information center documentation on how to implement a factory class.

By clicking **Properties** under Additional properties, you can specify name-value pair properties for the factory class.

Default bindings and security runtime properties:

Use this page to specify the configuration on the cell level in a WebSphere Application Server, Network Deployment environment. In addition, use this page to define the default generator bindings, default consumer bindings, and additional properties such as key locators, the collection certificate store, trust anchors, trusted ID evaluators, algorithm mappings, and login mappings.

Displayed options and the panel title depend on your server configuration and version.

To view this administrative console page for the cell level, Click **Security** > **JAX-WS and JAX-RPC security runtime**.

To view this administrative console page for the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

Nonce is a unique cryptographic number embedded in a message to help stop repeated, unauthorized attacks of user name tokens. In a WebSphere Application Server, Network Deployment environment, you must specify values for the Nonce cache timeout, the Nonce maximum age, and the Nonce clock skew fields for the cell level.

The default binding configuration provides a central location where reusable binding information is defined. The application binding file can reference the information in the default binding configuration.

Nonce cache timeout:

Specifies the timeout value, in seconds, for the nonce value that is cached on the server. Nonce is a randomly generated value.

The Nonce cache timeout field is required for the cell level.

The maximum value for the Nonce maximum age field cannot exceed the number of seconds that is specified for this Nonce cache timeout field. If you make changes to the field value, you must restart WebSphere Application Server for the changes to take effect.

Information	Value
Default	600 seconds
Minimum	300 seconds

Nonce maximum age:

Specifies the time, in seconds, before the nonce time stamp expires. Nonce is a randomly generated value.

The value that is specified in this cell-level field is the maximum value that you can specify for the Nonce maximum age field for the server level.

The Nonce maximum age field is required for the cell level.

Information	Value
Default	300 seconds
Range	300 to the Nonce cache timeout value in seconds

Nonce clock skew:

Specifies the clock skew value, in seconds, to consider when WebSphere Application Server checks the timeliness of the message. Nonce is a randomly generated value.

The Nonce clock skew field is required for the cell level.

Information	Value
Default	0 seconds
Range	0 to the Nonce maximum age value, in seconds

Custom properties:

The linked Properties panel specifies additional properties for the security runtime configuration.

Configuring token consumers using JAX-RPC to protect message authenticity at the server or cell level:

The token consumer on the server or cell level is used to specify the information that is needed to process the security token if it is not defined at the application level.

Before you begin

You need to understand that the keystore/alias information that you provide for the generator, and the keystore/alias information that you provide for the consumer are used for different purposes. The main difference applies to the Alias for an X.509 callback handler.

When used in association with an encryption consumer, the alias supplied for the consumer is used retrieve the private key to decrypt the message. A password is required. When associated with a signature consumer, the alias supplied for the consumer is used strictly to retrieve the public key that is used to resolve an X.509 certificate that is not passed in the SOAP security header as a BinarySecurityToken. A password is not required.

About this task

WebSphere Application Server provides default values for bindings. You must modify the defaults for a production environment.

You can configure the token consumers on the server level and the cell level. In the following steps, use the first step to access the server-level default bindings and use the second step to access the cell-level bindings.

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

      Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Default consumer bindings, click **Token consumers**.
4. Click **New** to create a token consumer configuration, click **Delete** to delete an existing configuration, or click the name of an existing token consumer configuration to edit its settings. If you are creating a new configuration, enter a unique name for the token consumer configuration in the **Token consumer name** field. For example, you might specify `sig_tcon`. This field specifies the name of the token consumer element.
5. Specify a class name in the Token consumer class name field. The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to validate (authenticate) the security token on the consumer side.

   Restriction: The com.ibm.wsspi.wssecurity.token.TokenConsumingComponent interface is not used with JAX-WS web services. If you are using JAX-RPC web services, this interface is still valid.

   The token consumer class name must be similar to the token generator class name.

   For example, if your application requires an X.509 certificate token consumer, you can specify the com.ibm.wsspi.wssecurity.token.X509TokenGenerator class name on the Token generator panel and the com.ibm.wsspi.wssecurity.token.X509TokenConsumer class name in this field. WebSphere Application Server provides the following default token consumer class implementations:

   com.ibm.wsspi.wssecurity.token.UsernameTokenConsumer
   This implementation integrates a user name token.

   com.ibm.wsspi.wssecurity.token.X509TokenConsumer
   This implementation integrates an X.509 certificate token.

   com.ibm.wsspi.wssecurity.token.LTPATokenConsumer
   This implementation integrates a Lightweight Third Party Authentication (LTPA) token.

   com.ibm.wsspi.wssecurity.token.IDAssertionUsernameTokenConsumer
   This implementation integrates an IDAssertionUsername token.

   A corresponding token generator class does not exist for this implementation.
6. Select a certificate path option. The certificate path specifies the certificate revocation list (CRL) that is used for generating a security token wrapped in a PKCS#7 with a CRL. WebSphere Application Server provides the following certificate path options:

   None If you select this option, the certificate path is not specified.

   Trust any
   If you select this option, any certificate is trusted. When the received token is consumed, the certificate path validation is not processed.

   Dedicated signing information
   If you select this option, you can specify a trust anchor and a certificate store. When you select the trust anchor or the certificate store of a trusted certificate, you must configure the

collection certificate store before setting the certificate path. To define a collection certificate store on the server or cell level, see “Configuring the collection certificate on the server or cell level” on page 973.

a. Select a trust anchor in the Trust anchor field. WebSphere Application Server provides two sample trust anchors. However, it is recommended that you configure your own trust anchors for a production environment. For information on configuring a trust anchor, see “Configuring trust anchors on the server or cell level” on page 960.

b. Select a collection certificate store in the Certificate store field. WebSphere Application Server provides a sample collection certificate store. If you select **None**, the collection certificate store is not specified. For information on specifying a list of certificate stores that contain untrusted, intermediary certificate files awaiting validation, see “Configuring trusted ID evaluators on the server or cell level” on page 975.

7. Select a trusted ID evaluator from the Trusted ID evaluation reference field. This field specifies a reference to the Trusted ID evaluator class name that is defined in Trusted ID evaluators panel. The trusted ID evaluator is used for evaluating whether the received ID is trusted. If you select **None**, the trusted ID evaluator is not referenced in this token consumer configuration. To configure a trusted ID evaluator, see “Configuring trusted ID evaluators on the server or cell level” on page 975.
8. Select the **Verify nonce** option if a nonce is included in a user name token on the generator side. Nonce is a unique cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens. The **Verify nonce** option is available if you specify a user name token for the token consumer and nonce is added to the user name token on the generator side.
9. Select the **Verify timestamp** option if a time stamp is included in the user name token on the generator side. The **Verify Timestamp** option is available if you specify a user name token for the token consumer and a time stamp is added to the user name token on the generator side.
10. Specify the local name of the value type for the integrated token. This entry specifies the local name of the value type for a security token that is referenced by the key identifier. This attribute is valid when **Key identifier** is selected as the key information type. To specify the key information type, see “Configuring the key information for the consumer binding using JAX-RPC on the server or cell level” on page 911. WebSphere Application Server has predefined value type local names for the user name token and the X.509 certificate security token. Enter one of the following local names for the user name token and the X.509 certificate security token. When you specify the following local names, you do not need to specify the URI of the value type:

Username token
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken

X.509 certificate token
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3

X.509 certificates in a PKIPath
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1

A list of X.509 certificates and CRLs in a PKCS#7
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

Note: To specify Lightweight Third Party Authentication (LTPA) or token propagation (LTPA_PROPAGATION), you must specify both the value type local name and the Uniform Resource Identifier (URI). For LTPA, specify `LTPA` for the local name and `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` for the URI. For LTPA token propagation, specify `LTPA_PROPAGATION` for the local name and `http://www.ibm.com/websphere/appserver/tokentype` for the URI.

For example, when an X.509 certificate token is specified, you can use `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3` for the local name. When you

specify the local name of another token, you must specify a value type Qname. For example: uri=`http://www.ibm.com/custom`, localName=`CustomToken`

11. Specify the value type uniform resource identifier (URI) in the URI field. This entry specifies the namespace URI of the value type for a security token that is referenced by the key identifier. This attribute is valid when **Key identifier** is selected as the key information type on the Key information panel for the default generator. When you specify the token consumer for the user name token or an X.509 certificate security token, you do not need to specify this option. If you specify another token, you need to specify the URI of the QName for the value type.
12. Click **OK** and then **Save** to save the configuration. After saving the token generator configuration, you can specify a JAAS configuration for your token consumer.
13. Click the name of your token generator configuration.
14. Under Additional properties, click **JAAS configuration**.
15. Select a JAAS configuration from the JAAS configuration name field.

    The field specifies the name of the JAAS system for application login configuration. You can specify additional JAAS system and application configurations by clicking **Security** > **Global security**. Expand Java Authentication and Authorization Service, then click **Application logins** > **New** or **System logins** > **New**.

    For more information on the JAAS configurations, see “JAAS configuration settings” on page 885. Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module. WebSphere Application Server provides the following predefined JAAS configurations:

    ClientContainer
    : This selection specifies the login configuration that is used by the client container applications. The configuration uses the CallbackHandler application programming interface (API) that is defined in the deployment descriptor for the client container. To modify this configuration, see the JAAS configuration panel for application logins.

    WSLogin
    : This selection specifies whether all of the applications can use the WSLogin configuration to perform authentication for the security run time. To modify this configuration, see the JAAS configuration panel for application logins.

    DefaultPrincipalMapping
    : This selection specifies the login configuration that is used by Java 2 Connectors (J2C) to map users to principals that are defined in the J2C authentication data entries. To modify this configuration, see the JAAS configuration panel for application logins.

    system.wssecurity.IDAssertion
    : This selection enables a Version 5.*x* application to use identity assertion to map a user name to a WebSphere Application Server credential principal. To modify this configuration, see the JAAS configuration panel for system logins.

    system.wssecurity.Signature
    : This selection enables a Version 5.*x* application to map a distinguished name (DN) in a signed certificate to a WebSphere Application Server credential principal. To modify this configuration, see the JAAS configuration panel for system logins.

    system.LTPA_WEB
    : This selection processes login requests that are used by the web container such as servlets and JavaServer Pages (JSP) files. To modify this configuration, see the JAAS configuration panel for system logins.

    system.WEB_INBOUND
    : This selection handles login requests for web applications, which include servlets and

JavaServer Pages (JSP) files. This login configuration is used by WebSphere Application Server Version 5.1.1. To modify this configuration, see the JAAS configuration panel for system logins.

system.RMI_INBOUND
This selection handles logins for inbound Remote Method Invocation (RMI) requests. This login configuration is used by WebSphere Application Server Version 5.1.1. To modify this configuration, see the JAAS configuration panel for system logins.

system.DEFAULT
This selection handles the logins for inbound requests that are made by internal authentications and most of the other protocols except web applications and RMI requests. This login configuration is used by WebSphere Application Server Version 5.1.1. To modify this configuration, see the JAAS configuration panel for system logins.

system.RMI_OUTBOUND
This selection processes RMI requests that are sent outbound to another server when the `com.ibm.CSIOutboundPropagationEnabled` property is `true`. This property is set in the CSIv2 authentication panel. To access the panel, click **Security** > **Global security**. Under Authentication, expand **RMI/IIOP security** and click **CSIv2 outbound authentication**. To set the `com.ibm.CSIOutboundPropagationEnabled` property, select **Security attribute propagation**. To modify this JAAS login configuration, see the JAAS - System logins panel.

system.wssecurity.X509BST
This section verifies an X.509 binary security token (BST) by checking the validity of the certificate and the certificate path. To modify this configuration, see the JAAS configuration panel for system logins.

system.wssecurity.PKCS7
This selection verifies an X.509 certificate with a certificate revocation list in a PKCS7 object. To modify this configuration, see the JAAS configuration panel for system logins.

system.wssecurity.PkiPath
This section verifies an X.509 certificate with a public key infrastructure (PKI) path. To modify this configuration, see the JAAS configuration panel for system logins.

system.wssecurity.UsernameToken
This selection verifies the basic authentication (user name and password) data. To modify this configuration, see the JAAS configuration panel for system logins.

system.wssecurity.IDAssertionUsernameToken
This selection enables Versions 6 and later applications to use identity assertion to map a user name to a WebSphere Application Server credential principal. To modify this configuration, see the JAAS configuration panel for system logins.

system.WSS_INBOUND
This selection specifies the login configuration for inbound or consumer requests for security token propagation using Web Services Security. To modify this configuration, see the JAAS configuration panel for system logins.

system.WSS_OUTBOUND
This selection specifies the login configuration for outbound or generator requests for security token propagation using Web Services Security. To modify this configuration, see the JAAS configuration panel for system logins.

None With this selection, you do not specify a JAAS login configuration.

16. Click **OK** and then **Save** to save the configuration.

Results

You have configured the token consumer at the server or cell level.

What to do next

You must specify a similar token generator configuration for the server or cell level.

Token consumer collection:

Use this page to view the token consumer. The information is used on the consumer side only to process the security token.

To view this administrative console page for the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Default Consumer Bindings, click **Token consumers**.

To view this administrative console page for the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Default Generator Bindings, click **Token consumers**.

To view this administrative console page for Version 6.*x* and later applications on the application level, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules** > ***URI_name*** .
3. Under Web Services Security Properties, you can access the signing information for the following bindings:
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**. Under Required properties, click **Token consumers**.
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Token consumers**.

Token consumer name:

Specifies the name of the token consumer configuration.

For example, the default X509 token consumer names can be either `con_enctcon` for encrypting or `con_signtcon` for signing. Or a custom token consumer name might be `sig_tcon` for signing.

Token consumer class name:

Specifies the name of the token consumer implementation class.

This class must implement the `com.ibm.wsspi.wssecurity.token.TokenConsumerComponent` interface.

Token consumer class name:

Specifies the name of the token consumer implementation class.

The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to validate (authenticate) the security token on the consumer side.

Token consumer configuration settings:

Use this page to specify the information for the token consumer. The information is used at the consumer side only to process the security token.

To view this administrative console page for the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Consumer Bindings, click **Token consumers** > ***token_consumer_name*** or click **New** to create a new token consumer.

To view this administrative console page for the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** .
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under JAX-RPC Default Consumer Bindings, click **Token consumers** > ***token_consumer_name*** or click **New** to create a new token consumer.

To view this administrative console page for Version 6 and later applications on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access the signing information for the following bindings:
   - For the Response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**. Under Required properties, click **Token consumers**.
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**. Under Required properties, click **Token consumers**.
4. Click **New** to specify a new configuration or click the name of an existing configuration to modify its settings.

Before specifying additional properties, specify a value in the Token consumer name, the Token consumer class name, and the Value type local name fields.

Token consumer name:

Specifies the name of the token consumer configuration.

For example, the default X509 token consumer names are either `con_enctcon` for encrypting or `con_signtcon` for signing. Or a custom, the token consumer name might be `sig_tcon` for signing.

Token consumer class name:

Specifies the name of the token consumer implementation class.

This class must implement the `com.ibm.wsspi.wssecurity.token.TokenConsumerComponent` interface.

Token consumer class name:

Specifies the name of the token consumer implementation class.

The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to validate (authenticate) the security token on the consumer side.

Part reference:

Specifies a reference to the name of the security token that is defined in the deployment descriptor.

On the application level, when the security token is not specified in the deployment descriptor, the Part reference field is not displayed.

Certificate path:

Specifies the trust anchor and the certificate store.

You can select the following options:

None If you select this option, the certificate path is not specified.

Trust any
If you select this option, any certificate is trusted. When the received token is incorporated, the certificate path validation is not processed.

Dedicated signing information
If you select this option, you can specify the trust anchor and the certificate store. When you select the trust anchor or the certificate store of a trusted certificate, you must configure the collection certificate store before setting the certificate path.

Trust anchor

You can specify a trust anchor for the following bindings on the following levels:

Table 186. Trust anchor binding settings. The trust anchor is used for signing messages.

Binding name	Server level, cell level, or application level	Path
Default consumer binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Trust anchors**.
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Trust anchors**.

Certificate store

You can specify a certificate path configuration for the following bindings on the following levels:

Table 187. Certificate store binding settings. The certificate is used for signing messages.

Binding name	Server level, cell level, or application level	Path
Default consumer binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Collection certificate store**.
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Collection certificate store**.

Trusted ID evaluator reference:

Specifies the reference to the Trusted ID evaluator class name that is defined in the Trusted ID evaluators panel. The trusted ID evaluator is used for determining whether the received ID is trusted.

You can select the following options:

None If you select this option, the trusted ID evaluator is not specified.

Existing evaluator definition
If you select this option, you can select one of the configured trusted ID evaluators.

You can specify a certificate path configuration for the following bindings on the following levels:

Table 188. Trusted ID evaluator bindings settings. The trusted ID evaluator is used to determine if a received ID is trusted.

Binding name	Server level, cell level, or application level	Path
Default consumer binding	Cell level	1. Click **Security** > **JAX-WS and JAX-RPC security runtime**. 2. Under Additional properties, click **Trusted ID evaluators**.
Default consumer binding	Server level	1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***. 2. Under Security, click **JAX-WS and JAX-RPC security runtime**. **Note:** In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**. 3. Under Additional properties, click **Trusted ID evaluators**.

Binding evaluator definition
If you select this option, you can specify a new trusted ID evaluator and its class name.

When you select a trusted ID evaluator reference, you must configure the trusted ID evaluators before setting the token consumer.

The Trusted ID evaluator field is displayed in the default binding configuration and the application server binding configuration.

Verify nonce:

Specifies whether the nonce of the user name token is verified.

This option is displayed on the cell, server, and application levels. This option is valid only when the type of incorporated token is the user name token.

Verify timestamp:

Specifies whether the time stamp of user name token is verified.

This option is displayed on the cell, server, and application levels. This option is valid only when the type of incorporated token is the user name token.

Value type local name:

Specifies the local name of value type for the consumed token.

This product has predefined value type local names for the user name token and the X.509 certificate security token. Use the following local names for the user name token and the X.509 certificate security token. When you specify the following local names, you do not need to specify the Uniform Resource Identifier (URI) of the value type:

Username token
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken

X509 certificate token
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3

 # X509 certificates in a PKIPath
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1

A list of X509 certificates and CRLs in a PKCS#7
: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

Lightweight Third Party Authentication (LTPA)

LTPA_PROPAGATION

Important: For Lightweight Third Party Authentication (LTPA), the value type local name is `LTPA`. If you enter `LTPA` for the local name, you must specify the `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` URI value in the Value type URI field as well. For LTPA token propagation, the value type local name is `LTPA_PROPAGATION`. If you enter `LTPA_PROPAGATION` for the local name, you must specify the `http://www.ibm.com/websphere/appserver/tokentype` URI value in the Value type URI field as well. For the other predefined value types (Username token, X509 certificate token, X509 certificates in a PKIPath, and a list of X509 certificates and CRLs in a PKCS#7), the value for the local name field begins with `http://`. For example, if you are specifying the username token for the value type, enter `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken` in the value type local name field and then you do not need to enter a value in the value type URI field.

When you specify a custom value type for custom tokens, you can specify the local name and the URI of the Quality name (QName) of the value type. For example, you might specify `Custom` for the local name and `http://www.ibm.com/custom` for the URI.

Value type URI:

Specifies the namespace URI of the value type for the integrated token.

When you specify the token consumer for the user name token or the X.509 certificate security token, you do not need to specify this option. If you want to specify another token, specify the URI of the QName for the value type.

The application server provides the following predefined value type URIs:
- For the LTPA token: `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`
- For the LTPA token propagation: `http://www.ibm.com/websphere/appserver/tokentype`

Configuring Web Services Security using JAX-RPC at the platform level:

In the platform configuration, general properties and additional properties can be specified, and the default binding is included. You can configure security for web services at a platform level with a variety of tasks including configuring key locators, trust anchors, and the collection certificate at the generator, consumer binding, and sever levels.

Before you begin

best-practices: IBM WebSphere Application Server supports the Java API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model. JAX-WS is the next generation web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, development of web services and clients is simplified through support of a standards-based annotations model. Although the JAX-RPC programming model and applications are still supported, take advantage of the easy-to-implement JAX-WS programming model to develop new web services applications and clients.

Besides the application-level constraints, there is a cell-level and server-level Web Services Security (WSS) configuration called a *platform-level configuration*:
- These configurations are global for all applications and include some configurations only for WebSphere Application Server Version 5.*x* applications and some only for version 6.0.*x* applications.
- You can use the default binding as an application-level binding configuration so that applications do not have to define the binding in the application. There is only one set of default bindings that can be shared by multiple applications. This set is only available for WebSphere Application Server Version 6.*x* applications.

Therefore, binding configuration files can be specified at these levels: application, server, and cell. Each binding configuration overrides the next higher one. For any deployed application, the nearest configuration binding is applied. The visibility scope of the binding depends on where the file is located. If the binding is defined in an application, its visibility is scoped to that particular application. If it is located at the server level, the visibility scope is all applications that are deployed on that server. For WebSphere Application Server, Network Deployment, if it is located at the cell level, the visibility scope is all applications deployed on all servers of the cell.

About this task

To ensure Web Services Security at the platform level, you can configure:
- A nonce on the server or cell level
- The key locator for the generator or consumer binding on the application level, server level, or cell level
- Trust anchors for the generator or consumer binding on the application level, server level, or cell level
- The collection certificate store for the generator or consumer binding on the application level, server level or cell level
- Trusted ID evaluators on the server or cell level
- Hardware cryptographic devices for Web Services Security

- The `rrdSecurity.props` property file

Procedure

- To configure a nonce on the server or cell level, see the steps in "Configuring a nonce on the server or cell level"
- To configure the key locator for the generator binding on the application level, see the steps in "Configuring the key locator using JAX-RPC for the generator binding on the application level" on page 943
- To configure the key locator for the consumer binding on the application level, see the steps in "Configuring the key locator using JAX-RPC for the consumer binding on the application level" on page 950
- To configure the key locator on the server or cell level, see the steps in "Configuring the key locator using JAX-RPC on the server or cell level" on page 952
- To configure trust anchors for the generator binding on the application level, see the steps in "Configuring trust anchors for the generator binding on the application level" on page 954
- To configure trust anchors for the consumer binding on the application level, see the steps in "Configuring trust anchors for the consumer binding on the application level" on page 959
- To configure trust anchors on the server or cell level, see the steps in "Configuring trust anchors on the server or cell level" on page 960
- To configure the collection certificate store for the generator binding on the application level, see the steps in "Configuring the collection certificate store for the generator binding on the application level" on page 961
- To configure the collection certificate store for the consumer binding on the application level, see the steps in "Configuring the collection certificate store for the consumer binding on the application level" on page 971
- To configure the collection certificate on the server or cell level, see the steps in "Configuring the collection certificate on the server or cell level" on page 973
- To configure trusted ID evaluators on the server or cell level, see the steps in "Configuring trusted ID evaluators on the server or cell level" on page 975
- To enable hardware cryptographic devices for Web Services Security, see the steps in "Enabling hardware cryptographic devices for Web Services Security" on page 980
- To work with the `rrdSecurity.props` file, see "rrdSecurity.props file" on page 978

Results

By completing these steps, you have configured Web Services Security at the platform level.

Configuring a nonce on the server or cell level:

You can configure nonce for the server or cell by using the WebSphere Application Server administrative console.

About this task

Nonce is a randomly generated, cryptographic token that is used to prevent replay attacks of user name tokens that are used with SOAP messages. Typically, nonce is used with the user name token.

You can configure nonce at the application level, the server level, and the cell level. However, you must consider the order of precedence.

The following list shows the order of precedence:

1. Application level

The application level settings for the nonce maximum age and nonce clock skew fields are specified through the additional properties.
2. Server level
3. Cell level

If you configure nonce on the application level and the server level, the values that are specified for the application level take precedence over the values that are specified for the server level. Likewise, the values that are specified for the application level take precedence over the values that are specified for the server level and the cell level. In the WebSphere Application Server, Network Deployment environment, the Nonce cache timeout, Nonce maximum age, and Nonce clock skew fields are required to use nonce effectively. However, these fields are optional on the server level.

You can configure a nonce on the server level and the cell level. In the following steps, use the first step to access the server-level default bindings and use the second step to access the cell-level bindings.

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers > Server Types > WebSphere application servers > *server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

      Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security > Web services** to access the default bindings on the cell level.
3. Specify a value, in seconds, for the **Nonce cache timeout** field. The value that is specified for the **Nonce cache timeout** field indicates how long the nonce remains cached before it is discarded. You must specify a minimum of 300 seconds. However, if you do not specify a value, the default is 600 seconds. This field is optional on the server level, but required on the cell level.
4. Specify a value, in seconds, for the **Nonce maximum age** field. The value that is specified for the **Nonce maximum age** field indicates how long the nonce is valid. You must specify a minimum of 300 seconds, but the value cannot exceed the number of seconds that is specified for the **Nonce cache timeout** field. If you do not specify a value, the default is 300 seconds.

   In a WebSphere Application Server, Network Deployment environment, this field is optional on the server level, but it is required on the cell level.
5. Specify a value, in seconds, for the **Nonce clock skew** field. The value that is specified for the **Nonce clock skew** field specifies the amount of time, in seconds, to consider when the message receiver checks the freshness of the value. Consider the following information when you set this value:
   - Difference in time between the message sender and the message receiver, if the clocks are not synchronized.
   - Time that is needed to encrypt and transmit the message.
   - Time that is needed to get through network congestion.

   At a minimum, you must specify 0 seconds in this field. However, the maximum value cannot exceed the number of seconds indicated in the Nonce maximum age field. If you do not specify a value, the default is 0 seconds. This field is optional on the server level, but required on the cell level.
6. Optional: For WebSphere Application Server, Network Deployment only, select **Distribute nonce caching**. This option enables you to distribute the caching for a nonce using a Data Replication Service (DRS). In previous releases of WebSphere Application Server, the nonce was cached locally. By selecting this option, the nonce is propagated to other servers in your environment. However, the nonce might be subject to a one-second delay in propagation and subject to any network congestion.
7. Enable the dynamic cache service for each one of the application servers in your cluster. To access the dynamic cache service through the administrative console, complete the following steps:
   a. Click **Servers > Server Types > WebSphere application servers > *server_name***.

   b. Under Container settings, click **Container services > Dynamic cache service**.
   c. Confirm that the **Enable service at server startup** option is selected.
8. Specify the number of replication domains. To specify the number of replicas, click **Environment > Replication domains**. In a WebSphere Application Server, Network Deployment environment, the **Entire domain** option for the number of replicas is recommended.
9. Restart the server. If you change the nonce cache timeout value and do not restart the server, the change is not recognized by the server.

Distributing nonce caching to servers in a cluster:

Distributed nonce caching enables you to distribute the cache for a nonce to different servers in a cluster.

Before you begin

Before configuring distributed nonce caching, configure cache replication.

For more information, read about configuring cache replication.

Important: When you configure the cache replication, do not use the default value of a single replica for the Number of replicas for dynamic cache replication domains. Instead, use a full group replica for any replication domains that you configure for dynamic cache. If you cannot select the option, verify your cache replication configuration.

About this task

In previous releases of WebSphere Application Server, the nonce was cached locally. To use this feature, you must complete the following actions:

Procedure

1. Verify that you created an appropriate domain setting when you form a cluster.

   For more information, read about creating clusters.
2. Verify that replication domain is properly secured. The nonce cache is crucial to the integrity of the nonce validation process. If the nonce cache is compromised, then you cannot trust the result of the validation process.
3. In the administrative console for the cell level, set the Distribute nonce caching option by enabling the distributed cache option in the Security cache panel. You can enable the option by completing the following steps:
   a. Click **Services** > **Security cache**
   b. Click the check box to select the **Enable distributed caching** option.
4. Verify that the dynamic cache service is enabled for each one of the application servers in your cluster. To access the dynamic cache service through the administrative console, complete the following steps:
   a. Click **Servers > Server Types > WebSphere application servers >** ***server_name***.
   b. Under Container settings, click **Container services > Dynamic cache service**.
   c. Confirm that the **Enable service at server startup** option is selected.
5. In the administrative console for the server level, select the **Distribute nonce caching** option. You can enable the option by completing the following steps:
   a. Click **Security > Web services**.
   b. Select the **Distribute nonce caching** option.
6. Restart the servers within your cluster.

Results

When you select the **Distribute nonce caching** option in the administrative console, the nonce is propagated to other servers in your environment. However, the nonce might be subject to a one-second delay in propagation and subject to any network congestion.

What to do next

For more information on distributed nonce caching, see "Web Services Security enhancements" on page 197.

Configuring the key locator using JAX-RPC for the generator binding on the application level:

The key locator information for the default generator specifies which key locator implementation is used to locate the key to be used for signature and encryption information. The key locator information for the generator specifies which key locator implementation is used to locate the key to be used for signature validation or encryption.

About this task

WebSphere Application Server provides default values for the bindings. However, you must modify the defaults for a production environment.

Complete the following steps to configure the key locator for the generator binding on the application level:

Procedure

1. Locate the encryption information configuration panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security Properties you can access the key information for the request generator and response generator bindings.
      - For the request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
      - For the response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   d. Under Additional properties, click **Key locators**.
   e. Click **New** to create a key locator configuration, select the box next to the configuration and click **Delete** to delete an existing configuration, or click the name of an existing key locator configuration to edit its settings. If you are creating a new configuration, enter a unique name in the **Key locator name** field. For example, you might specify `gen_keyloc`.
2. Specify a class name for the key locator class implementation in the **Key locator class name** field. The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side. Specify a class name according to the requirements of the application. For example, if the application requires that the key is read from a keystore file, specify the com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator implementation. WebSphere Application Server supports the following default key locator class implementations for Versions 6.0.*x* and later applications that are available to use with the request generator or response generator:

   com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator
      This implementation locates and obtains the key from the specified keystore file.

   com.ibm.wsspi.wssecurity.keyinfo.SignerCertKeyLocator
      This implementation uses the public key from the signer certificate and is used by the response generator.

3. Specify the keystore password, the keystore location, and the keystore type. Keystore files contain public and private keys, root certificate authority (CA) certificates, the intermediate CA certificate, and so on. Keys retrieved from the keystore are used to sign and validate or encrypt and decrypt messages or message parts. If you specified the com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator implementation for the key locator class implementation, you must specify a keystore password, location, and type.
   a. Specify a password in the keystore **Password** field. This password is used to access the keystore file.
   b. Specify the location of the keystore file in the keystore **Path** field.
   c. Select a keystore type from the **Type** field. The Java Cryptography Extension (JCE) that is used by IBM supports the following keystore types:

      JKS Use this option if you are not using Java Cryptography Extensions (JCE) and if your keystore file uses the Java Keystore (JKS) format.

      JCEKS
      Use this option if you are using Java Cryptography Extensions.

      JCERACFKS
      Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

      PKCS11KS (PKCS11)
      Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

      PKCS12KS (PKCS12)
      Use this option if your keystore uses the PKCS#12 file format.

      WebSphere Application Server provides some sample keystore files in the `${USER_INSTALL_ROOT}/etc/ws-security/samples` directory. For example, you might use the `enc-receiver.jceks` keystore file for encryption keys. The password for this file is `Storepass` and the type is `JCEKS`.

      Restriction: Do not use the sample keystore files in a production environment. These samples are provided for testing purposes only.
4. Click **OK** and then click **Save** to save the configuration.
5. Under Additional properties, click **Keys**.
6. Click **New** to create a key configuration, select the box next to the configuration and click **Delete** to delete an existing configuration, or click the name of an existing key configuration to edit its settings. This entry specifies the name of the key object within the keystore file. If you are creating a new configuration, enter a unique name in the **Key name** field. For digital signatures, the key name is used by the request generator or the response generator signing information to determine which key is used to digitally sign the message.

   You must use a fully qualified distinguished name for the key name. For example, you might use `CN=Bob,O=IBM,C=US`.

   Important: Do not use the sample key files in a production environment. These samples are provided for testing purposes only.
7. Specify an alias in the **Key alias** field. The key alias is used by the key locator to search for key objects in the keystore.
8. Specify a password in the **Key password** field. The password is used to access the key object within the keystore file.
9. Click **OK** and **Save** to save the configuration.

Results

You have configured the key locator for the generator binding at the application level.

What to do next

You must specify a similar key information configuration for the consumer.

Key locator collection:

Use this page to view a list of key locator configurations that retrieve keys from the keystore for digital signature and encryption. A key locator must implement the `com.ibm.wsspi.wssecurity.config.KeyLocator` interface.

To view the administrative console panel for the key locator collection on the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Key locators**.

To view this administrative console page for the key locator collection on the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Key locators**.

To use this administrative console page for the key locator collection on the application level, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access key locators for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Key locators**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Key locators**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Key locators**.
   - For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Key locators**.
4. Under Additional properties, you can access key locators for the following bindings:
   - For the Request sender, click **Web services: Client security bindings**. Under Request sender binding, click **Edit** > **Key locators**.
   - For the Request receiver, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit** > **Key locators**.
   - For the Response sender, click **Web services: Server security bindings**. Under Response sender binding, click **Edit** > **Key locators**.
   - For the Response receiver, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit** > **Key locators**.

Tip: The bindings for a Version 6.x. or later application has a link that says **Edit custom**.

Using this **Key locator collection** panel, complete the following steps:
1. Specify a key locator name and a key locator class name on the panel.

2. Save your changes by clicking **Save** in the messages section at the top of the administrative console. The administrative console home panel is displayed.
3. After saving your changes, update the Web Services Security run time with the default binding information by clicking **Update runtime**. When you click **Update runtime**, the configuration changes made to the other Web services also are updated in the Web Services Security run time.
4. After you define key locators, click the key locator name to specify additional properties and keys under **Additional Properties**.

Key locator name:

Specifies the unique name of the key locator.

Key locator class name:

Specifies the class name of the key locator, which retrieves the key that is used for digital signing and encryption.

Key locator configuration settings:

Use this page to specify the settings for a key locator configuration. The key locators retrieve keys from the keystore file for digital signature and encryption. This product enables you to plug in a custom key locator configuration.

To view the administrative console panel for the key locator collection on the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Key locators**.
3. Click **New** to create a new configuration or click the name of a configuration to modify its settings.

To view this administrative console page for the key locator collection on the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Key locators**.
4. Click **New** to create a new configuration or click the name of a configuration to modify its settings.

To use this administrative console page for the key locator collection on the application level, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, you can access key locators for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Key locators**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Key locators**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Key locators**.

- For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Key locators**.

4. Click **New** to create a new configuration or click the name of a configuration to modify its settings.

Key locator name:

Specifies the name of the key locator.

Information	Value
Data type	String

Key locator class name:

Specifies the name for the key locator class implementation.

Key locators that are associated with Versions 6 and later applications must implement the `com.ibm.wsspi.wssecurity.keyinfo.KeyLocator` interface. This product provides the following default key locator class implementations for Versions 6 and later applications:

com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator
This implementation locates and obtains the key from the specified keystore file.

com.ibm.wsspi.wssecurity.keyinfo.SignerCertKeyLocator
This implementation uses the public key from the certificate of the signer. This class implementation is used by the response generator.

This property is for the JAX-RPC programming model only. To implement signer certificate encryption for the JAX-WS programming model, set a custom property on the callback handler for the encryption token generator. For more information, read the topic Callback handler settings.

com.ibm.wsspi.wssecurity.keyinfo.X509TokenKeyLocator
This implementation uses the X.509 security token from the sender message for digital signature validation and encryption. This class implementation is used by the request consumer and the response consumer.

Information	Value
Data type	String

Keystore: Specifies information about the key store that is used by this key locator configuration.

None Use this option if a key store is not required to be specified for this key locator configuration.

Predefined keystore
Use this option if you want to specify a predefined keystore for this key locator configuration.

User-defined keystore
Use this option if you want to specify a user-defined key store for this key locator configuration.

Keystore configuration name:

Specifies the name of the key store configuration that is defined in the keystore settings in secure communications.

The keystore configuration name is located under the **Predefined keystore** field, which is located under the **Keystore** section of the page.

Information	Value
Data type	String

Keystore password:

Specifies the password that is used to access the keystore file.

The keystore password is located under the **User-defined keystore** field, which is located under the **Keystore** section of the page.

Information	Value
Data type	String

Keystore path:

Specifies the location of the keystore file.

The path is located under the **User-defined keystore** field, which is located under the **Keystore** section of the page.

Information	Value
Data type	String

Keystore type:

Specifies the type of keystore file.

The type is located under the **User-defined keystore** field, which is located under the **Keystore** section of the page.

JKS
Use this option if you are not using Java Cryptography Extensions (JCE) and if your keystore file uses the Java Keystore (JKS) format.

JCEKS
Use this option if you are using Java Cryptography Extensions.

JCERACFKS
Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

PKCS11KS (PKCS11)
Use this format if your keystore file uses the PKCS#11 file format. Keystores files that use this format might contain Rivest Shamir Adleman (RSA) keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

PKCS12KS (PKCS12)
Use this option if your keystore file uses the PKCS#12 file format.

Information	Value
Default	JKS
Range	JKS, JCEKS, PKCS11KS (PKCS11), PKCS12KS (PKCS12)

Web Services Security property collection:

Use this page to a view a list of additional properties for the configuration.

You can view a Web Services Security property collection panel at the cell level. Complete the following steps to view one of these administrative console pages:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.

2. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Properties**.
3. Click **New** to create a new property.
4. Click **Delete** to a delete a property that you specified previously.

Property name:

Specifies the name of the property.

Property value:

Specifies the value for the property.

Web Services Security property configuration settings:

Use this page to configure additional security properties.

You can view a Web Services Security property configuration settings panel at the cell level. Complete the following steps to view one of these administrative console pages:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Generator Bindings or JAX-RPC Default Consumer Bindings, click **Properties** > **New**.

Property Name:

Specifies the name of the property.

Information	**Value**
Data type:	String

Property Value:

Specifies the value for the property.

Information	**Value**
Data type:	String

The following table lists the properties that you can configure by using the Web Services Security property panels.

Table 189. Property configuration settings. The properties are used to secure web services.

Configuration panel name	Property name	Property value	Description
JAAS configuration	com.ibm.wsspi.wssecurity.token.X509.issuerName	Specify the SubjectDN or the IssuerDN of the issuer for the X.509 certificate.	This property is used to specify the issuer of the certificate in the token consumer component.
JAAS configuration	com.ibm.wsspi.wssecurity.token.X509.issuerSerial	Specify the serial number of the X.509 certificate.	This property is used to specify the serial number of the certificate in the token consumer component.
Key information	com.ibm.wsspi.wssecurity.keyinfo.EncodingNS	Specify the namespace Uniform Resource Identifier (URI) for the qualified name (QName).	This property is used to specify the namespace URI part of the QName that represents the encoding method.

Table 189. Property configuration settings (continued). The properties are used to secure web services.

Configuration panel name	Property name	Property value	Description
Properties	com.ibm.ws.wssecurity.handler. hardwareCacheEntryRefreshHours	Specify a numeric value from 1 to 24 that represents the number of hours that a temporary key is valid.	This property is used to specify the amount of time before a key is retranslated. Temporary keys outside the keystore typically expire in a short period of time, measured in days or hours. If the server is configured to use a hardware acceleration card, but not the hardware keystore, you can configure it to translate the temporary keys periodically before they expire. If this property is not set, a key will be retranslated after 8 hours. Setting this value to 0 disables retranslation.
Request generator and Response generator	com.ibm.wsspi.wssecurity. timestamp.SOAPHeaderElement	Specify `1` or `true`.	This property is used with the **Add nonce** option to set the mustUnderstand flag in the deployment descriptor.
Request generator and Response generator	com.ibm.wsspi.wssecurity. timestamp.dialect	• A WebSphere Application Server special keywords • An XPath • A WS-Policy function The default value is `dialect-was`. See the com.ibm.wsspi.wssecurity Interface Constants for more information about the values that can be specified.	This property is used in conjunction with the com.ibm.wsspi.wssecurity.timestamp.keyword, which is used to place the timestamp header in a specific position in a message.
Signing information	com.ibm.wsspi.wssecurity. dsig.dumpPath	Specify the path used to locate the output file.	This property is used to specify an output file for dumping the target UTF-8 binary data before signing and verifying messages.
Token generator	com.ibm.wsspi.wssecurity. token.username.timestampExpires	Specify `1` or `true`.	This property is used to specify an expiration date for the user name token.
Transform algorithms	com.ibm.wsspi.wssecurity. dsig.XPathExpression	`not(ancestor-or-self::* [namespace-uri()='http://www. w3.org/2000/09/xmldsig#' and local-name()='Signature'])`	This property is used with this algorithm: `http://www.w3.org/TR/1999 /REC-xpath-19991116`

Configuring the key locator using JAX-RPC for the consumer binding on the application level:

The key locator information for the consumer at the application level specifies which key locator implementation is used. The key locator implementation locates the key to be used to validate the digital signature or the encryption information by the application.

About this task

Complete the following steps to configure the key locator for the consumer binding on the application level:

Procedure

1. Locate the key locator configuration panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security Properties, you can access the key information for the request consumer and response consumer bindings.
      - For the request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.

- For the response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.

d. Under Additional properties, click **Key locators**.

e. Click **New** to create a key locator configuration, click **Delete** and select the box next to the configuration to delete an existing configuration, or click the name of an existing key locator configuration to edit its settings. If you are creating a new configuration, enter a unique name in the **Key locator name** field. For example, you might specify `klocator`.

2. Specify a name for the key locator class implementation. The Java Authentication and Authorization Service (JAAS) Login Module implementation is used to validate (authenticate) the security token on the consumer side. Specify a class name according to the requirements of the application. For example, if the application requires that the key is read from a keystore file, specify the com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator implementation. WebSphere Application Server provides the following default key locator class implementations for Version 6.0.*x* applications that are available to use with the request consumer or response consumer:

com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator
This implementation locates and obtains the key from the specified keystore file.

com.ibm.wsspi.wssecurity.keyinfo.X509TokenKeyLocator
This implementation uses the X.509 security token from the sender message for digital signature validation and encryption. This class implementation is used by the request consumer and the response consumer.

3. Specify the keystore password, the keystore location, and the keystore type. Keystore files contain public and private keys, root certificate authority (CA) certificates, the intermediate CA certificate, and so on. Keys that are retrieved from the keystore files are used to sign and validate or encrypt and decrypt messages or message parts. If you specified the com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator implementation for the key locator class implementation, you must specify a keystore password, location, and type.

a. Specify a password in the keystore **Password** field. This password is used to access the keystore file.

b. Specify the location of the keystore file in the keystore **Path** field.

c. Select a keystore type from the keystore **Type** field. The Java Cryptography Extension (JCE) that is used by IBM supports the following keystore types:

JKS Use this option if you are not using Java Cryptography Extensions (JCE) and if your keystore file uses the Java Keystore (JKS) format.

JCEKS
Use this option if you are using Java Cryptography Extensions.

JCERACFKS
Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

PKCS11KS (PKCS11)
Use this format if your keystore file uses the PKCS#11 file format. Keystore files that use this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

PKCS12KS (PKCS12)
Use this option if your keystore uses the PKCS#12 file format.

WebSphere Application Server provides some sample keystore files in the `${USER_INSTALL_ROOT}/etc/ws-security/samples` directory. For example, you might use the `enc-receiver.jceks` keystore file for encryption keys. The password for this file is `Storepass` and the type is `JCEKS`.

Attention: Do not use these keystore files in a production environment. These samples are provided for testing purposes only.

4. Click **OK** and **Save** to save the configuration.

5. Under Additional properties, click **Keys**.

6. Click **New** to create a key configuration, click **Delete** and select the box next to the configuration to delete an existing configuration, or click the name of an existing key configuration to edit its settings. This entry specifies the name of the key object within the keystore file. If you are creating a new configuration, enter a unique name in the **Key name** field.

   It is recommended that you use a fully qualified distinguished name for the key name. For example, you might use `CN=Bob,O=IBM,C=US`.
7. Specify an alias in the **Key alias** field. The key alias is used by the key locator to search for key objects in the keystore file.
8. Specify a password in the **Key password** field. The password is used to access the key object within the keystore file.
9. Click **OK** and then click **Save** to save the configuration.

Results

You have configured the key locator for the consumer binding at the application level.

What to do next

You must specify a similar key information configuration for the generator.

Configuring the key locator using JAX-RPC on the server or cell level:

The key locator information for the default generator bindings specifies which key locator implementation is used to locate the key for signature and encryption information if these bindings are not defined at the application level.

About this task

The key locator information for the default consumer bindings specifies which key locator implementation is used to locate the key that is used for signature validation or decryption if these bindings are not defined at the application level. WebSphere Application Server provides default values for the bindings. However, you must modify the defaults for a production environment.

You can configure the key locator on the server level and the cell level. In the following steps, use the first step to access the server-level default bindings and use the second step to access the cell-level bindings.

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

      Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Additional properties, click **Key locator**. You can configure the key locator configurations for both the default generator and the default consumer in this location.
4. Click one of the following to work with the key locator configurations:

   New To create a key locator configuration. Enter a unique name for the key locator configuration in the **Key locator name** field. For example, you might specify `sig_klocator`.

   Delete To delete an existing configuration

   an existing key locator configuration
   To edit the settings of an existing configuration.

5. Specify a name for the key locator class implementation in the **Key locator class name** field. The key locators that are associated with Version 6.0.*x* applications must implement the `com.ibm.wsspi.wssecurity.keyinfo.KeyLocator` interface.

   Note: This interface is valid only for JAX-RPC applications. For JAX-WS applications, the Java Authentication and Authorization Service (JAAS) Login Module implementation is used to create the security token on the generator side and to validate (authenticate) the security token on the consumer side.

   WebSphere Application Server provides the following default key locator class implementations for Version 6.0.*x* applications:

   com.ibm.wsspi.wssecurity.keyinfo.KeyStoreLeyLocator
   This implementation locates and obtains the key from a specified keystore file.

   com.ibm.wsspi.wssecurity.keyinfo.SignerCertKeyLocator
   This implementation uses the public key from the certificate of the signer. This class implementation is used by the response generator.

   com.ibm.wsspi.wssecurity.keyinfo.X509TokenKeyLocator
   This implementation uses the X.509 security token from the sender message for digital signature validation and encryption. This class implementation is used by the request consumer and the response consumer.

   For example, you might specify the `com.ibm.wsspi.wssecurity.keyinfo.KeyStoreLeyLocator` implementation if you need the configuration to be the key locator for signing information.

6. Specify the keystore password, the keystore location, and the keystore type. Keystore files contain public and private keys, root certificate authority (CA) certificates, the intermediate CA certificate, and so on. Keys that are retrieved from the keystore file are used to sign and validate or encrypt and decrypt messages or message parts. If you specified the `com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator` implementation for the key locator class implementation, you must specify a key store password, location, and type.
   a. Specify a password in the **Key store password** field. This password is used to access the keystore file.
   b. Specify the location of the keystore file in the **Key store path** field.
   c. Select a keystore type from the **Key store type** field. The Java Cryptography Extension (JCE) that is used supports the following key store types:

      JKS Use this option if you are not using Java Cryptography Extensions (JCE) and if your keystore file uses the Java Keystore (JKS) format.

      JCEKS
      Use this option if you are using Java Cryptography Extensions.

      JCERACFKS
      Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

      PKCS11
      Use this format if your keystore file uses the PKCS#11 file format. Keystore files that use this format might contain Rivest Shamir Adleman (RSA) keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

      PKCS12
      Use this option if your keystore file uses the PKCS#12 file format.

      WebSphere Application Server provides some sample keystore files in the `${USER_INSTALL_ROOT}/etc/ws-security/samples` directory. For example, you might use the `enc-receiver.jceks` keystore file for encryption keys. The password for this file is `storepass` and the type is `JCEKS`.

      Restriction: Do not use these keystore files in a production environment. These samples are provided for testing purposes only.

7. Click **OK** and **Save** to save the configuration.
8. Under Additional properties, click **Keys**.
9. Click one of the following to work with the key configurations:

   New To create a key configuration. Enter a unique name in the **Key name** field. You must use a fully qualified distinguished name for the key name. For example, you might use `CN=Bob,O=IBM,C=US`.

   Delete To delete an existing configuration.

   an existing key configuration
   To edit the settings of the existing configuration.

   This entry specifies the name of the key object within the keystore file.
10. Specify an alias in the **Key alias** field. The key alias is used by the key locator to search for key objects in the keystore file.
11. Specify a password in the **Key password** field. The password is used to access the key object within the keystore file.
12. Click **OK** and then click **Save** to save the configuration.

Results

You have configured the key locator for the server or cell level.

What to do next

Configure the key information for the default generator and the default consumer bindings that reference this key locator.

Configuring trust anchors for the generator binding on the application level:

A *trust anchor* specifies key stores that contain trusted root certificates, which validate the signer certificate. These key stores are used by the request generator and the response generator (when web services are acting as client) to generate the signer certificate for the digital signature. You can configure trust anchors for the generator binding at the application level by using the administrative console.

Before you begin

You can configure a trust anchor using an assembly tool or the administrative console. This task describes how to configure the application-level trust anchor using the administrative console. For more information on assembly tools, see the related information.

About this task

The keystores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these keystores. The binding configuration that is specified for the request generator must match the binding configuration for the response generator.

The trust anchor configuration for the request generator on the client must match the configuration for the request consumer on the server. Also, the trust anchor configuration for the response generator on the server must match the configuration for the response consumer on the client.

Trust anchors defined at the application level have a higher precedence over trust anchors defined at the server or cell level. How to configure trust anchors at the server or cell level is not described in this task.

For more information on creating and configuring trust anchors on the server or cell level, see "Configuring trust anchors on the server or cell level" on page 960.

Complete the following steps to configure trust anchors for the generator binding on the application level:

Procedure

1. Locate the trust anchor panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security Properties you can access the trust anchor configuration for the following bindings:
      - For the request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
      - For the response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   d. Under Additional properties, click **Trust anchors**.
   e. Click **New** to create a trust anchor configuration, click **Delete** to delete an existing configuration, or click the name of an existing trust anchor configuration to edit its settings. If you are creating a new configuration, enter a unique name in the **Trust anchor name** field.
2. Specify the keystore password, the keystore location, and the keystore type. Key store files contain public and private keys, root certificate authority (CA) certificates, the intermediate CA certificate, and so on. Keys retrieved from the keystore are used to sign and validate or encrypt and decrypt messages or message parts. If you specified the `com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator` implementation for the key locator class implementation, you must specify a key store password, location, and type.
   a. Specify a password in the **Key store password** field. This password is used to access the keystore file.
   b. Specify the location of the key store file in the **Key store path** field.
   c. Select a keystore type from the **Key store type** field. The Java Cryptography Extension (JCE) used by IBM supports the following key store types:

      JKS
      Use this option if you are not using Java Cryptography Extensions (JCE) and if your keystore file uses the Java Keystore (JKS) format.

      JCEKS
      Use this option if you are using Java Cryptography Extensions.

      JCERACFKS
      Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

      PKCS11KS (PKCS11)
      Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

      PKCS12KS (PKCS12)
      Use this option if your keystore uses the PKCS#12 file format.

      WebSphere Application Server provides some sample keystore files in the following directory, using the *USER_INSTALL_ROOT* variable:

      For example, you might use the `enc-receiver.jceks` keystore file for encryption keys. The password for this file is `Storepass` and the type is `JCEKS`.

      Restriction: Do not use these keystore files in a production environment. These samples are provided for testing purposes only.

Results

This task configures trust anchors for the generator binding at the application level.

What to do next

You must specify a similar trust anchor configuration for the consumer.

Trust anchor collection:

Use this page to view a list of keystore objects that contain trusted root certificates. These objects are used for certificate path validation of incoming X.509-formatted security tokens. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trust of a certificate chain.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

To create the keystore file, use the key tool that is located in the `install_dir\java\jre\bin\keytool` directory.

To view this administrative console page for trust anchors on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Trust anchors**.

To view this administrative console page for trust anchors on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Trust anchors**.

To view this administrative console page for trust anchors on the application level,

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access trust anchors information for the following bindings:
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   - For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
4. Under Additional properties, you can access the trust anchors information for the following bindings:
   - For the Response receiver binding, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit**.
   - For the Request receiver binding, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit**.
5. Under Additional properties, click **Trust anchors**.

If you click **Update runtime**, the Web Services Security run time is updated with the default binding information, which is contained in the `ws-security.xml` file that was previously saved. If you make changes on this panel, you must complete the following steps:

1. Save your changes by clicking **Save** at the top of the administrative console. When you click **Save**, you are returned to the administrative console home panel.
2. Return to the Trust anchors collection panel and click **Update runtime**. When you click **Update runtime**, the configuration changes made to the other web services also are updated in the Web Services Security run time.

Trust anchor name:

Specifies the unique name that is used to identify the trust anchor.

Key store path:

Specifies the location of the keystore file that contains the trust anchors.

Key store type:

Specifies the type of keystore file.

The value for this field is **JKS**, **JCEKS**, **JCERACFKS** (z/OS only), **JCE4758RACFKS** (z/OS only), **PKCS11KS (PKCS11)**, or **PKCS12KS (PKCS12)**.

Trust anchor configuration settings:

Use this information to configure a trust anchor. Trust anchors point to keystores that contain trusted root or self-signed certificates. This information enables you to specify a name for the trust anchor and the information that is needed to access a keystore. The application binding uses this name to reference a predefined trust anchor definition in the binding file (or the default).

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

To view this administrative console page for trust anchors on the cell level, complete the following steps:
1. Click **Security > JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Trust anchors**.
3. Click **New** to create a trust anchor or click the name of an existing configuration to modify its settings.

To view this administrative console page for trust anchors on the server level, complete the following steps:
1. Click **Servers > Server Types > WebSphere application servers >** ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Trust anchors**.
4. Click **New** to create a trust anchor or click the name of an existing configuration to modify its settings.

To view this administrative console page for trust anchors on the application level,
1. Click **Applications > Application Types > WebSphere enterprise applications >** ***application_name***.
2. Under Modules, click **Manage modules >** ***URI_name***.
3. Under Web Services Security Properties, you can access trust anchors information for the following bindings:
   - For the Response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.

- For the Request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
4. Under Additional properties, you can access the trust anchors information for the following bindings:
   - For the Response receiver binding, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit**.
   - For the Request receiver binding, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit**.
5. Under Additional properties, click **Trust anchors**.
6. Click **New** to create a trust anchor or click the name of an existing configuration to modify its settings.

Trust anchor name:

Specifies the unique name that is used by the application binding to reference a predefined trust anchor definition in the default binding.

Key store configuration name:

Specifies the name of the key store configuration defined in the keystore settings in secure communications.

Key store password:

Specifies the password that is needed to access the key store file.

Key store path:

Specifies the location of the keystore file.

Use ${*USER_INSTALL_ROOT*} as this path expands to the WebSphere Application Server path on your machine.

Key store type:

Specifies the type of keystore file.

Choose from the following options:

JKS
Use this option if you are not using Java Cryptography Extensions (JCE).

JCEKS
Use this option if you are using Java Cryptography Extensions.

JCERACFKS
Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

PKCS11KS (PKCS11)
Use this format if your keystore uses the PKCS#11 file format. Keystores that use this format might contain Rivest Shamir Adleman (RSA) keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

PKCS12KS (PKCS12)
Use this option if your keystore uses the PKCS#12 file format.

Information	Value
Default	JKS
Range	JKS, JCEKS, PKCS11KS (PKCS11), PKCS12KS (PKCS12)

Configuring trust anchors for the consumer binding on the application level:

You can configure trust anchors for the consumer binding at the application level.

About this task

This article does not describe how to configure trust anchors at the server or cell level. Trust anchors that are defined at the application level have a higher precedence over trust anchors that are defined at the server or cell level. For more information on creating and configuring trust anchors on the server or cell level, see "Configuring trust anchors on the server or cell level" on page 960.

You can configure a trust anchor at the application level using an assembly tool or the administrative console. This article describes how to configure the application-level trust anchor using the administrative console.

A trust anchor specifies key stores that contain trusted root Certificate Authority (CA) certificates, which validate the signer certificate. These keystores are used by the request consumer (as defined in the `ibm-webservices-bnd.xmi` file) and the response consumer (as defined in the `ibm-webservicesclient-bnd.xmi` file when a web service is acting as a client) to validate the X.509 certificate in the SOAP message. The keystores are critical to the integrity of the digital signature validation. If the keystores are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these keystores. The binding configuration specified for the request consumer in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response consumer in the `ibm-webservicesclient-bnd.xmi` file. The trust anchor configuration for the request consumer on the server side must match the request generator configuration on the client side. Also, the trust anchor configuration for the response consumer on the client side must match the response generator configuration on the server side.

Complete the following steps to configure trust anchors for the consumer binding on the application level:

Procedure

1. Locate the trust anchor panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security Properties you can access the trust anchor configuration for the following bindings:
      - For the request consumer (receiver) binding, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom**.
      - For the response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   d. Under Additional properties, click **Trust anchors**.
   e. Click one of the following to work with trust anchor configuration:

      New To create a trust anchor configuration. Enter a unique name in the Trust anchor name field.

      Delete To delete the existing configuration selected in the box next to the configuration.

      an existing trust anchor configuration
      To edit the settings of an existing trust anchor configuration.
2. Specify the keystore password, the keystore location, and the keystore type. A trust anchor keystore file contains the trusted root Certificate Authority (CA) certificates that are used for validating the X.509 certificate that is used in digital signature or XML encryption.
   a. Specify a password in the Key store password field. This password is used to access the keystore file.

b. Specify the location of the keystore file in the Key store path field.

c. Select a keystore type from the Key store type field. The Java Cryptography Extension (JCE) that is used by IBM supports the following keystore types:

JKS Use this option if you are not using Java Cryptography Extensions (JCE) and if your keystore file uses the Java Keystore (JKS) format.

JCEKS
Use this option if you are using Java Cryptography Extensions.

JCERACFKS
Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

PKCS11KS (PKCS11)
Use this format if your keystore file uses the PKCS#11 file format. Keystore files that use this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

PKCS12KS (PKCS12)
Use this option if your keystore file uses the PKCS#12 file format.

WebSphere Application Server provides some sample keystore files in the following directory, using the *USER_INSTALL_ROOT* variable:

For example, you might use the `enc-receiver.jceks` keystore file for encryption keys. The password for this file is `storepass` and the type is `JCEKS`.

Restriction: Do not use these keystore files in a production environment. These samples are provided for testing purposes only.

Results

You have configured trust anchors for the consumer binding at the application level.

What to do next

You must specify a similar trust anchor information for the generator.

Configuring trust anchors on the server or cell level:

You can configure a list of keystore objects that contain trusted root certificates to be used for certificate path validation of incoming X.509-formatted security tokens.

Before you begin

Prior to completing the steps to configure trust anchors, you must create the keystore file using the key tool. WebSphere Application Server provides the key tool in the *install_dir*`/java/jre/bin/keytool` file.

About this task

This task provides the steps that are needed to configure a list of keystore objects that contain trusted root certificates. These objects are used for certificate path validation of incoming X.509-formatted security tokens. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath application programming interface (API) to determine whether to trust a certificate chain.

You can configure trust anchors on the server level and the cell level. In the following steps, use the first step to access the server-level default bindings and use the second step to access the cell-level bindings.

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

      Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Additional properties, click **Trust anchors**.
4. Click one of the following to work with trust anchor configuration:

   New To create a trust anchor configuration. Enter a unique name for the trust anchor in the Trust anchor name field.

   Delete To delete an existing configuration.

   an existing trust anchor configuration
   To edit the settings for an existing trust anchor.
5. Specify a password in the Key store password field that is used to access the keystore file.
6. Specify the absolute location of the keystore file in the **Key store path** field. It is recommended that you use the *USER_INSTALL_ROOT* variable as a portion of the keystore path. To change this predefined variable, click **Environment** > **WebSphere variables**. The *USER_INSTALL_ROOT* variable might display on the second page of variables.
7. Specify the type of keystore file in the key store type field. WebSphere Application Server supports the following keystore types:

   JKS Use this option if you are not using Java Cryptography Extensions (JCE) and your keystore file uses the Java Key Store (JKS) format.

   JCEKS
   Use this option if you are using Java Cryptography Extensions.

   JCERACFKS
   Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only).

   PKCS11KS (PKCS11)
   Use this option if your keystore file uses the PKCS#11 file format. Keystore files that use this format might contain Rivest Shamir Adleman (RSA) keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection.

   PKCS12KS (PKCS12)
   Use this option if your keystore file uses the PKCS#12 file format.
8. Click **OK** and **Save** to save your configuration.

Results

You have configured trust anchors at the server or cell level.

Configuring the collection certificate store for the generator binding on the application level:

You can configure a collection certificate for the generator bindings on the application level.

About this task

A *collection certificate store* is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs is used to check for a valid signature in a digitally signed SOAP message.

Complete the following steps to configure a collection certificate for the generator bindings on the application level:

Procedure

1. Locate the collection certificate store configuration panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security Properties, you can access the key information for the request generator and response generator bindings.
      - For the request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
      - For the response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   d. Under Additional properties, click **Collection certificate store**.
2. Specify the Certificate store name. Click **New** to create a collection certificate store configuration, select the box next to the configuration and click **Delete** to delete an existing configuration, .or click the name of an existing collection certificate store configuration to edit its settings. If you are creating a new configuration, enter a name in the Certificate store name field.

   The name of the collection certificate store must be unique to the level of the application server. For example, if you create the collection certificate store for the application level, the store name must be unique to the application level. The name that is specified in the Certificate store name field is used by other configurations to refer to a predefined collection certificate store. WebSphere Application Server searches for the collection certificate store based on proximity.

   For example, if an application binding refers to a collection certificate store named `cert1`, the Application Server searches for `cert1` at the application level before searching the server level and then the cell level.
3. Specify a certificate store provider in the Certificate store provider field. WebSphere Application Server supports the IBMCertPath certificate store provider. To use another certificate store provider, you must define the provider implementation in the provider list within the *install_dir*/java/jre/lib/security/java.security file. However, make sure that your provider supports the same requirements of the certificate path algorithm as WebSphere Application Server.
4. Click **OK** and **Save** to save the configuration.
5. Click the name of your certificate store configuration. After you specify the certificate store provider, you must specify either the location of a certificate revocation list or the X.509 certificates. However, you can specify both a certificate revocation list and the X.509 certificates for your certificate store configuration.
6. Under Additional properties, click **Certificate revocation lists**.
7. Click **New** to specify a certificate revocation list path, click **Delete** to delete an existing list reference, or click the name of an existing reference to edit the path. You must specify the fully qualified path to the location where WebSphere Application Server can find your list of certificates that are not valid. For portability reasons, it is recommended that you use the WebSphere Application Server variables to specify a relative path to the certificate revocation lists (CRL). This recommendation is especially important when you are working in a WebSphere Application Server, Network Deployment environment. For example, you might use the *USER_INSTALL_ROOT* variable to define a path such as $*USER_INSTALL_ROOT*/mycertstore/mycrl1. For a list of supported variables, click **Environment** > **WebSphere variables** in the administrative console. The following list provides recommendation for using certificate revocation lists:
   - If CRLs are added to the collection certificate store, add the CRLs for the root certificate authority and each intermediate certificate, if applicable. When the CRL is in the certificate collection store, the certificate revocation status for every certificate in the chain is checked against the CRL of the issuer.

- When the CRL file is updated, the new CRL does not take effect until you restart the web service application.
- Before a CRL expires, you must load a new CRL into the certificate collection store to replace the old CRL. An expired CRL in the collection certificate store results in a certificate path (CertPath) build failure.

8. Click **OK** and **Save** to save the configuration.
9. Return to the collection certificate store configuration panel. To access the panel, complete the following steps:
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Manage modules, click ***URI_name***.
   c. Under Web Services Security properties, you can access the key information for the request generator and response generator bindings.
      - For the request generator (sender) binding, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom**.
      - For the response generator (sender) binding, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom**.
   d. Under Additional properties, click **Collection certificate store** > ***certificate_store_name***.
10. Under Additional properties, click **X.509 certificates**.
11. Click **New** to create a X.509 certificate configuration, click **Delete** to delete an existing configuration, or click the name of an existing X.509 certificate configuration to edit its settings. If you are creating a new configuration, enter a name in the Certificate store name field.
12. Specify a path in the X.509 certificate path field. This entry is the absolute path to the location of the X.509 certificate. The collection certificate store is used to validate the certificate path of incoming X.509-formatted security tokens.

    You can use the *USER_INSTALL_ROOT* variable as part of path name. For example, you might type: `USER_INSTALL_ROOT/etc/ws-security/samples/intca2.cer`. Do not use this certificate path for production use. You must obtain your own X.509 certificate from a certificate authority before putting your WebSphere Application Server environment into production.

    Click **Environment** > **WebSphere variables** in the administrative console to configure the *USER_INSTALL_ROOT* variable.
13. Click **OK** and then **Save** to save your configuration.

Results

You have configured the collection certificate store for the generator binding.

What to do next

You must specify a similar collection certificate store configuration for the consumer.

Collection certificate store collection:

Use this page to view a list of certificate stores that contains untrusted, intermediary certificate files awaiting validation. Validation might consist of checking to see if the certificate is on a certificate revocation list (CRL), checking that the certificate is not expired, and checking that the certificate is issued by a trusted signer.

The following list provides recommendations for using CRLs:

- If CRLs are added to the collection certificate store collection, add the CRLs for the root certificate authority and each intermediate certificate, if applicable. When the CRL is in the certificate collection store, the certificate revocation status for every certificate in the chain is checked against the CRL of the issuer.
- When the CRL file is updated, the new CRL does not take effect until you restart the web service application.
- Before a CRL expires, you must load a new CRL into the certificate collection store to replace the old CRL. An expired CRL in the collection certificate store results in a certificate path (CertPath) build failure.

To view the administrative console panel for the collection certificate store on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Collection certificate store**.

To view the administrative console panel for the collection certificate store on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Collection certificate store**.

To view this administrative console page for the collection certificate store on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access collection certificate stores for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
4. Under Additional properties, you can access collection certificate stores for the following bindings:
   - For the Request receiver binding, click **Web services: Server security bindings**. Under Response receiver binding, click **Edit** > **Collection certificate store**.
   - For the Response receiver binding, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit** > **Collection certificate store**.

Complete the following steps:

1. Click **New** to specify a new certificate store name and certificate store provider.
2. Click **OK** and messages display at the top of the administrative console panel.
3. Within the messages at the top of the administrative console panel, click **Save**.
4. Return to the collection certificate store collection panel and click **Update runtime** to update the Web Services Security run time with the default binding information, which is found in the `ws-security.xml`

file. When you click **Update runtime**, the configuration changes made to the other web services are also updated in the Web Services Security run time.

Certificate store name:

Specifies the name of the certificate store.

Certificate store provider:

Specifies the provider of the certificate store.

Collection certificate store configuration settings:

Use this page to specify the name and the provider for a collection certificate store. A *collection certificate store* is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs is used to check the signature of a digitally signed SOAP message.

To view the administrative console panel for the collection certificate store on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Collection certificate store**.
3. Specify a new collection certificate store by clicking **New** or click the collection certificate store name to modify its settings.

To view the administrative console panel for the collection certificate store on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Collection certificate store**.
4. Specify a new collection certificate store by clicking **New** or by clicking the collection certificate store name to modify its settings.

To view this administrative console page for the collection certificate store on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access collection certificate stores for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
4. Under Additional properties, you can access collection certificate stores for the following bindings:

- For the Request receiver binding click **Edit** > **Collection certificate store**.
- For the Response receiver binding, click **Edit** > **Collection certificate store**.

5. Specify a new collection certificate store by clicking **New** or by clicking the collection certificate store name to modify its settings.

After configuring a collection certificate store, you can select the new configuration under Certificate store on the token generator and token consumer panels. To access these panels, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under JAX-RPC Default Generator Bindings, click **Token generators** or under JAX-RPC Default Consumer Bindings, click **Token consumers**.
3. Click **New** to create a new token generator or token consumer, or click the name of an existing configuration to make modifications.

After you configure your collection certificate store on this panel, you must click **Apply** before configuring either the certificate revocation list or an X.509 certificate. After you configure your certificate revocation list or X.509 certificate, complete the following steps:

1. Click **Save**, at the top of the administrative console panel, which returns you to the list of the configured collection certificate stores.
2. Click **Update runtime** to update the Web Services Security run time with the default binding information, which is found in the `ws-security.xml` file.

Certificate store name:

Specifies the name for the certificate store.

The name of the collection certificate store must be unique in the scope. For example, the name must be unique at the server level. The name specified in **Certificate store name** field is used by other configurations to refer to a pre-defined collection certificate store. For example, the application binding refers to a collection certificate store that is defined on the server level. The application server looks up the collection certificate store based on proximity. For example, if *cert1* is defined as the name of the certificate store on the cell and server levels and *cert1* is referenced in the application binding, the application server uses the server-level collection certificate store.

Certificate Store Provider:

Specifies the provider for the certificate store implementation.

This product supports the IBMCertPath certificate path provider. If you need to use another certificate path provider, define the provider implementation in the provider list within the `java.security` file in the Software Development Kit (SDK).

Information	Value
Data type	String
Default	IBMCertPath

X.509 certificates collection:

Use this page to view a list of untrusted, intermediate certificate files. This collection certificate store is used for certificate path validation of incoming X.509-formatted security tokens.

To view the administrative console page for the collection certificate store on the cell level in a network deployment (ND) environment, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.

2. Under Additional properties, click **Collection certificate store**.
3. Click the name of a configured collection certificate store or create a new collection certificate store first.
4. Under Additional properties, click **X.509 certificates**.

To view the administrative console page for the collection certificate store on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Collection certificate store**.
4. Click the name of a configured collection certificate store or create a new collection certificate store first.
5. Under Additional properties, click **X.509 certificates**.

To view this administrative console page for an X.509 certificate on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access collection certificate stores for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
4. Under Additional properties, you can access the collection certificate stores for the following bindings.
   - For the Response receiver binding, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit** > **Collection certificate store**.
   - For the Request receiver binding, click **Web services: Server security bindings**. Under Request receiver binding, click **Edit** > **Collection certificate store**.
5. Click the name of a configured collection certificate store or create a new collection certificate store first.
6. Under Additional properties, click **X.509 certificates**.

X.509 certificate path:

Specifies the location of the X.509 certificate.

X.509 certificate configuration settings:

Use this page to specify a list of untrusted, intermediate certificate files. This collection certificate store is used for certificate path validation of incoming X.509-formatted security tokens.

To view the administrative console page for the collection certificate store on the cell level in a network deployment (ND) environment, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under additional properties, click **Collection certificate store**.
3. Click the name of a configured collection certificate store or create a new collection certificate store first.
4. Under Additional properties, click **X.509 certificates**.
5. Specify a new X.509 certificate path by clicking **New** or by clicking the X.509 certificate path to modify its settings.

To view the administrative console page for the collection certificate store on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Collection certificate store**.
4. Click the name of a configured collection certificate store or create a new collection certificate store first.
5. Under Additional properties, click **X.509 certificates**.
6. Specify a new X.509 certificate path by clicking **New** or by clicking the X.509 certificate path to modify its settings.

To view this administrative console page for an X.509 certificate on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access collection certificate stores for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
4. Click the name of a configured collection certificate store or create a new collection certificate store first.
5. Under Additional properties, click **X.509 certificates**.
6. Specify a new X.509 certificate path by clicking **New** or click the X.509 certificate path to modify its settings.

X.509 Certificate Path:

Specifies the absolute path to the location of the X.509 certificate.

As shown in the following example, you can use the USER_INSTALL_ROOT variable as part of the path name: `{USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`. This X.509 certificate path is not for production use. Obtain your own X.509 from a certificate authority before putting your application server environment into production.

You can configure the USER_INSTALL_ROOT variable in the administrative console by clicking **Environment** > **WebSphere Variables**.

Certificate revocation list collection:

Use this page to determine the location of the certificate revocation list (CRL) known to the application server. The Application Server checks the CRL to determine the validity of the client certificate. A certificate that is found in a certificate revocation list might not be expired, but is no longer trusted by the certificate authority (CA) that issued the certificate. The CA might add the certificate to the certificate revocation list if it believes that the client authority is compromised.

View the administrative console panel for the collection certificate store on the cell level.

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under additional properties, click **Collection certificate store**.
3. Click the name of a configured collection certificate store or create a new collection certificate store first.
4. Under Additional properties, click **Certificate revocation list** > **New** to specify the path to a new list or click the name of the certificate revocation list to modify its path.

View the administrative console panel for the collection certificate store on the server level.

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Collection certificate store**.
4. Click the name of a configured collection certificate store or create a new collection certificate store first.
5. Under Additional properties, click **Certificate revocation list** > **New**to specify the path to a new list or click the name of the certificate revocation list to modify its path.

View the administrative console page for the collection certificate store on the application level.

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access collection certificate stores for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Collection certificate store**.
   - For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
4. Click the name of a configured collection certificate store or create a new collection certificate store first.

5. Under Additional properties, click **Certificate revocation list** > **New** to specify the path to a new list or click the name of the certificate revocation list to modify its path.
6. Under Additional properties, you can access collection certificate stores for the following bindings:
   - For the Response receiver binding, click **Web services: Client security bindings**. Under Response receiver binding, click **Edit**.
7. Under Additional properties, click **Collection certificate store** > ***certificate_store_name***.
8. Under Additional properties, click **X.509 certificates**.
9. Click **New** and specify the path to the certificate revocation list.

Certificate revocation list path:

Specifies the location where you can find the list of certificates that are not valid.

Certificate revocation list configuration settings:

Use this page to specify a list of certificate revocations that check the validity of a certificate. The application server checks the certificate revocation lists (CRL) to determine the validity of the client certificate. A certificate that is found in a certificate revocation list might not be expired, but is no longer trusted by the certificate authority (CA) that issued the certificate. The CA might add the certificate to the certificate revocation list if it believes that the client authority is compromised.

To view the administrative console panel for the collection certificate store on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under additional properties, click **Collection certificate store**.
3. Click the name of a configured collection certificate store or create a new collection certificate store first.
4. Under Additional properties, click **Certificate revocation lists** > **New** to specify the path to a new list or click the name of a certificate revocation list to modify its path.

To view the administrative console panel for the collection certificate store on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Collection certificate store**.
4. Click the name of a configured collection certificate store or create a new collection certificate store first.
5. Under Additional properties, click **Certificate revocation lists** > **New** to specify the path to a new list or click the name of a certificate revocation list to modify its path.

To view this administrative console page for the collection certificate store on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, you can access collection certificate stores for the following bindings:
   - For the Request generator, click **Web services: Client security bindings**. Under Request generator (sender) binding, click **Edit custom** > **Collection certificate store**.

- For the Request consumer, click **Web services: Server security bindings**. Under Request consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.
- For the Response generator, click **Web services: Server security bindings**. Under Response generator (sender) binding, click **Edit custom** > **Collection certificate store**.
- For the Response consumer, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom** > **Collection certificate store**.

4. Click the name of a configured collection certificate store or create a new collection certificate store first.
5. Under Additional properties, click **Certificate revocation lists** > **New** to specify the path to a new list or click the name of a certificate revocation list to modify its path.

Certificate revocation list path:

Specifies a fully qualified path to the location where you can find the list of certificates that are not valid.

For portability reasons, it is recommended that you use application server variables to specify a relative path to the certificate revocation list. This recommendation is especially important when you are working in a WebSphere Application Server, Network Deployment environment. For example, you might use the *USER_INSTALL_ROOT* variable to define a path such as *$USER_INSTALL_ROOT/mycertstore/mycrl* where *mycertstore* represents the name of your certificate store and *mycrl* represents the certificate revocation list. For a list of the supported variables, click **Environment** > **WebSphere variables** in the administrative console.

The following list provides recommendations for using CRLs:
- If CRLs are added to the collection certificate store collection, add the CRLs for the root certificate authority and each intermediate certificate, if applicable. When the CRL is in the certificate collection store, the certificate revocation status for every certificate in the chain is checked against the CRL of the issuer.
- When the CRL file is updated, the new CRL does not take effect until you restart the web service application.
- Before a CRL expires, you must load a new CRL into the certificate collection store to replace the old CRL. An expired CRL in the collection certificate store results in a certificate path (CertPath) build failure.

Configuring the collection certificate store for the consumer binding on the application level:

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs is used to check for a valid signature in a digitally signed SOAP message.

About this task

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs) that can be used to check for a valid signature in a digitally signed SOAP message. Complete the following steps to configure a collection certificate for the consumer bindings on the application level:

Procedure

1. Locate the collection certificate store configuration panel in the administrative console.
   a. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
   b. Under Modules, click **Manage modules** > ***URI_name***.

c. Under Web Services Security properties, you can access the collection certificate store information for the response consumer and request consumer bindings.
   - For the response consumer (receiver) binding, click **Web services: Client security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.
   - For the request consumer (receiver) binding, click **Web services: Server security bindings**. Under Response consumer (receiver) binding, click **Edit custom**.

d. Under Additional properties, click **Collection certificate store**.

2. Click **New** to create a collection certificate store configuration, click **Delete** to delete an existing configuration, or click the name of an existing collection certificate store configuration to edit its settings. If you are creating a new configuration, enter a name in the Certificate store name field.

   The name of the collection certificate store must be unique to the level of the application server. For example, if you create the collection certificate store for the application level, the store name must be unique to the application level. The name that is specified in the Certificate store name field is used by other configurations to refer to a predefined collection certificate store. WebSphere Application Server searches for the collection certificate store based on proximity.

   For example, if an application binding refers to a collection certificate store named `cert1`, the Application Server searches for `cert1` at the application level before searching the server level and then the cell level.

3. Specify a certificate store provider in the Certificate store provider field. WebSphere Application Server supports the IBMCertPath certificate store provider. To use another certificate store provider, you must define the provider implementation in the provider list within the `install_dir/properties/java.security/java.security` file. However, make sure that your provider supports the same requirements of the certificate path algorithm as WebSphere Application Server.
4. Click **OK** and **Save** to save the configuration.
5. Click the name of your certificate store configuration. After you specify the certificate store provider, you must specify either the location of a certificate revocation list or the X.509 certificates. However, you can specify both a certificate revocation list and the X.509 certificates for your certificate store configuration.
6. Under Additional properties, click **Certificate revocation lists**.
7. Click **New** to specify a certificate revocation list path, click **Delete** to delete an existing list reference, or click the name of an existing reference to edit the path. You must specify the fully qualified path to the location where WebSphere Application Server can find your list of certificates that are not valid. For portability reasons, it is recommended that you use the WebSphere Application Server variables to specify a relative path to the certificate revocation lists (CRL). This recommendation is especially important when you are working in a WebSphere Application Server, Network Deployment environment. For example, you might use the *USER_INSTALL_ROOT* variable to define a path such as $*USER_INSTALL_ROOT*/mycertstore/mycrl1. For a list of supported variables, click **Environment** > **WebSphere variables** in the administrative console. The following list provides recommendation for using certificate revocation lists:
   - If CRLs are added to the collection certificate store, add the CRLs for the root certificate authority and each intermediate certificate, if applicable. When the CRL is in the certificate collection store, the certificate revocation status for every certificate in the chain is checked against the CRL of the issuer.
   - When the CRL file is updated, the new CRL does not take effect until you restart the web service application.
   - Before a CRL expires, you must load a new CRL into the certificate collection store to replace the old CRL. An expired CRL in the collection certificate store results in a certificate path (CertPath) build failure.
8. Click **OK** and **Save** to save the configuration.
9. Return to the Collection certificate store configuration panel. See the first few steps of this article to locate the collection certificate store panel.
10. Under Additional properties, click **X.509 certificates**.

11. Click **New** to create a new configuration for X.509 certificates, click **Delete** to delete an existing configuration, or click the name of an existing X.509 certificate configuration to edit its settings. If you are creating a new configuration, enter a name in the Certificate store name field.
12. Specify a path in the X.509 certificate path field. This entry is the absolute path to the location of the X.509 certificates. The collection certificate store is used to validate the certificate path of incoming X.509-formatted security tokens.

    You can use the *USER_INSTALL_ROOT* variable as part of the path name. For example, you might type: `USER_INSTALL_ROOT/etc/ws-security/samples/intca2.cer`. Do not use this certificate path for production use. You must obtain your own X.509 certificate from a certificate authority before putting your WebSphere Application Server environment into production.

    Click **Environment** > **WebSphere variables** in the administrative console to configure the *USER_INSTALL_ROOT* variable.
13. Click **OK** and then **Save** to save your configuration.

Results

You have configured the collection certificate store for the consumer binding.

What to do next

You must configure a token consumer configuration that references this certificate store configuration.

Configuring the collection certificate on the server or cell level:

Collection certificate stores contain untrusted, intermediary certificate files awaiting validation. You can configure a collection certificate store on the server level.

About this task

Validation might consist of checking for a valid signature in a digitally signed SOAP message to see if the certificate is on a certificate revocation list (CRLs), checking that the certificate is not expired, and checking that the certificate is issued by a trusted signer.

In the following steps, use the first step to configure the collection certificate store for the server level and use the second step to configure the collection certificate store for the cell level:

Procedure

1. Access the default bindings for the server level.
    a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
    b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

    Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Additional properties, click **Collection certificate store**.
4. Click **New** to create a collection certificate store configuration, click **Delete** to delete an existing configuration, or click the name of an existing collection certificate store configuration to edit its settings. If you are creating a new configuration, enter a name in the Certificate store name field. For example, you might name your certificate store `sig_certstore`.

    The name of the collection certificate store must be unique to the level of the application server. For example, if you create the collection certificate store for the server level, the store name must be unique to the server level. The name that is specified in the Certificate store name field is used by

other configurations to refer to a predefined collection certificate store. WebSphere Application Server searches for the collection certificate store based on proximity.

For example, if an application binding refers to a collection certificate store named `cert1`, the Application Server searches for `cert1` at the application level before searching the server level and then the cell level.

5. Specify a certificate store provider in the Certificate store provider field. WebSphere Application Server supports the IBMCertPath certificate store provider. To use another certificate store provider, you must define the provider implementation in the provider list within the *install_dir*`/properties/java.security` file. However, make sure that your provider supports the same requirements of the certificate path algorithm as WebSphere Application Server.
6. Click **OK** and **Save** to save the configuration.
7. Click the name of your certificate store configuration. After you specify the certificate store provider, you must specify either the location of a certificate revocation list or the X.509 certificates. However, you can specify both a certificate revocation list and the X.509 certificates for your certificate store configuration.
8. Under Additional properties, click **Certificate revocation lists**. For the generator binding, a certificate revocation list (CRL) is used when it is included in a generated security token. For example, a security token might be wrapped in a PKCS#7 format with a CRL. For more information on certificate revocation lists, see "Certificate revocation list" on page 272.
9. Click **New** to specify a certificate revocation list path, click **Delete** to delete an existing list reference, or click the name of an existing reference to edit the path. You must specify the fully qualified path to the location where WebSphere Application Server can find your list of certificates that are not valid. WebSphere Application Server uses the certificate revocation list to check the validity of the sender certificate.

   For portability reasons, it is recommended that you use the WebSphere Application Server variables to specify a relative path to the certificate revocation lists. This recommendation is especially important when you are working in a WebSphere Application Server, Network Deployment environment.

   For example, you might use the *USER_INSTALL_ROOT* variable to define a path such as *$USER_INSTALL_ROOT/mycertstore/mycrl1* where *mycertstore* represents the name of your certificate store and *mycrl1* represents the certificate revocation list. For a list of supported variables, click **Environment** > **WebSphere variables** in the administrative console. The following list provides recommendations for using certificate revocation lists:

   - If CRLs are added to the collection certificate store, add the CRLs for the root certificate authority and each intermediate certificate, if applicable. When the CRL is in the certificate collection store, the certificate revocation status for every certificate in the chain is checked against the CRL of the issuer.
   - When the CRL file is updated, the new CRL does not take effect until you restart the web service application.
   - Before a CRL expires, you must load a new CRL into the certificate collection store to replace the old CRL. An expired CRL in the collection certificate store results in a certificate path (CertPath) build failure.
10. Click **OK** and then **Save** to save the configuration.
11. Return to the Collection certificate store configuration panel.
12. Under Additional properties, click **X.509 certificates**. The X.509 certificate configuration specifies intermediate certificate files that are used for certificate path validation of incoming X.509-formatted security tokens.
13. Click **New** to create an X.509 certificate configuration, click **Delete** to delete an existing configuration, or click the name of an existing X.509 certificate configuration to edit its settings. If you are creating a new configuration, enter a name in the Certificate store name field.

14. Specify a path in the X.509 certificate path field. This entry is the absolute path to the location of the X.509 certificate. The collection certificate store is used to validate the certificate path of the incoming X.509-formatted security tokens.

    You can use the *USER_INSTALL_ROOT* variable as part of path name. For example, you might type: `$USER_INSTALL_ROOT/etc/ws-security/samples/intca2.cer`. Do not use this certificate path for production use. You must obtain your own X.509 certificate from a certificate authority before putting your WebSphere Application Server environment into production.

    Click **Environment** > **WebSphere variables** in the administrative console to configure the *USER_INSTALL_ROOT* variable.
15. Click **OK** and then **Save** to save your configuration.
16. Return to the Collection certificate store collection panel and click **Update run time** to update the Web Services Security run time with the default binding information, which is located in the `ws-security.xml` file. When you click **Update run time**, the configuration changes made to other web services are also updated in the run time for Web Services Security. Policy sets can only be used with JAX-WS applications. Policy sets cannot be used for JAX-RPC applications.

Results

You have configured the collection certificate store for the server or cell level.

Configuring trusted ID evaluators on the server or cell level:

You can configure trusted identity (ID) evaluators. The trusted ID evaluator determines whether or not to trust the identity-asserting authority.

About this task

This task provides the steps that are needed to configure trusted identity (ID) evaluators. The trusted ID evaluator determines whether to trust the identity-asserting authority. After the ID is trusted, the WebSphere Application Server issues the proper credentials based on the identity, which are used in a downstream call to another server for invoking resources. The trusted ID evaluator implements the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` interface.

You can configure the trusted ID evaluators on the server level and the cell level. In the following steps, use the first step to access the server-level default bindings and use the second step to access the cell-level bindings:

Procedure

1. Access the default bindings for the server level.
   a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
   b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

      Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
2. Click **Security** > **Web services** to access the default bindings on the cell level.
3. Under Additional properties, click **Trusted ID evaluators**.
4. Click **New** to create a trusted ID evaluator configuration, click **Delete** to delete an existing configuration, or click the name of an existing configuration to edit the settings. If you are creating a new configuration, enter a unique name for the trusted ID evaluator configuration in the Trusted ID evaluator name field. This field specifies the name that is used by the application binding to refer to a trusted identity (ID) evaluator that is defined in the default binding.
5. Specify a class name in the Trusted ID evaluator class name field. The default class name is `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl`. The specified trusted ID evaluator class name

must implement the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` class. When you use the default `TrustedIDEvaluator` class, you must specify the name and value properties for the default trusted ID evaluator to create the trusted ID list for evaluation.

6. Under Additional properties, click **Properties > New**.
7. Specify the trusted ID evaluator name as a property name. You must specify the trusted ID evaluator name in the form, `trustedId_n`, where *_n* is an integer from zero (0) to n.
8. Specify the trusted ID as a property value.

```
property name="trustedId_0", value="CN=Bob,O=ACME,C=US"
property name="trustedId_1, value="user1"
```

If a distinguished name (DN) is used, the space is removed for comparison.

9. Click **OK** and then **Save**.

Results

You have configured the trusted ID evaluators at the server or cell level.

Trusted ID evaluator collection:

Use this page to view a list of trusted identity (ID) evaluators. The trusted ID evaluator determines whether to trust the identity-asserting authority. After the ID is trusted, the application server issues the proper credentials based on the identity, which are used in a downstream call for invoking resources. The trusted ID evaluator implements the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` interface.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

To view this administrative console page for trusted ID evaluators on the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Trusted ID evaluators**.
3. Click **New** to create a trusted ID evaluator or click **Delete** to delete a trusted ID evaluator.

To view this administrative console page for trusted ID evaluators on the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Trusted ID evaluators**.
4. Click **New** to create a trusted ID evaluator or click **Delete** to a delete a trusted ID evaluator.

To view this administrative console page for trusted ID evaluators on the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage Modules** > ***URI_name***.
3. Under Web Services Security Properties, click **Web services: Server security bindings**.
4. Under Request receiver binding, click **Edit**.
5. Click **Trusted ID evaluators**.
6. Click **New** to create a trusted ID evaluator or click **Delete** to delete a trusted ID evaluator.

Important: Trusted ID evaluators are only required for the request consumer (Version 6.*x* applications), if identity assertion is configured.

Using this trusted ID evaluator collection panel, complete the following steps:
1. Specify a trusted ID evaluator name and a trusted ID evaluator class name.
2. Save your changes by clicking **Save** in the messages section at the top of the administrative console.
3. Click **Update run time** to update the Web Services Security run time with the default binding information, which is found in the `ws-security.xml` file. The configuration changes made to the other web services also are updated in the Web Services Security run time.

Trusted ID evaluator name:

Specifies the unique name of the trusted ID evaluator.

Trusted ID evaluator class name:

Specifies the class name of the trusted ID evaluator.

Trusted ID evaluator configuration settings:

Use this information to configure trust identity (ID) evaluators.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

To view this administrative console page for trusted ID evaluators on the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.
2. Under Additional properties, click **Trusted ID evaluators**.
3. Click **New** to create a trusted ID evaluator or click the name of an existing configuration to modify its settings.

To view this administrative console page for trusted ID evaluators on the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Trusted ID evaluators**.
4. Click **New** to create a trusted ID evaluator or click the name of an existing configuration to modify the settings.

To view this administrative console page for trusted ID evaluators on the application level, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, click **Web services: Server security bindings**.
4. Under Request receiver binding, click **Edit**.
5. Click **Trusted ID evaluators**.
6. Click **New** to create a trusted ID evaluator or click **Delete** to delete a trusted ID evaluator.

Important: Trusted ID evaluators are only required for the request consumer (Version 6.*x* applications), if identity assertion is configured.

You can specify one of the following options:

None Choose this option if you are not specifying a trusted ID evaluator.

Existing evaluator definition
Choose this option to specify a currently defined trusted ID evaluator.

Binding evaluator definition
Choose this option to specify a new trusted ID evaluator. A description of the required fields follows.

Trusted ID evaluator name:

Specifies the name that is used by the application binding to refer to a trusted identity (ID) evaluator that is defined in the default binding.

Trusted ID evaluator class name:

Specifies the class name of the trusted ID evaluator.

The specified trusted ID evaluator class name must implement the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` interface. The default TrustedIDEvaluator class is `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl`. When you use this default TrustedIDEvaluator class, you must specify the name and the value properties for the default trusted ID evaluator to create the trusted ID list for evaluation.

To specify the name and value properties, complete the following steps:

1. Under Additional properties, click **Properties** > **New**.
2. Specify the trusted ID evaluator name as a property name. You must specify the trusted ID evaluator name in the form, `trustedId_n`, where `_n` is an integer from zero (0) to n.
3. Specify the trusted ID as a property value.

For example:

```
property name="trustedId_0", value="CN=Bob,O=ACME,C=US"
property name="trustedId_1", value="user1"
```

If a distinguished name (DN) is used, the space is removed for comparison.

Information	Value
Default	`com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl`

See the programming model information in the documentation for an explanation of how to implement the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` interface.

rrdSecurity.props file:

Remote request dispatcher (RRD) supports LTPA and security attribute propagation for Web Services Security (WS-Security). You can enable token propagation in the `<was_install>/profiles/<profileName>/properties/rrdSecurity.props` file.

The `rrdSecurity.props` file contains comments to describe the security attributes.

The following is the format of the `rrdSecurity.props` file. The default values are in bold face type.

- `LTPAPropagation= (` **`True`** `| False)`

- `SecurityAttributePropagation= ( True | **False**)`
- `SSLRequired= ( **True** | False)`

The WS-Security run time inspects the run as (invocation) subject and propagates the security tokens in the subject. The default setting is to only propagate the LTPA tokens.

Custom security tokens can be passed as attributes of the LTPA tokens. The security attribute propagation support uses the same pluggable JAAS login module as the CSIv2 support. The security attribute is not signed or encrypted, therefore, you should not send the attribute in clear text form. You must require SSL to ensure integrity and confidentiality. If SSL is not required, RRD uses the same scheme, such as HTTP or HTTPS, to make the web services call that the original request used.

You must also configure the target web service to validate the LTPA tokens and security attributes.

Enabling or disabling single sign-on interoperability mode for the LTPA token:

You can set an interoperability flag on the token generator to determine whether an LTPA Version 1 token or an LTPA Version 2 token is retrieved when a request message is received.

About this task

In WebSphere Application Server Version 7.0 and later, a flag is set in the global security settings to enable single sign-on interoperability mode for the LTPA token. This option determines whether an LTPA Version 1 token or an LTPA Version 2 token is sent when a message request is received. When the interoperability flag is set to `true`, then the AuthenticationToken is an LTPA Version 1 token, and the SingleSignonToken is an LTPA Version 2 token. When the interoperability flag is set to `false`, then both the AuthenticationToken and the SingleSignonToken are LTPA Version 2 tokens.

When the interoperability mode is enabled (the flag is set to `true`), and the Web Services Security binding configuration specifies LTPA Version 1 as the token, the AuthenticationToken is used to retrieve the token that is sent with the message. If interoperability mode is not enabled (the flag is set to `false`), and the Web Services Security binding configuration specifies LTPA Version 1 as the token, an exception error is logged.

You can disable the interoperability checking function by setting the custom property, com.ibm.wsspi.wssecurity.tokenGenerator.ltpav1.pre.v7, on the token generator. This setting determines the LTPA token without checking the state of the interoperability flag, providing compatibility with servers running WebSphere Application Server Version 6.1 and earlier.

To enforce use of the LTPA Version 2 token, edit the token settings, and set the **Enforce token version** option for the token.

Procedure

1. Click **Applications > Application Types > WebSphere enterprise applications**.
2. Select an application that contains web services. The application must contain a service provider or a service client.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** link in the Web Services Properties section.
4. Select a binding. You must have previously attached a policy set and assigned an application specific binding.
5. Click the **WS-Security** policy in the Policies table.
6. Click the **Authentication and protection** link in the Main message security policy bindings section.
7. Click a consumer or generator token link from the Protection Tokens table.
8. Select the **Enforce token version** check box after the **Token type** field.

Enabling cryptographic keys stored in hardware devices for Web Services Security

You can enable Web Services Security by using cryptographic hardware devices for both web service clients and web service providers that are running in the WebSphere® Application Server environment.

Enabling hardware cryptographic devices for Web Services Security:

You can enable Web Services Security by using cryptographic hardware devices for both web service clients and web service providers that are running in the WebSphere Application Server environment. A cryptographic token is a hardware or software device with a built-in keystore implementation. Cryptographic devices are used to manage certificates stored on the cryptographic tokens. These devices are also called *smartcards*. You enable hardware cryptographic devices for Web Service Security by either using keys that are stored in hardware devices or by using keys stored in a Java keystore file.

About this task

Web Services Security using cryptographic hardware devices is supported for both web (JavaServer Pages (JSP) or servlet) and Enterprise JavaBeans (EJB) web service clients. You can enable Web Services Security by using cryptographic hardware devices for both web service clients and web service providers that are running in the WebSphere Application Server environment.

There are two ways to enable hardware cryptographic devices for Web Service Security: use keys that are stored in hardware devices or use keys stored in a Java keystore file.

Procedure

1. Determine whether to use keys that are stored in hardware devices or in a Java keystore file for the individual application.
2. Enable hardware cryptographic devices for Web Service Security by using one of the following two methods:
   - Enable cryptographic operations on hardware devices. See "Configuring hardware cryptographic devices for Web Services Security" for more details.
   - Enable cryptographic keys that are stored in hardware devices. See "Enabling cryptographic keys stored in hardware devices in Web Services Security" on page 982

   Note: Hardware cryptographic devices for Web Services Security are not supported on the Java Platform, Enterprise Edition (Java EE) Application Client on distributed platform.

Configuring hardware cryptographic devices for Web Services Security:

Before you can use a hardware cryptographic device, you must configure and enable it. You must first configure a hardware cryptographic device using the Secure Sockets Layer (SSL) certificate and key management panels in the administrative console. The key for the cryptographic operation can be stored in an ordinary Java keystore file and need not be stored on the hardware devices. You enable cryptographic operations by performing specific file setup procedures to ensure that the cryptographic device can be used.

Before you begin

You must first configure a hardware cryptographic device using the Secure Sockets Layer (SSL) certificate and key management panels in the administrative console.

Note: Fix packs that include updates to the Software Development Kit (SDK) might overwrite unrestricted policy files. Back up unrestricted policy files before you apply a fix pack and reapply these files after the fix pack is applied.

Procedure

1. Stop the application server.
2. Download and install the new policy files.

   Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

   a. Click **Java SE 6**
   b. Scroll down the page then click **IBM SDK Policy files**.

      The Unrestricted JCE Policy files for SDK 6 Web site displays.
   c. Click **Sign in** and provide your IBM.com ID and password.
   d. Select **Unrestricted JCE Policy files for SDK 6** and click **Continue**.
   e. View the license and click **I Agree** to continue.
   f. Click **Download Now**.
   g. Extract the unlimited jurisdiction policy files that are packaged in the ZIP file. The ZIP file contains a `US_export_policy.jar` file and a `local_policy.jar` file.
   h. In your WebSphere Application Server installation, mount your product HFS as read/write. Go to the `$JAVA_HOME/lib/security` directory and back up your `US_export_policy.jar` and `local_policy.jar` files.
   i. Replace your `US_export_policy.jar` and `local_policy.jar` files with the two files that you downloaded from the `IBM.com` Web site.
   j. Re-mount your product HFS as read/only.

   Following is an example of this copy operation.

```
$JAVA_HOME/demo/jce/policy-files/unrestricted/* to
$JAVA_HOME/lib/security
```

   The embedded Software Development Kit (SDK) ships with the unrestricted jurisdiction policy Java archive (JAR) files. Therefore, instead of downloading these files from the Web site, you can symbolically link to the files as allowed by your local country regulations. These unrestricted policy files are located in the *install_root*`/java/demo/jce/policy-files/unrestricted/` directory. The following UNIX-based commands enable you to symbolically link to these files:

```
# Export the paths. You can find the values of the following
# variables in the joblog by searching for was.install.root,
# java.home, and so on:
export was.install.root=<was.install.root>
export java.home=<java.home>
# The previous paths apply to both 31- and 64-bit configurations
# of WebSphere Application Server for z/OS. For a 64-bit
# configuration, the java.home path points to the 64-bit embedded
# Java virtual machine (JVM).

# Delete the original policy .jar files. Because a backup is
# automatically present in the smpe.home HFS, an explicit
# backup is not needed:
cd $java.home/lib/security
rm US_export_policy.jar
rm local_policy.jar

# Issue the following commands on separate lines to create
# the symbolic links to the unrestricted policy files:
ln -s $java.home/demo/jce/policy-files/unrestricted/US_export_po licy.jar US_export_policy.jar
ln -s $java.home/demo/jce/policy-files/unrestricted/local_policy .jar local_policy.jar
```

   Issue the following UNIX-based commands to remove the symbolic links to the unrestricted policy files in the demo directory and link to the original files:

```
# Export the paths. You can find the values of the following
# variables in the joblog by searching for was.install.root,
# java.home, and so on:
export was.install.root=<was.install.root>
```

```
export java.home=<java.home>
export smpe.install.root=<smpe.install.root>
# The previous paths apply to both 31- and 64-bit configurations
# of WebSphere Application Server for z/OS. For a 64-bit
# configuration, the java.home path points to the 64-bit embedded
# Java virtual machine (JVM).

# Delete the current policy .jar files. You might want
# to back up the following files:
cd $java.home/lib/security
rm US_export_policy.jar
rm local_policy.jar

# Issue the following commands on separate lines to create
# symbolic links to the smpe HFS where the original files
# are kept:
ln -s $smpe.install.root/java/lib/security/US_export_policy.jar US_export_policy.jar
ln -s $smpe.install.root/java/lib/security/local_policy.jar local_policy.jar
```

3. Alter the `java.security` file.

   The `java.security` file is located in the *app_server_root*/`properties` directory.

   The following changes need to be made to this file:

   a. Uncomment the following line of the file:

```
#security.provider.1=com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
```

   b. Reorder the list of providers and preference orders as follows:

```
security.provider.1=com.ibm.crypto.hdwrCCA.provider.IBMJCECCA
#security.provider.1=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.2=com.ibm.crypto.provider.IBMJCE
security.provider.3=com.ibm.jsse.IBMJSSEProvider
security.provider.4=com.ibm.jsse2.IBMJSSEProvider2
security.provider.5=com.ibm.security.jgss.IBMJGSSProvider
security.provider.6=com.ibm.security.cert.IBMCertPath
security.provider.7=com.ibm.security.sasl.IBMSASL
security.provider.8=com.ibm.security.cmskeystore.CMSProvider
security.provider.9=com.ibm.security.jgss.mech.spnego.IBMSPNEGO
security.provider.9=com.ibm.xml.crypto.IBMXMLCryptoProvider
security.provider.10=com.ibm.xml.enc.IBMXMLEncProvider
security.provider.11=org.apache.harmony.security.provider.PolicyProvider
```

   The file structure and content are ready for use.

4. Start the application server. The cryptographic device is enabled for all Web service security applications that run on this application server.

Results

This procedure configures and enables a hardware cryptographic device for all Web Services Security applications running on this application server.

Enabling cryptographic keys stored in hardware devices in Web Services Security:

You can enable individual web service applications to use cryptographic keys stored in hardware devices in Web Services Security.

Before you begin

You must first configure the hardware acceleration device using the key management panels in the administrative console. See “Configuring hardware cryptographic devices for Web Services Security” on page 980

Procedure

1. In the administrative console, click **Servers** > **Server types** > **WebSphere application servers** and then select the server name.
2. Under **Security**, click **JAX-WS and JAX-RPC security runtime**.
3. Under **Additional properties**, click **key locators**.

4. Select the key locator name.
5. Under **Key store**, specify the name of the keystore configuration.

   If the keystore reference is specified to a hardware device configuration, the Web Services Security runtime first attempts to obtain the cryptographic algorithm from the hardware device. If the hardware device is not supported or if it fails, the runtime for Web Services Security obtains the cryptographic algorithm from the security providers list. Read about creating a keystore configuration for a preexisting keystore file for more information about how to create the name of a keystore configuration.
6. Click **OK**.

Results

If the name of the keystore reference is a Java keystore file, a hardware acceleration device that is configured at the application server level (`ws-security.xml`) will be used for cryptographic operations.

Configuring XML digital signature for Version 5.x web services with the administrative console

XML digital signature provides both message integrity and authentication capabilities when it is used with SOAP messages. XML digital signature is one of the methods WebSphere® Application Server provides to secure web services. You can use the WebSphere® Application Server administrative console to configure XML digital signature.

Login mappings collection:

Use this page to view a list of configurations for validating security tokens within incoming messages. Login mappings map an authentication method to a Java Authentication and Authorization Service (JAAS) login configuration to validate the security token. Four authentication methods are predefined in the WebSphere Application Server: BasicAuth, Signature, IDAssertion, and Lightweight Third Party Authentication (LTPA).

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications. Version 5.*x* applications are based on Java 2 platform, Enterprise Edition (J2EE) 1.3.

To view this administrative console page for the cell level, complete the following steps:
1. Click **Security** > **JAX-WS and JAX-RPC security runtime**
2. Under Additional properties, click **Login mappings**.
3. Click **New** to create a new login mapping or click an existing configuration to modify its settings.

To view this administrative console page for the server level, complete the following steps:
1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Login mappings**.
4. Click either **New** to create a new login mapping configuration or click the name of an existing configuration.

To view this administrative console page for the application level, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.

2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security properties, click **Web services: Server security bindings**.
4. Click **Edit** under Request receiver binding.
5. Click **Login mappings**.

If you click **Update runtime**, the Web Services Security run time is updated with the default binding information, which is contained in the `ws-security.xml` file that was previously saved. After you specify the authentication method, the JAAS configuration name, and the Callback Handler Factory class name on this panel, you must complete the following steps:

1. Click **Save** in the messages section at the top of the administrative console.
2. Click **Update runtime**. When you click **Update runtime**, the configuration changes made to the other web services also are updated in the Web Services Security run time.

Important: If the login mapping configuration is not found on the application level, the web services run time searches for the login mapping configuration on the server level. If the configuration is not found on the server level, the web services run time searches the cell.

Authentication method:

Specifies the authentication method used for validating the security tokens.

The following authentication methods are available:

BasicAuth
: The basic authentication method includes both a user name and a password in the security token. The information in the token is authenticated by the receiving server and is used to create a credential.

Signature
: The signature authentication method sends an X.509 certificate as a security token. For Lightweight Directory Access Protocol (LDAP) registries, the distinguished name (DN) is mapped to a credential, which is based on the LDAP certificate filter settings. For local OS registries, the first attribute of the certificate, usually the common name (CN) is mapped directly to a user name in the registry.

IDAssertion
: The identity assertion method maps a trusted identity (ID) to a WebSphere Application Server credential. This authentication method only includes a user name in the security token. An additional token is included in the message for trust purposes. When the additional token is trusted, the IDAssertion token user name is mapped to a credential.

LTPA
: Lightweight Third Party Authentication (LTPA) validates an LTPA token.

JAAS configuration name:

Specifies the name of the Java Authentication and Authorization Service (JAAS) configuration.

Callback handler factory class name:

Specifies the name of the factory for the CallbackHandler class.

Login mapping configuration settings:

Use this page to specify the Java Authentication and Authorization Service (JAAS) login configuration settings that are used to validate security tokens within incoming messages.

Important: There is an important distinction between Version 6 and later applications. The information in this article supports Version 6.*x* applications only that are used with WebSphere Application Server Version 6.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

To view this administrative console page for the cell level, complete the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**
2. Under Additional properties, click **Login mappings**.
3. Click either **New** to create a new login mapping configuration or click the name of an existing configuration.

To view this administrative console page for the server level, complete the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
3. Under Additional properties, click **Login mappings**.
4. Click either **New** to create a new login mapping configuration or click the name of an existing configuration.

To use this administrative console page for the application level, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_name***.
3. Under Web Services Security Properties, click **Web services: Server security bindings**.
4. Click **Edit** under Request receiver binding.
5. Click **Login mappings**.
6. Click either **New** to create a new login mapping configuration or click the name of an existing configuration.

Important: If the login mapping configuration is not found on the application level, the web services run time searches for the login mapping configuration on the server level. If the configuration is not found on the server level, the web services run time searches the cell.

Authentication method:

Specifies the method of authentication.

You can use any string, but the string must match the element in the service-level configuration. The following words are reserved and have special meanings:

BasicAuth
Uses both a user name and a password.

IDAssertion
Uses only a user name, but requires that additional trust is established on the receiving server using a TrustedIDEvaluator mechanism.

Signature
Uses the distinguished name (DN) of the signer.

LTPA
Validates a token.

JAAS configuration name:

Specifies the name of the Java Authentication and Authorization Service (JAAS) configuration.

Among the predefined system login configurations that you can use are the following:

system.wssecurity.IDAssertion
: Enables a version 6.*x* application to use identity assertion to map a user name to a WebSphere Application Server credential principal.

system.wssecurity.Signature
: Enables a version 6.*x* application to map a distinguished name (DN) in a signed certificate to a WebSphere Application Server credential principal.

system.LTPA_WEB
: Processes login requests that are used by the web container such as servlets and JavaServer Pages (JSP) files.

system.WEB_INBOUND
: Handles logins for web application requests, which include servlets and JavaServer Pages..

system.RMI_INBOUND
: Handles logins for inbound Remote Method Invocation (RMI) requests.

system.DEFAULT
: Handles the logins for inbound requests made by internal authentications and most of the other protocols except web applications and RMI requests.

system.RMI_OUTBOUND
: Processes RMI requests that are sent outbound to another server when the com.ibm.CSIOutboundPropagationEnabled property is `true`. This property is set in the CSIv2 authentication panel. To access the panel, click **Security** > **Global security**. Expand RMI/IIOP security, then click on **CSIv2 Outbound authentication**. To set the `com.ibm.CSIOutboundPropagationEnabled` property, select **Security attribute propagation**.

system.wssecurity.X509BST
: Verifies an X.509 binary security token (BST) by checking the validity of the certificate and the certificate path.

system.wssecurity.PKCS7
: Verifies an X.509 certificate with a certificate revocation list in a PKCS7 object.

system.wssecurity.PkiPath
: Verifies an X.509 certificate with a public key infrastructure (PKI) path.

system.wssecurity.UsernameToken
: Verifies basic authentication (user name and password).

These system login configurations are defined on the System logins panel, which is accessible by completing the following steps:

1. Click **Security** > **Global security**.
2. Expand Java Authentication and Authorization Service, then click **System logins**.

Attention: The predefined system login configurations are listed on the System logins configuration panel without the system prefix. For example, the system.wssecurity.UsernameToken configuration listed in the Java Authentication and Authorization Service (JAAS) configuration name option corresponds to the wssecurity.UsernameToken configuration that is on the System logins configuration panel.

You can use the following predefined application login configurations:

ClientContainer
: Specifies the login configuration that is used by the client container application, which uses the CallbackHandler API that is defined in the deployment descriptor of the client container.

WSLogin
: Specifies whether all applications can use the WSLogin configuration to perform authentication for the WebSphere Application Server security run time.

DefaultPrincipalMapping
: Specifies the login configuration used by Java 2 Connectors (J2C) to map users to principals that are defined in the J2C authentication data entries.

These application login configurations are defined on the Application logins panel, which is accessible by completing the following steps:

1. Click **Security** > **Global security**.
2. Expand Java Authentication and Authorization Service, then click **Application logins**.

Do not remove these predefined system or application login configurations. Within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module.

Callback handler factory class name:

Specifies the name of the factory for the CallbackHandler class.

You must implement the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` class in this field.

Token type URI:

Specifies the namespace Uniform Resource Identifiers (URI), which denotes the type of security token that is accepted.

If binary security tokens are accepted, the value denotes the ValueType attribute in the element. The ValueType element identifies the type of security token and its namespace. If Extensible Markup Language (XML) tokens are accepted, the value denotes the top-level element name of the XML token.

If the reserved words are specified previously in the Authentication method field, this field is ignored.

Information	Value
Data type:	Unicode characters except for non-ASCII characters, but including the number sign (#), the percent sign (%), and the square brackets ([ ]).

Token type local name:

Specifies the local name of the security token type, for example, X509v3.

If binary security tokens are accepted, the value denotes the ValueType attribute in the element. The ValueType attribute identifies the type of security token and its namespace. If Extensible Markup Language (XML) tokens are accepted, the value denotes the top-level element name of the XML token.

If the reserved words are specified previously in the Authentication method field, this field is ignored.

Nonce maximum age:

Specifies the time, in seconds, before the nonce timestamp expires. Nonce is a randomly generated value.

You must specify a minimum of 300 seconds for the Nonce maximum age field. However, the maximum value cannot exceed the number of seconds specified in the Nonce cache timeout field for either the cell level or the server level.

You can specify the Nonce maximum age value for the cell level by completing the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.

You can specify the Nonce maximum age value for the server level by completing the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

Important: The Nonce maximum age field on this panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value: `Nonce is not supported for authentication methods other than BasicAuth`.

If you specify the BasicAuth method, but do not specify values for the Nonce maximum age field, the Web Services Security run time searches for a Nonce maximum age value on the server level. If a value is not found on the server level, the run time searches the cell level. If a value is not found on either the server level or the cell level, the default is 300 seconds.

Information	Value
Default	300 seconds
Range	300 to Nonce cache timeout seconds

Nonce clock skew:

Specifies the clock skew value, in seconds, to consider when WebSphere Application Server checks the freshness of the message. Nonce is a randomly generated value.

You can specify the Nonce clock skew value for the cell level by completing the following steps:

1. Click **Security** > **JAX-WS and JAX-RPC security runtime**.

You can specify the **Nonce clock skew** value for the server level by completing the following steps:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

You must specify a minimum of zero (0) seconds for the Nonce Clock Skew field. However, the maximum value cannot exceed the number of seconds that is specified in the Nonce maximum age field on this Login mappings panel.

Important: The Nonce clock skew field on this panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value: `Nonce is not supported for authentication methods other than BasicAuth`.

Note: If you specify BasicAuth, but do not specify values for the Nonce clock skew field, WebSphere Application Server searches for a Nonce clock skew value on the server level. If a value is not found on the server level, the run time searches the cell level. If a value is not found on either the server level or the cell level, the default is zero (0) seconds.

Information	Value
Default	0 seconds
Range	0 to Nonce Maximum Age seconds

Configuring nonce using Web Services Security tokens:

Nonce is a randomly generated, cryptographic token that is used to thwart the highjacking of user name tokens, which are used with SOAP messages. Use nonce in conjunction with the BasicAuth authentication method.

About this task

Important: The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

You can configure nonce at the application level, the server level, and cell level.

If you configure nonce on the application level and the server level, the values specified for the application level take precedence over the values specified for the server level.

Likewise, the values specified for the application level take precedence over the values specified for the server level and cell level.

You must consider the order of precedence:
1. Application level
2. Server level
3. Cell level

Complete these high-level tasks in the order listed:

Procedure
1. Configure nonce for the application level.
2. Configure nonce for the server level.
3. Configure nonce for the cell level.

What to do next

After completing these steps, restart the server if it has not already been restarted.

Configuring nonce for the server level:

Nonce is a randomly generated, cryptographic token that is used to prevent the theft of username tokens, which are used with SOAP messages. Nonce is used in conjunction with the basic authentication (BasicAuth) method. You can configure nonce for the server level by using the WebSphere Application Server administrative console.

About this task

Important: The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

You can configure nonce at the application level, the server level, and cell level.

However, you must consider the order of precedence:

1. Application level
2. Server level
3. Cell level

If you configure nonce on the application level and the server level, the values specified for the application level take precedence over the values specified for the server level.

Likewise, the values specified for the application level take precedence over the values specified for the server level and the cell level.

In a WebSphere Application Server (base) or WebSphere Application Server, Express environment, you must specify values for the Nonce cache timeout, Nonce maximum age, and Nonce clock skew fields on the server level to use nonce effectively.

However, in a WebSphere Application Server, Network Deployment environment, these fields are optional on the server level, but required on the cell level.

Complete the following steps to configure nonce on the server level:

Procedure

1. Connect to the administrative console.

   Type `http://localhost:`*`port_number`*`/ibm/console` in your web browser unless you have changed the port number.
2. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_level***.
3. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
4. Specify a value, in seconds, for the Nonce cache timeout field. The value specified for the Nonce cache timeout field indicates how long the nonce remains cached before it is expunged. You must specify a minimum of 300 seconds. However, if you do not specify a value, the default is 600 seconds. This field is required for the server level.

   However, in a Network deployment environment or on the z/OS platform, this field is optional on the server level, but required on the cell level.
5. Specify (optional) a value, in seconds, for the Nonce maximum age field.

   The value specified for the Nonce Maximum Age field indicates how long the nonce is valid. You must specify a minimum of 300 seconds, but the value cannot exceed the number of seconds specified for the Nonce cache timeout field on the server level.

   The value specified for the Nonce maximum age field must not exceed the Nonce maximum age value set on the cell level. You can specify the Nonce cache timeout value for the cell level by clicking **Security** > **Web Services**. This field is optional on the server level, but required on the cell level.
6. Specify a value, in seconds, for the Nonce clock skew field. The value specified for the Nonce clock skew field specifies the amount of time, in seconds, to consider when the message receiver checks the timeliness of the value. Consider the following information when you set this value:

- Difference in time between the message sender and the message receiver if the clocks are not synchronized.
- Time needed to encrypt and transmit the message.
- Time needed to get through network congestion.

You must specify at least 0 seconds for the Nonce clock skew field. However, the maximum value cannot exceed the number of seconds specified in the Nonce maximum age field on the server level. If you do not specify a value, the default is 0 seconds.

7. Restart the server. If you change the Nonce cache timeout value and do not restart the server, the change is not recognized by the server.

Configuring nonce for the application level:

Nonce is a randomly generated, cryptographic token that is used to thwart the highjacking of Username tokens, which are used with SOAP messages. Use nonce in conjunction with the basic authentication (BasicAuth) method. You can configure nonce for the application level by using the WebSphere Application Server administrative console.

About this task

Important: The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

You can configure nonce at the application level, the server level, and cell level.

However, you must consider the order of precedence:
1. Application level
2. Server level
3. Cell level

If you configure nonce on the application level and the server level, the values specified for the application level take precedence over the values specified for the server level.

Likewise, the values specified for the application level take precedence over the values specified for the server level and cell level.

Procedure
1. Connect to the administrative console.

   Type `http://localhost:`*`port_number`*`/ibm/console` in your web browser unless you have changed the port number.
2. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
3. Under Manage modules, click ***URI_name***.
4. Under Web Services Security Properties, click **Web services: Server security bindings**.
5. Click **Edit** under Request receiver binding
6. Under Additional properties, click **Login mappings** > **New**.
7. Specify (optional) a value, in seconds, for the **Nonce maximum age** field. This panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value:

```
Nonce is not supported for authentication methods other than
BasicAuth.
```

If you specify BasicAuth, but do not specify values for the **Nonce maximum age** field, the Web Services Security runtime searches for a nonce maximum age value on the server level.

If a value is not found on the server level, the runtime searches the cell level. If a value is not found on either the server level or the cell level, the default is 300 seconds.

The value specified for the **Nonce maximum age** field indicates how long the nonce is valid. You must specify a minimum of 300 seconds; however, the value cannot exceed the number of seconds that is specified for the **Nonce cache timeout** field for the server level. Nor can it exceed the number of seconds specified for the **Nonce cache timeout** field for the cell level.

You can specify the nonce cache timeout value for the server level by completing the following steps:

a. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
b. Under Security, click **JAX-WS and JAX-RPC security runtime**.

Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

You can specify the nonce cache timeout value for the cell level by clicking **Security** > **Web services**.

8. Specify (optional) a value, in seconds, for the **Nonce clock skew** field. The value specified for the **Nonce clock skew** field specifies the amount of time, in seconds, to consider when the message receiver checks the timeliness of the value. This panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value:

```
Nonce is not supported for authentication methods other than
BasicAuth.
```

If you specify BasicAuth, but do not specify values for the **Nonce clock skew** field, the Web Services Security runtime searches for a Nonce clock skew value on the server level.

If a value is not found on the server level, the runtime searches the cell level. If a value is not found on either the server level or the cell level, the default is 0 seconds.

Consider the following information when you set this value:

- Difference in time between the message sender and the message receiver if the clocks are not synchronized.
- Time needed to encrypt and transmit the message.
- Time needed to get through network congestion.

9. Restart the server.

Configuring nonce for the cell level:

Nonce is a randomly generated, cryptographic token that is used to prevent the theft of username tokens, which are used with SOAP messages. Nonce is used in conjunction with the basic authentication (BasicAuth) method. You can configure nonce for the cell level by using the WebSphere Application Server administrative console.

About this task

Important: The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

You can configure nonce at the application level, the server level, and cell level. However, you must consider the order of precedence:

1. Application level
2. Server level
3. Cell level

If you configure nonce on the application level and the server level, the values specified for the application level take precedence over the values specified for the server level. Likewise, the values specified for the application level take precedence over the values specified for the server level and the cell level. In WebSphere Application Server, Network Deployment, the **Nonce cache timeout**, **Nonce maximum age**, and **Nonce clock skew** fields are required to use nonce effectively. However, these fields are optional on the server level. Complete the following steps to configure nonce on the cell level:

Procedure

1. Connect to the administrative console.

   Type `http://localhost:`*`port_number`*`/ibm/console` in your web browser unless you have changed the port number.
2. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
3. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
4. Specify a value, in seconds, for the **Nonce cache timeout** field. The value specified for the **Nonce cache timeout** field indicates how long the nonce remains cached before it is expunged. You must specify a minimum of 300 seconds. However, if you do not specify a value, the default is 600 seconds. This field is optional on the server level, but required on the cell level.
5. Specify a value, in seconds, for the **Nonce maximum age** field. The value specified for the **Nonce maximum age** field indicates how long the nonce is valid. You must specify a minimum of 300 seconds, but the value cannot exceed the number of seconds specified for the **Nonce cache timeout** field in the previous step. If you do not specify a value, the default is 600 seconds. In a WebSphere Application Server, Network Deployment environment or on the z/OS platform, if you specify a value on the server level for the **Nonce cache timeout** field, the value cannot exceed the value specified for on the cell level for the **Nonce cache timeout** field. This field is optional on the server level, but required on the cell level
6. Specify a value, in seconds, for the **Nonce clock skew** field. The value specified for the **Nonce clock skew** field specifies the amount of time, in seconds, to consider when the message receiver checks the freshness of the value. Consider the following information when you set this value:
   - Difference in time between the message sender and the message receiver if the clocks are not synchronized.
   - Time needed to encrypt and transmit the message.
   - Time needed to get through network congestion.

   At a minimum, you must specify 0 seconds in this field. However, the maximum value cannot exceed the number of seconds indicated in the **Nonce maximum age** field. If you do not specify a value, the default is 0 seconds. This field is optional on the server level but required on the cell level.
7. Restart the server. If you change the Nonce cache timeout value and do not restart the server, the change is not recognized by the server.

Configuring trust anchors using the administrative console:

Use the WebSphere Application Server administrative console to configure trust anchors that specify key stores which contain trusted root certificates to validate the signer certificate.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This document describes how to configure trust anchors or trust stores at the application level. It does not describe how to configure trust anchors at the server or cell level. Trust anchors defined at the application level have a higher precedence over trust anchors defined at the server or cell level. For more information on creating and configuring trust anchors at the server or cell level, see either "Configuring the server security bindings using an assembly tool" on page 627 or "Configuring the server security bindings using the administrative console" on page 1004.

You can configure an application-level trust anchor using an assembly tool or the administrative console. This document describes how to configure the application-level trust anchor using the administrative console.

About this task

A trust anchor specifies key stores that contain trusted root certificates, which validate the signer certificate. These key stores are used by the request receiver (as defined in the `ibm-webservices-bnd.xmi` file) and the response receiver (as defined in the `ibm-webservicesclient-bnd.xmi` file when web services are acting as client) to validate the signer certificate of the digital signature. The keystores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these keystores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `ibm-webservicesclient-bnd.xmi` file.

The following steps are for the client-side response receiver, which is defined in the `ibm-webservicesclient-bnd.xmi` file and the server-side request receiver, which is defined in the `ibm-webservices-bnd.xmi` file.

Procedure

1. Configure an assembly tool to work with a Java Platform, Enterprise Edition (Java EE) enterprise application. For more information, see the related information on Assembly Tools.
2. Create a web services-enabled Java EE enterprise application. See either "Configuring the server security bindings using an assembly tool" on page 627 or "Configuring the server security bindings using the administrative console" on page 1004 for an introduction on how to manage Web Services Security binding information on the server.
3. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***enterprise_application***.
4. Under Manage modules, click ***URI_name***.
5. Under Web Services Security Properties, click **Web services: client security bindings** to edit the response receiver binding information, if web services are acting as a client.
   a. Under Response receiver binding, click **Edit**.
   b. Under Additional properties, click **Trust anchors**.
   c. Click **New** to create a new trust anchor.
   d. Enter a unique name within the request receiver binding for the Trust anchor name field. The name is used to reference the trust anchor that is defined.
   e. Enter the key store password, path, and key store type.
   f. Click the trust anchor name link to edit the selected trust anchor.
   g. Click **Remove** to remove the selected trust anchor or anchors.

      When you start the application, the configuration is validated in the run time while the binding information is loading.
6. Return to the web services-enabled module panel accessed in step 2.
7. Under Web Services Security Properties, click **Web services: server security bindings** to edit the request receiver binding information.
   a. Under Request receiver binding, click **Edit**.

b. Under Additional properties, click **Trust anchors**.
c. Click **New** to create a new trust anchor

   Enter a unique name within the request receiver binding for the Trust anchor name field. The name is used to reference the trust anchor that is defined.

   Enter the key store password, path, and key store type.

   Click the trust anchor name link to edit the selected trust anchor.

   Click **Remove** to remove the selected trust anchor or anchors.

   When you start the application, the configuration is validated in the run time while the binding information is loading.

8. Save the changes.

Results

This procedure defines trust anchors that can be used by the request receiver or the response receiver (if the web services is acting as client) to verify the signer certificate.

Example

The request receiver or the response receiver (if the web service is acting as a client) uses the defined trust anchor to verify the signer certificate. The trust anchor is referenced using the trust anchor name.

What to do next

To complete the signing information configuration process for request receiver, complete the following tasks:

1. “Configuring the server for request digital signature verification: Verifying the message parts” on page 613
2. “Configuring the server for request digital signature verification: choosing the verification method” on page 614

To complete the process for the response receiver, if the web services is acting as client, complete the following tasks:

1. “Configuring the client for response digital signature verification: verifying the message parts” on page 620
2. “Configuring the client for response digital signature verification: choosing the verification method” on page 622

Configuring the client-side collection certificate store using the administrative console:

You can configure the client-side collection certificate store by using the administrative console.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

A *collection certificate store* is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using the assembly tools or the WebSphere Application Server administrative console. Complete the following steps to configure the client-side collection certificate store using the administrative console.

Procedure

1. Connect to the WebSphere Application Server administrative console.

   You can connect to the administrative console by typing `http://localhost:`*`port_number`*`/ibm/console` in your web browser unless you have changed the port number.
2. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
3. Under Manage modules, click *URI_name*.
4. Under Web Services Security Properties, click **Web services: Client security bindings** to add the collection certificate store to the client security bindings. If you do not see any entries, return to the assembly tool and configure the security extensions for either the client or the server.

   To configure the security extensions for the client, see the following topics:
   - "Configuring the client for response digital signature verification: verifying the message parts" on page 620
   - "Configuring the client for response digital signature verification: choosing the verification method" on page 622
5. Under Response receiver binding, click **Edit** to edit the client security bindings.
6. Click **Collection certificate store**.
7. Click a Certificate store name to edit an existing certificate store or click **New** to add a new certificate store name.
8. Enter a name in the Certificate store name field. The name entered in this field is a name that is referenced in the Certificate store field on the Signing information configuration page.
9. Leave the Certificate store provider field value as `IBMCertPath`.
10. Click **Apply**.
11. Under Additional properties, click **X.509 certificates** > **New**.
12. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`. If you have any additional certificate store paths to enter, click **New** and add the path names.
13. Click **OK**.

Configuring the server-side collection certificate store using the administrative console:

You can configure the collection certificate either by using an assembly tool or the WebSphere Application Server administrative console.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

A *collection certificate store* is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs is used to check the signature of a digitally signed SOAP message.

Complete the following steps to configure the server-side collection certificate store using the administrative console.

Procedure

1. Connect to the WebSphere Application Server administrative console.

   You can connect to the administrative console by typing `http://localhost:port_number/ibm/console` in your web browser unless you have changed the port number.
2. Click **Applications > Application Types > WebSphere enterprise applications > *application_name***.
3. Under Manage modules, click ***URI_name***
4. Under Web Services Security Properties, click **Web services: server security bindings** to add the collection certificate store to the server security bindings. If you do not see any entries, return to the assembly tool and configure the security extensions for the server.

   To configure the security extensions for the server, see the following topics:
   - "Configuring the server for request digital signature verification: Verifying the message parts" on page 613
   - "Configuring the server for request digital signature verification: choosing the verification method" on page 614
5. Click **Edit** under Request Receiver Binding to edit the server security bindings.
6. Click **Collection certificate store**.
7. Click a Certificate store name to edit an existing certificate store or click **New** to add a new certificate store name.
8. Enter a name in the Certificate store name field. The name entered in this field is a name that is referenced in the Certificate store field on the Signing information configuration page.
9. Leave the Certificate store provider field as `IBMCertPath`.
10. Click **Apply**.
11. Under Additional Properties, click **X.509 Certificates > New**.
12. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`. If you have any additional certificate store paths to enter, click **New** and add the path names.
13. Click **OK**.

Configuring default collection certificate stores at the server level in the WebSphere Application Server administrative console:

You can define a single collection certificate store for all of the applications that need to use the same certificates. Use the WebSphere Application Server administrative console to configure the default collection certificate store at the server level.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

A *collection certificate store* is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message. A certificate store typically refers to a certificate store located in the file system. The location of the certificate store can vary from machine to machine, so you might configure a default collection certificate store for a specific machine and reference it from within the signing information. The signing information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single collection certificate store for all of the applications that need to use the same certificates. You also can specify the default binding information at the cell level.

Complete the following steps to configure the default collection certificate store at the server level using the WebSphere Application Server administrative console:

Procedure

1. Connect to the administrative console.

   You can access the administrative console by typing `http://localhost:port_number/ibm/console` in your web browser unless you have changed the port number.
2. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
3. Under Security, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using Websphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
4. Under Additional properties, click **Collection certificate store**.
5. Enter a name in the **Certificate store name** field. This name is referenced in the **Certificate store** field on the Signing information configuration page.
6. Leave the **Certificate store provider** field value as `IBMCertPath`.
7. Click **Apply**.
8. Under Additional properties, click **X.509 certificates** > **New**.
9. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`.

   If you have any additional certificate store paths to enter, click **New** and add the path names.
10. Click **OK**.

Configuring default collection certificate stores at the cell level in the WebSphere Application Server administrative console:

A collection certificate store is a collection of non-root certificate authority (CA) certificates and certificate revocation lists (CRLs). Use this collection of CA certificates and CRLs to check the signature of a digitally signed SOAP message. A certificate store typically refers to a certificate store that is located in the file system.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

The location of the certificate store can vary from machine to machine, so you might configure a default collection certificate store for a specific machine and reference it from within the signing information. The signing information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single collection certificate store for all of the applications that need to use the same certificates.

You also can specify the default binding information at the server level.

Complete the following steps to configure the default collection certificate store at the cell level by using the WebSphere Application Server administrative console:

Procedure

1. Connect to the administrative console.

   You can access the administrative console by typing `http://localhost:port_number/ibm/console` in your web browser unless you have changed the port number.

2. Click **Security** > **Web services**.
3. Under Additional properties, click **Collection certificate store**.
4. Click the name of a certificate store name to edit an existing store, or click **New** to add a new store. This name is referenced in the Certificate store field on the Signing information configuration page.
5. Leave the Certificate store provider field value as `IBMCertPath`.
6. Click **Apply**.
7. Under Additional properties, click **X.509 certificates** > **New**.
8. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`

   If you have any additional certificate store paths to enter, click **New** and add the path names.
9. Click **OK**.

Configuring key locators using the administrative console:

You can configure binding information and key locators using the WebSphere Application Server administrative console.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task provides instructions on how to configure key locators using the WebSphere Application Server administrative console. You can configure binding information in the administrative console. You must use an assembly tool to configure extensions. The following steps are used to configure a key locator in the administrative console for a specific application:

Procedure

1. Open the administrative console.

   Type `http://localhost:`*`port_number`*`/ibm/console` in your web browser unless you have changed the port number.
2. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
3. Under Related Items, click either **Web Modules** or **EJB Modules**, depending on the type of module you are securing.
4. Click the name of the module you are securing.
5. Under Additional Properties, click either **Web services: Client security bindings** or **Web services: Server security bindings**, depending on whether you are adding the key locator to the client security bindings or to the server security bindings. If you do not see any entries, return to the assembly tool and configure the security extensions.
6. Edit the Request Sender Binding, Response Receiver Binding, Request Receiver Binding, or Response Sender Binding.
   - If you are editing your client security bindings, click **Edit** for either the Request Sender Binding or the Response Receiver Binding.
   - If you are editing your server security bindings, click **Edit** for either the Request Receiver Binding or the Response Sender Binding.
7. Click **Key Locators**.

8. Click **New** to configure a new key locator, select the box next to a key locator name and click **Delete** to delete a key locator, or click the name of a key locator to edit its configuration. If you are configuring a new key locator or editing an existing one, complete the following steps:
   a. Specify a name for the key locator in the **Key Locator Name** field.
   b. Specify a name for the key locator class implementation in the **Key Locator Classname** field. WebSphere Application Server has the following default key locator class implementations:

      com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator
      : This class is used by the response sender to map an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator` class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, `user1` is mapped to `mappedName_1`. Otherwise, `name="default"`. When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

      com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator
      : This class is used by the response receiver, the request sender, and the request receiver to map a name to an alias. Encryption uses this class to obtain a key to encrypt a message and digital signature uses this class to obtain a key to sign a message. The `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` class maps a logical name to a key alias in the key store file. For example, key #105115176771 maps to `CN=Alice, O=IBM, C=US`.
   c. Specify the password used to access the key store password in the **Key Store Password** field. This field is optional because the key locator does not use a key store.
   d. Specify the path name used to access the key store in the **Key Store Path** field. This field is optional because the key locator does not use a key store. Use `${USER_INSTALL_ROOT}` because this path expands to the WebSphere Application Server path on your machine.
   e. Select a keystore type from the **Key Store Type** field. This field is optional because the key locator does not use a key store. Use the `JKS` option if you are not using the Java Cryptography Extensions (JCE) policy and use `JCEKS` if you are using the JCE policy.

Configuring server and cell level key locators using the administrative console:

A key locator typically locates a key store in the file system. You can configure server and cell-level key locators for a specific application by using the WebSphere Application Server administrative console. You can configure binding information in the administrative console; however, for extensions, you must use an assembly tool.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

The location of key stores can vary from machine to machine so it is often helpful to configure a default key locator for a specific machine and reference it from within the encryption or signing information. This information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single key locator for all applications that need to use the same keys. In a WebSphere Application Server, Network Deployment environment, you also can specify the default binding information at the cell level.

Procedure

- Configure default key locators at the server level
  1. Open the administrative console.

     Type `http://localhost:port_number/ibm/console` in your web browser unless you have changed the port number.
  2. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
  3. Under Security, click **JAX-WS and JAX-RPC security runtime**.

     Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.
  4. Under Additional properties, click **Key locators**
  5. Click **New** to configure a new key locator. Select the box next to a key locator name and click **Delete** to delete a key locator; or click the name of a key locator to edit its configuration. If you are configuring a new key locator or editing an existing one, complete the following steps:
     a. Specify a name for the key locator in the **Key locator name** field.
     b. Specify a name for the key locator class implementation in the **Key locator class name** field.

        WebSphere Application Server has the following default key locator class implementations:

        com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator
        This class, used by the response sender, maps an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator` class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, `user1` is mapped to `mappedName_1`. Otherwise, `name="default"`. When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

        com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator
        This class, used by the response receiver, request sender, and request receiver, maps a name to an alias. Encryption uses this class to obtain a key to encrypt a message and digital signature uses this class to obtain a key to sign a message. The `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` class maps a logical name to a key alias in the key store file. For example, key #105115176771 is mapped to `CN=Alice`, `O=IBM`, `c=US`.
     c. Specify the password that is used to access the keystore password in the **Key store password** field.

        This field is optional is the key locator does not use a keystore.
     d. Specify the path name that is used to access the keystore in the **Key store path** field.

        This field is optional is the key locator does not use a keystore. Use `${USER_INSTALL_ROOT}` as this path expands to the WebSphere Application Server path on your machine.
     e. Select a keystore type from the **Key store type** field.

        This field is optional is the key locator does not use a keystore. Use the `JKS` option if you are not using the Java Cryptography Extensions (JCE) keystore type, and use `JCEKS` if you are using the JCE type.
- Configure default key locators at the cell level.
  1. Open the administrative console.

     Type `http://localhost:port_number/ibm/console` in your web browser unless you have changed the port number.
  2. Click **Security** > **Web services**.
  3. Under Additional properties, click **Key locators**.

4. Click **New** to configure a new key locator; select the box next to a key locator name and click **Delete** to delete a key locator; or click the name of a key locator to edit its configuration. If you are configuring a new key locator or editing an existing one, complete the following steps:
   a. Specify a name for the key locator in the **Key locator name** field.
   b. Specify a name for the key locator class implementation in the **Key locator class name** field.

      WebSphere Application Server has the following default key locator class implementations:

      com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator
      : This class, used by the response sender, maps an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator` class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, `user1` is mapped to `mappedName_1`. Otherwise, `name="default"`. When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

      com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator
      : This class, used by the response receiver, request sender, and request receiver, maps a name to an alias. Encryption uses this class to obtain a key to encrypt a message and digital signature uses this class to obtain a key to sign a message. The `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` class maps a logical name to a key alias in the key store file. For example, key #105115176771 is mapped to `CN=Alice`, `O=IBM`, `c=US`.
   c. Specify the password that is used to access the keystore password in the **Key store password** field.

      This field is optional is the key locator does not use a keystore.
   d. Specify the path name that is used to access the keystore in the **Key store path** field.

      This field is optional is the key locator does not use a keystore. Use `${`*`USER_INSTALL_ROOT`*`}` as this path expands to the WebSphere Application Server path on your machine.
   e. Select a keystore type from the **Key store type** field.

      This field is optional if the key locator does not use a keystore. Use the `JKS` option if you are not using the Java Cryptography Extensions (JCE) keystore type, and use `JCEKS` if you are using the JCE type.

Configuring the security bindings on a server acting as a client using the administrative console:

Use the web services client editor within an assembly tool to include the binding information, that describes how to run the security specifications found in the extensions, in the client enterprise archive (EAR) file.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

When configuring a client for Web Services Security, the bindings describe how to run the security specifications found in the extensions. Use the web services client editor within an assembly tool to include the binding information in the client enterprise archive (EAR) file.

You can configure the client-side bindings from a pure client accessing a web service or from a web service accessing a downstream web service. Complete the following steps to find the location in which to

edit the client bindings from a web service that is running on the server. When a web service communicates with another web service, you must configure client bindings to access the downstream web service.

Procedure

1. Deploy the web service using the WebSphere Application Server administrative console. Click **Applications** > **Install New Application**.

   You can access the administrative console by typing `http://localhost:`*`port_number`*`/ibm/console` in your web browser unless you have changed the port number.

   For more information, read about installing a new application.
2. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
3. Under Manage modules, click ***URI_name***.
4. Under Web Services Security Properties, click **Web Services: Client security bindings**. A table displays with the following columns:
   - Component Name
   - Port
   - Web Service
   - Request Sender Binding
   - Request Receiver Binding
   - HTTP Basic Authentication
   - HTTP SSL Configuration

   For Web Services Security, you must edit the request sender binding and response receiver binding configurations. You can use the defaults for some of the information at the server level and at the cell level in WebSphere Application Server, Network Deployment environments. Default bindings are convenient because you can configure commonly reused elements such as key locators once and then reference their aliases in the application bindings.
5. View the default bindings for the server using the administrative console by clicking **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** . Under Additional Properties, click **JAX-WS and JAX-RPC security runtime**.

   Note: In a mixed node cell with a server using WebSphere Application Server version 6.1 or earlier, click **Web services: Default bindings for Web Services Security**.

   You can configure the following sections. These topics are discussed in more detail in other sections of the documentation.
   - Request sender binding
     - “Signing parameter configuration settings” on page 343
     - “Encryption information configuration settings: Methods” on page 897
     - “Key locator configuration settings” on page 946
     - “Login bindings configuration settings” on page 1005
   - Response receiver binding
     - “Signing information configuration settings” on page 827
     - “Encryption information configuration settings: Message parts” on page 891
     - “Trust anchor configuration settings” on page 957
     - “Collection certificate store configuration settings” on page 965
     - “Key locator configuration settings” on page 946

What to do next

Important: When configuring the security request sender binding configuration, you must synchronize the information used to perform the specified security with the security request receiver binding configuration, which is configured in the server EAR file. These two configurations must be synchronized in all respects because there is no negotiation during run time to determine the requirements of the server. For example, when configuring the encryption information in the security request sender binding configuration, you must use the public key from the server for encryption. Therefore, the key locator that you choose must contain the public key from the server configuration. The server must contain the private key to decrypt the message. This example illustrates the important relationship between the client and server configuration. Additionally, when configuring the security response receiver binding configuration, the server must send the response using security information known by this client security response receiver binding configuration.

The following table shows the related configurations between the client and the server. The client request sender and the server request receiver are relative configurations that must be synchronized with each other. The server response sender and the client response receiver are related configurations that must be synchronized with each other. Note that related configurations are end points for any request or response. One end point must communicate its actions with the other end point because run time requirements are not required.

Table 190. Related configurations. The configurations must be synchronized with each other.

Client configuration	Server configuration
Request sender	Request receiver
Response receiver	Response sender

Configuring the server security bindings using the administrative console:

Use the WebSphere Application Server administrative console to edit bindings for a web service after these bindings are deployed on a server.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Create an Enterprise JavaBeans (EJB) file Java archive (JAR) file or web application archive (WAR) file containing the security binding file (`ibm-webservices-bnd.xmi`) and the security extension file (`ibm-webservices-ext.xmi`). If this archive is acting as a client to a downstream service, you also need the client-side binding file (`ibm-webservicesclient-bnd.xmi`) and the client-side extension file (`ibm-webservicesclient-ext.xmi`). These files are generated using the **`WSDL2Java`** command. For more information, read about the **`WSDL2Java`** command for JAX-RPC applications. You can edit these files using the Web Services Editor in the assembly tools. For more information, read about assembly tools.

When configuring server-side security for Web Services Security, the security extensions configuration specifies what security is to be performed while the security bindings configuration indicates how to perform what is specified in the security extensions configuration. You can use the defaults for some elements at the cell and server levels in the bindings configuration, including key locators, trust anchors, the collection certificate store, trusted ID evaluators, and login mappings and reference them from the WAR and JAR binding configurations.

The following steps describe how to edit bindings for a web service after these bindings are deployed on a server. When one web service communicates with another web service, you also must configure the client bindings to access the downstream web service.

Procedure

1. Deploy the web service using the WebSphere Application Server administrative console.

   Type `http://localhost:`*`port_number`*`/ibm/console` in your web browser unless you have changed the port number.

   After you log into the administration console, click **Applications** > **Install new application** to deploy the web service. For more information, read about installing enterprise application files with the console.
2. After you deploy the web service, click **Applications** > **Enterprise applications** > ***application_name*** .
3. Under Manage modules, click ***URI_name***.
4. Under Web Services Security Properties, click **Web services: client security bindings** for outbound requests and inbound responses. Click **Web services: server security bindings** for inbound requests and outbound responses.
5. If you click **Web services: server security bindings**, the following sections can be configured. These topics are discussed in more detail in other sections of the documentation.
   - Request receiver binding
     – Signing information
     – Encryption information
     – Trust anchors
     – Collection certificate store
     – Key locator
     – Trusted ID evaluator
     – Login mappings
   - Response sender binding
     – Signing parameters
     – Encryption information
     – Key locator

Configuring XML encryption for Version 5.x web services with the administrative console

XML encryption is one method that WebSphere® Application Server provides to secure web services. You can use XML encryption in conjunction with XML digital signature to scramble the content while verifying the authenticity of the message sender. Using XML encryption, you can encrypt an XML element, the content of an XML element, or arbitrary data such as an XML document.

Login bindings configuration settings:

Use this page to specify the Java Authentication and Authorization Service (JAAS) login configuration settings that are used to validate security tokens within incoming messages.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications. Version 5.*x* applications are based on Java 2 platform, Enterprise Edition (J2EE) 1.3.

The pluggable token uses the Java Authentication and Authorization Service (JAAS) CallBackHandler (javax.security.auth.callback.CallBackHandler) interface to generate the token that is inserted into the message. The following list describes the CallBack support implementations:

com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
This implementation is used for generating binary tokens inserted as `<wsse:BinarySecurityToken/ @ValueType>` in the message.

javax.security.auth.callback.NameCallback and javax.security.auth.callback.PasswordCallback
This implementation is used for generating user name tokens inserted as `<wsse:UsernameToken>` in the message.

com.ibm.wsspi.wssecurity.auth.callback.XMLTokenSenderCallback
This implementation is used to generate Extensible Markup Language (XML) tokens and is inserted as the `<SAML: Assertion>` element in the message.

com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback
This implementation is used to obtain properties that are specified in the binding file.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_file_name***. Under Web Services Security Properties, click **Web Services: Client security bindings**.
3. Under Request Sender Bindings, click **Edit**.
4. Under Additional properties, click **Login binding**.

If the encryption information is not available, select **None**.

If the encryption information is available, select **Dedicated login binding** and specify the configuration in the following fields:

Authentication method:

Specifies the unique name for the authentication method.

You can uses any string to name the authentication method. However, the string must match the element in the server-level configuration. The following words are reserved by WebSphere Application Server:

BasicAuth
This method uses both a user name and a password.

IDAssertion
This method uses a user name, but it requires that additional trust is established by the receiving server using a trusted ID evaluator mechanism.

Signature
This method uses the distinguished name (DN) of the signer.

LTPA This method validates the token.

Callback handler:

Specifies the name of the callback handler. The callback handler must implement the `javax.security.auth.callback.CallbackHandler` interface.

Basic authentication user ID:

Specifies the user name for basic authentication. With the basic authentication method, you can define a user name and a password in the binding file.

Basic authentication password:

Specifies the password for basic authentication.

Token type URI:

Specifies the namespace Uniform Resource Identifiers (URI), which denotes the type of security token that is accepted.

The value of this field if is impacted by the following conditions:
- If binary security tokens are accepted, the value denotes the ValueType attribute in the element. The ValueType element identifies the type of security token and its namespace.
- If Extensible Markup Language (XML) tokens are accepted, the value denotes the top-level element name of the XML token.
- The Token type URI field is ignored if the reserved words, which are listed in the description of the Authentication method field, are specified.

This information is inserted as `<wsse:BinarySecurityToken>/ValueType` for the <SAML: Assertion> XML token.

Token type local name:

Specifies the local name of the security token type. For example, X509v3.

The value of this field if is impacted by the following conditions:
- If binary security tokens are accepted, the value denotes the ValueType attribute in the element. The ValueType element identifies the type of security token and its namespace.
- If Extensible Markup Language (XML) tokens are accepted, the value denotes the top-level element name of the XML token.
- The Token type URI field is ignored if the reserved words, which are listed in the description of the Authentication method field, are specified.

This information is inserted as `<wsse:BinarySecurityToken>/ValueType` for the <SAML: Assertion> XML token.

Request sender binding collection:

Use this page to specify the binding configuration to send request messages for Web Services Security.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications. Version 5.*x* applications are based on Java 2 platform, Enterprise Edition (J2EE) 1.3.

To view this administrative console page, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_file_name***.
3. Under Web Services Security Properties, click **Web services: Client security bindings**.
4. Under Request sender binding, click **Edit**.

Web Services Security namespace: Specifies the namespace that is used by Web Services Security to send a request. However, this field configures the namespace value only and does not enforce the semantics of the specification related to the namespace. Web Services Security uses the processing semantic only in draft 13 of the OASIS specification. The following schemas are available:

- http://schemas.xmlsoap.org/ws/2003/06/secext
- http://schemas.xmlsoap.org/ws/2002/07/secext
- http://schemas.xmlsoap.org/ws/2002/04/secext
- None

The namespace used by the response sender is based on the namespace of the incoming message in the request receiver.

Signing information:

Specifies the configuration for the signing parameters. Signing information is used to sign and validate parts of the message including the body and time stamp.

You can also use these parameters for X.509 validation when the Authentication method is `IDAssertion` and the ID Type is `X509Certificate`, in the server-level configuration. In such cases, you must fill in the Certificate Path fields only.

Encryption information:

Specifies the configuration for the encrypting and decrypting parameters. Encryption information is used for encrypting and decrypting various parts of a message, including the body and user name token.

Key locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a keystore file or a repository. The key locator maps a name or a logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Login mappings:

Specifies a list of configurations for validating tokens within incoming messages.

Login mappings map the authentication method to the Java Authentication and Authorization Service (JAAS) configuration.

To configure JAAS, complete the following steps:

1. Click **Security** > **Global security**.
2. Under the Java Authentication and Authorization Service field, select **Application logins** or **System logins**.

Request receiver binding collection:

Use this page to specify the binding configuration to receive request messages for Web Services Security.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications. Version 5.*x* applications are based on Java 2 platform, Enterprise Edition (J2EE) 1.3.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
2. Under Modules, click **Manage modules** > ***URI_file_name***.
3. Under Web Services Security Properties, click **Web services: Server security bindings**.
4. Under Request receiver binding, click **Edit**.

Signing information:

Specifies the configuration for the signing parameters. Signing information is used to sign and validate parts of a message including the body, the timestamp, and the user name token.

You also can use these parameters for X.509 certificate validation when the authentication method is `IDAssertion` and the ID Type is `X509Certificate` in the server-level configuration. In such cases, you must fill in the Certificate Path fields only.

Encryption information:

Specifies the configuration for the encrypting and decrypting parameters. This configuration is used to encrypt and decrypt parts of the message that include the body and the user name token.

Trust anchors:

Specifies a list of keystore objects that contain the trusted root certificates that are issued by a certificate authority (CA).

The certificate authority authenticates a user and issues a certificate. The CertPath API uses the certificate to validate the certificate chain of incoming, X.509-formatted security tokens or trusted, self-signed certificates.

Collection certificate store:

Specifies a list of the untrusted, intermediate certificate files.

The collection certificate store contains a chain of untrusted, intermediate certificates. The CertPath API attempts to validate these certificates, which are based on the trust anchor.

Key locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a keystore file or a repository. The key locator maps a name or a logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Trusted ID evaluators:

Specifies a list of trusted ID evaluators that determine whether to trust the identity-asserting authority or message sender.

The trusted ID evaluators are used to authenticate additional identities from one server to another server. For example, a client sends the identity of user A to server 1 for authentication. Server 1 calls downstream to server 2, asserts the identity of user A, and includes the user name and password of server 1. Server 2 attempts to establish trust with server 1 by authenticating its user name and password and checking the trust based on the TrustedIDEvaluator implementation. If the authentication process and the trust check are successful, server 2 trusts that server 1 authenticated user A and a credential is created for user A on server 2 to invoke the request.

Login mappings:

Specifies a list of configurations for validating tokens within incoming messages.

Login mappings map the authentication method to the Java Authentication and Authorization Service (JAAS) configuration.

To configure JAAS, complete the following steps:
1. Click **Security** > **Global security**.
2. Under the Java Authentication and Authorization Service, click **Application logins** or **System logins**.

Response sender binding collection:

Use this page to specify the binding configuration for sender response messages for Web Services Security.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications. Version 5.*x* applications are based on Java 2 platform, Enterprise Edition (J2EE) 1.3.

To view this administrative console page, complete the following steps:
1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** .
2. Under Modules, click **Manage modules >** ***URI_file_name***.
3. Under Web Services Security Properties, click **Web services: Server security bindings**.
4. Under Response sender binding, click **Edit**.

Signing information:

Specifies the configuration for the signing parameters.

You also can use these parameters for X.509 certificate validation when the authentication method is IDAssertion and the ID Type is X509Certificate in the server-level configuration. In such cases, you must fill-in the Certificate Path fields only.

Encryption information:

Specifies the configuration for the encryption and decryption parameters.

Key locators:

Specifies a list of key locator objects that retrieve the keys for a digital signature and encryption from a keystore file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Response receiver binding collection:

Use this page to specify the binding configuration for receiver response messages for Web Services Security.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications. Version 5.*x* applications are based on Java 2 platform, Enterprise Edition (J2EE) 1.3.

To view this administrative console page, complete the following steps:

1. Click **Applications > Application Types > WebSphere enterprise applications***application_name*.
2. Under Modules, click **Manage modules >** ***URI_file_name*** **> Web Services: Client security bindings**.
3. Under Response receiver binding, click **Edit**.

Signing information:

Specifies the configuration for the signing parameters. Signing information is used to sign and to validate parts of the message including the body and the timestamp.

You can also use these parameters for X.509 validation when the authentication method is IDAssertion and the ID type is X509Certificate, in the server-level configuration. In such cases, you must fill in the certificate path fields only.

Encryption information:

Specifies the configuration for the encryption and decryption parameters.

Encryption information is used for encrypting and decrypting various parts of a message, including the body and the user name token.

Trust anchors:

Specifies a list of keystore objects that contain the trusted root certificates that are self-signed or issued by a certificate authority.

The certificate authority authenticates a user and issues a certificate. After the certificate is issued, the keystore objects, which contain these certificates, use the certificate for certificate path or certificate chain validation of incoming X.509-formatted security tokens.

Collection certificate store:

Specifies a list of the untrusted, intermediate certificate files.

The collection certificate store contains a chain of untrusted, intermediate certificates. The CertPath API attempts to validate these certificates, which are based on the trust anchor.

Key locators:

Specifies a list of key locator objects that retrieve the keys for a digital signature and encryption from a keystore file or a repository.

The key locator maps a name or a logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Configuring pluggable tokens using the administrative console:

You can configure the client-side request sender (`ibm-webservicesclient-bnd.xmi` file) or server-side request receiver (`ibm-webservices-bnd.xmi` file) by using the WebSphere Application Server administrative console.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, it is assumed that you have already created a web service that is based on the Java Platform, Enterprise Edition (Java EE) specification. See either of the following topics for an introduction of how to manage Web Services Security binding information for the server:

- "Configuring the server security bindings using an assembly tool" on page 627
- "Configuring the server security bindings using the administrative console" on page 1004

About this task

This document describes how to configure a pluggable token in the request sender (`ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file) and request receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` file).

Important: The pluggable token is required for the request sender and request receiver as they are a pair. The request sender and the request receiver must match for a request to be accepted by the receiver.

Prior to completing these steps, it is assumed that you deployed a web services-enabled enterprise application to the WebSphere Application Server.

Use the following steps to configure the client-side request sender (`ibm-webservicesclient-bnd.xmi` file) or server-side request receiver (`ibm-webservices-bnd.xmi` file) using the WebSphere Application Server administrative console.

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***enterprise_application***.
2. Under Modules, click **Manage modules** > ***URI_name***. The *URI* is the web services-enabled module.
   a. Under Web Services Security Properties, click **Web services: client security bindings** to edit the response sender binding information, if web services are acting as client.
      1) Under Response sender binding, click **Edit**.
      2) Under Additional Properties, click **Login binding**.
      3) Select **Dedicated login binding** to define a new login binding.
         a) Enter the authentication method, this must match the authentication method defined in IBM extension deployment descriptor. The authentication method must be unique in the binding file.
         b) Enter an implementation of the JAAS `javax.security.auth.callback.CallbackHandler` interface.
         c) Enter the basic authentication information (User ID and Password) and the basic authentication information is passed to the construct of the CallbackHandler implementation. The usage of the basic authentication information is up to the implementation of the CallbackHandler.
         d) Enter the token value type, it is optional for BasicAuth, Signature and IDAssertion authentication methods but required for any other authentication method. The token value type is inserted into the <wsse:BinarySecurityToken>@ValueType for binary security token and used as the namespace of the XML based token.
         e) Click **Properties**. Define the property with name and value pairs. These pairs are passed to the construct of the CallbackHandler implementation as `java.util.Map`.

Select **None** to deselect the login binding.

b. Under Web Services Security Properties, click **Web services: server security bindings** to edit the request receiver binding information.
   1) Under Request Receiver Binding, click **Edit**.
   2) Under Additional Properties, click **Login mappings**.
   3) Click **New** to create new login mapping.
      a) Enter the authentication method, this must match the authentication method defined in the IBM extension deployment descriptor. The authentication method must be unique in the login mapping collection of the binding file.
      b) Enter a JAAS Login Configuration name. The JAAS Login Configuration must be defined under **Security** > **Global security**. Under Authentication, click **Java Authentication and Authorization Service** > **Application logins**. For more information, read about configuring programmatic logins for Java Authentication and Authorization Service.
      c) Enter an implementation of the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface. This is a mandatory field.
      d) Enter the token value type, it is optional for BasicAuth, Signature and IDAssertion authentication methods but required for any other authentication method. The token value type is used to validate against the <wsse:BinarySecurityToken>@ValueType for binary security token and against the namespace of the XML based token.
      e) Enter the name and value pairs for the “Login Mapping Property” by clicking **Properties** . These name and value pairs are available to the JAAS Login Module or Modules by `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback` JAAS Callback. **Note**: This is true when editing existing login mappings but not when creating new login mappings.
      f) Enter the name and value pairs for the “Callback Handler Factory Property”, this name and value pairs is passed as `java.util.Map` to the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory.init()` method. The usage of these name and value pairs is up to the `CallbackHandlerFactory` implementation.

c. Click authentication method link to edit the selected login mapping.

d. Click **Remove** to remove the selected login mapping or mappings.

3. Click **Save**.

Results

The previous steps define how to configure the request sender to create security tokens in the SOAP message and the request receiver to validate the security tokens found in the incoming SOAP message. WebSphere Application Server supports pluggable security tokens.

You can use the authentication method defined in the login bindings and login mappings to generate security tokens in the request sender and validate security tokens in the request receiver.

What to do next

After you have configured pluggable tokens, you must configure both the client and the server to support pluggable tokens. See the following topics to configure the client and the server:

- “Configuring the client for LTPA token authentication: specifying LTPA token authentication” on page 666
- “Configuring the client for LTPA token authentication: collecting the authentication method information” on page 667
- “Configuring the server to handle LTPA token authentication information” on page 668
- “Configuring the server to validate LTPA token authentication information” on page 669

Deploying applications that use SAML

After SAML policy sets and bindings have been configured, and SAML tokens created, the SAML token information can be sent from the original login server to other servers using the SAML propagation feature. You can also extract SAML attributes from an existing SAML token and then create additional tokens using the extracted attributes.

About this task

Use the SAML propagation feature of WebSphere Application Server to send SAML token information based on the original login to other servers using a SAML token. Four propagation methods are provided. You can propagate the original SAML token, the SAML token identity and attributes, the WSCredential and WSPrincipal information, or a pre-existing SAML token inserted in the RequestContext.

When SAML is installed on a WebSphere server, you can create SAML attributes using the SAML runtime API. The SAML attributes are added to a CredentialConfig object, which is used to generate a SAML token. The API also provides a function that extracts SAML attributes from an existing SAML token and processes the attributes.

The following topics provide more information about deploying SAML applications.

Propagating SAML tokens

You can use various SAML token propagation methods to include SAML tokens in outbound web services messages.

About this task

A web services client can include two types of tokens in outbound web services messages:

- Original SAML tokens the client received from inbound web services messages.
- New self-issued SAML tokens.

New SAML tokens can be generated using attributes from the original SAML tokens, or using attributes from the WSPrincipal user name in the RunAs Subject. The web services policy configuration determines which SAML tokens will be propagated. You can override the policy configuration by programmatically inserting SAML tokens that you want to propagate into the Axis2 RequestContext object.

Four propagation methods are enabled. This table summarizes the propagation methods and the associated binding options:

Table 191. Propagation methods and associated binding options. Use propagation to include SAML tokens in web services messages.

SAML token propagation method	Binding option	Implementation details
Propagate the original SAML token.	The tokenRequest binding option is set to the value, `propagation`.	Sends the original SAML token from the server where the token was received to other servers using WS-Security.
Propagate the user security name, unique security name, group IDs, and security realm name.	The tokenRequest binding option is set to the value, `issueByWSCredential`.	Overrides the default system implementation. The self-issued SAML token contains user security name, user unique security name, group IDs, and security realm name that are specified by the WSCredential object in user security context.
Propagate the SAML token identity and attributes.	No binding option is set.	Default system implementation. The server self-generates a new SAML token containing the original SAML attributes, Authentication method, and NameIdentifier or SAML NameID, and sends the new self-generated SAML token to downstream servers using WS-Security. The new SAML token issuer name, issuer signing certificate, and lifetime are determined by the SAML provider configuration properties.

Table 191. Propagation methods and associated binding options (continued). Use propagation to include SAML tokens in web services messages.

SAML token propagation method	Binding option	Implementation details
Propagate the WSPrincipal.	The tokenRequest binding option is set to the value, `issueByWSPrincipal`.	Overrides the default system implementation. The self-issued SAML token contains WSPrincipal information in the RunAs subject. The information is stored as NameIdentity or NameID without copying anything from the original SAML token, even if the token exists in the subject.
Programmatically propagate a pre-existing SAML token.	Insert the SAML token that you want to propagate into the RequestContext using the property, `com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.`*`SAMLTOKEN_IN_MESSAGECONTEXT`*.	Overrides all other existing binding options.

Procedure

1. Propagate the original SAML token by setting the tokenRequest binding option to the value, `propagation`, in the bindings.xml file, as shown in the steps. This method sends the original SAML token to other servers using WS-Security. In order for the propagation to succeed, there must be a valid SAML token in the RunAs subject. The server extracts the SAML token from the RunAs subject in the current security context and validates the following conditions. If any of these conditions are invalid, the WS-Security runtime environment does not propagate the SAML token, and the propagation request fails.
   - The SAML token has not expired, and the expiration time is within the time window of the notOnOrAfter value.
   - The ConfirmationMethod setting in the SAML token is the same as the confirmationMethod binding option defined in the token generator configuration.
   - The token ValueType in the SAML token matches the ValueType in the token generator configuration.

   Perform these steps to set the correct value for the tokenRequest binding option. This procedure assumes that a web services client application named JaxWSServicesSamples is deployed, and that the Saml Bearer Client sample binding is attached.
   a. Click **Applications > Application types > WebSphere enterprise Applications >** ***JaxWSServicesSamples*** **> Service client policy sets and bindings >** ***Saml Bearer Client sample*** **> WS-Security > Authentication and protection**.
   b. Click **gen_saml11token** in the Authentication tokens table.
   c. Click **Callback handler**.
   d. Add the custom property `tokenRequest` and set the property value to `propagation`.
2. To propagate the SAML token identity and attributes using a self-issuing SAML token, modify the outbound tokenGenerator in the bindings.xml file. This method sends the original SAML attributes, NameIdentifier or NameID, and authentication method from the original SAML token to other servers using WS-Security. If there is no SAML token in the subject, the server uses the WSPrincipal, stored as NameIdentifier or NameID, to create a self-issued SAML token. This propagation method is the default system implementation. In this method, the binding option is not set.

   The following limitations apply to the bindings.xml file when you are using this propagation method:
   - Do not set the tokenRequest binding option in the bindings.xml file.
   - Do not set the stsURI binding option in the bindings.xml file, or set the option to this value: `www.websphere.ibm.com/SAML/Issuer/Self`.
3. To propagate the WSPrincipal, modify the bindings.xml file as shown in the steps. Set the tokenRequest binding option to the value, `issueByPrincipal`, in the bindings.xml file. Using this method, the self-issued SAML token is always based on the WSPrincipal even if there is a SAML token in the subject. The new SAML token contains the WSPrincipal user name as the NameId or NameIdentifier. The token does not contain any other attributes in the WSPrincipal or WSCredential objects.

   The following limitation applies to the bindings.xml file when you are using this propagation method:

- Do not set the stsURI binding option in the bindings.xml file, or set the option to the value, www.websphere.ibm.com/SAML/Issuer/Self.

Perform these steps to set the correct value for the tokenRequest binding option. This procedure assumes that a web services client application named JaxWSServicesSamples is deployed, and that the Saml Bearer Client sample binding is attached.

a. Click **Applications > Application types > WebSphere enterprise Applications > *JaxWSServicesSamples* > Service client policy sets and bindings > *Saml Bearer Client sample* > WS-Security > Authentication and protection**.
b. Click **gen_saml11token** in the Authentication tokens table.
c. Click **Callback handler**.
d. Add the custom property tokenRequest and set the property value to issueByPrincipal.

4. To propagate a pre-existing SAML token by inserting SAMLToken in the RequestContext, follow these steps. Use this method to send a SAML token that you created to downstream servers using WS-Security. The propagation is automatically triggered if the WS-Security runtime detects a SAML token in the RequestContext. The pre-existing token overrides any other existing binding options. To use this propagation method, save the existing SAML token in the RequestContext by specifying com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.*SAMLTOKEN_IN_MESSAGECONTEXT* as the key, as shown in the steps.
   a. Generate a SAML token using the method SAMLToken samlToken = <token type>, for example:

```
SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

   b. Save the SAMLToken to the RequestContext, for example:

```
Map requestContext = ((BindingProvider)port).getRequestContext();
   requestContext.put("com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.SAMLTOKEN_IN_MESSAGECONTEXT", samlToken );
```

   This propagation option can co-exist with the other propagation methods, and overrides the other methods. If the SAML token in the RequestContext is expired, or the token expiration time is less than current time plus the cache cushion, the WS-Security runtime environment ignores the SAML token, and uses one of the other three propagation methods that is configured in the bindings.xml file. To avoid using the other three propagation methods, add the following binding option to the custom properties under callback handler in the TokenGenerator configuration: failOverToTokenRequest = false.

5. To propagate a user's group memberships, unique security name, and realm name contained in the com.ibm.websphere.security.cred.WSCredential object, modify the bindings.xml file as shown in the steps. Set the tokenRequest binding option to the value, issueByWSCredential, in the bindings.xml file. Using this method, the self-issued SAML token is always based on the WSCredential even if there is a SAML token in the subject.

   The new SAML 1.1 token contains the following assertions:

   - The NameIdentifier element contains the SecurityName value from WSCredential with the NameQualifier element set to the realm name from WSCredential. The SecurityName is obtained by calling the WSCredential.getSecurityName() method. The realm name is obtained by calling the WSCredential.getRealmName() method.
   - All attributes have an AttributeNamespace set to com.ibm.websphere.security.cred.WSCredential as the value.
   - The GroupIds attribute contains all group names that a user belongs to. The group names are obtained by calling the WSCredential.getGroupIds() method.
   - The UniqueSecurityName attribute contains the unique security name, which is obtained by calling the WSCredential.getUniqueSecurityName() method.
   - Optionally, you can assert the realm name from WSCredential by adding the includeRealmName=true custom property in the callback handler.

   The new SAML 2.0 token contains the following assertions:

- The NameID element contains the SecurityName value from WSCredential with the NameQualifier element set to the realm name from WSCredential. The SecurityName is obtained by calling the WSCredential.getSecurityName() method. The realm name is obtained by calling the WSCredential.getRealmName() method.
- All attributes have a NameFormat set to com.ibm.websphere.security.cred.WSCredential as the value.
- The GroupIds attribute contains all group names that a user belongs to. The group names are obtained by calling the WSCredential.getGroupIds() method.
- The UniqueSecurityName attribute contains the unique security name, which is obtained by calling the WSCredential.getUniqueSecurityName() method.
- Optionally, you can assert the realm name from WSCredential by adding the includeRealmName=true custom property in the callback handler.

The following limitation applies to the bindings.xml file when you use the propagation method:

- Do not set the stsURI binding option in the bindings.xml file.

Perform these steps to set the correct value for the tokenRequest binding option. This procedure assumes that a Web services client application named JaxWSServicesSamples is deployed, and that the Saml Bearer Client sample binding is attached.

a. Click **Applications > Application types > WebSphere enterprise Applications > *JaxWSServicesSamples* > Service client policy sets and bindings > *Saml Bearer Client sample* > WS-Security > Authentication and protection.**
b. Click **gen_saml11token** in the Authentication tokens table.
c. Click **Callback handler**.
d. Add the tokenRequest custom property and set the property value to issueByWSCredential.

The following example illustrates the NameIdentifier and Attribute statement from a self-issued SAML 1.1 assertion based on WSCredential.

```
<saml:AttributeStatement>
    <saml:Subject>
        <saml:NameIdentifier NameQualifier="ldap.acme.com:9080">uid=alice,dc=acme,dc=com</saml:NameIdentifier>
        <saml:SubjectConfirmation>
            <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:bearer</saml:ConfirmationMethod>
        </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Attribute AttributeName="UniqueSecurityName" AttributeNamespace="com.ibm.websphere.security.cred.WSCredential">
        <saml:AttributeValue>uid=alice,dc=acme,dc=com</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute AttributeName="GroupIds" AttributeNamespace="com.ibm.websphere.security.cred.WSCredential">
        <saml:AttributeValue>cn=development,dc=acme,dc=com</saml:AttributeValue>
        <saml:AttributeValue>cn=deployment,dc=acme,dc=com</saml:AttributeValue>
        <saml:AttributeValue>cn=test,dc=acme,dc=com</saml:AttributeValue>
    </saml:Attribute>
</saml:AttributeStatement>
```

The following example illustrates the NameID and Attribute statement from a self-issued SAML 2.0 assertion based on WSCredential.

```
<saml2:AttributeStatement>
    <saml2:Attribute Name="UniqueSecurityName"
          NameFormat="com.ibm.websphere.security.cred.WSCredential">
        <saml2:AttributeValue>uid=alice,dc=acme,dc=com</saml2:AttributeValue>
    <saml2:Attribute>
    <saml2:Attribute AttributeName="GroupIds"
          NameFormat="com.ibm.websphere.security.cred.WSCredential">
        <saml2:AttributeValue>cn=development,dc=acme,dc=com</saml2:AttributeValue>
        <saml2:AttributeValue>cn=deployment,dc=acme,dc=com</saml2:AttributeValue>
        <saml2:AttributeValue>cn=test,dc=acme,dc=com</saml2:AttributeValue>
    </saml2:Attribute>
</saml2:AttributeStatement>

<saml2:NameID NameQualifier="ldap.acme.com:9060">alice</saml2:NameID>
```

Creating SAML attributes in SAML tokens

Using the SAML runtime API, you can create SAML tokens containing SAML attributes. You can also extract the SAML attributes from an existing SAML token.

About this task

Using WebSphere Application Server, you can create SAML attributes using the SAML token library APIs. The SAML attributes are added to a CredentialConfig object, which is used to generate a SAML token. The API also provides a function that extracts SAML attributes from an existing SAML token and processes the attributes.

To create a SAML token containing SAML attributes, perform the following steps:

Procedure

1. Initialize a com.ibm.wsspi.wssecurity.saml.data.SAMLAttribute object. This creates a SAML attribute based on an address, for example:

```
SAMLAttribute sattribute =
        new SAMLAttribute("urn:oid:2.5.4.20", //Name
      new String[] {" any address"}, //Attribute Values
      null,  /*XML Attributes empty on this example*/
      "urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500", //NameSpace
      "urn:oasis:names:tc:SAML:2.0:attrname-format:uri",  //format
      "Address");
```

2. Use the SAMLTokenFactory to create a CredentialConfig object containing a SAML attribute. This method requires the Java security permisson wssapi.SAMLTokenFactory.newCredentialConfig.
   a. Create a com.ibm.wsspi.wssecurity.saml.config.CredentialConfig object and set a valid principal name.
   b. Create a SAML attribute.
   c. Create a list of SAML attributes and add the SAML attribute to the list.
   d. Add the SAML attribute list to the CredentialConfig object.

   See the following example:

```
SAMLTokenFactory samlFactory =
  SAMLTokenFactory.getInstance("http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0");//samlTokenType

CredentialConfig credentialConfig = samlFactory.newCredentialConfig();
credentialConfig.setRequesterNameID("any name");

SAMLAttribute sattribute =
        new SAMLAttribute("urn:oid:2.5.4.20", //Name
      new String[] {" any address"}, //Attribute Values
      null,  /*XML Attributes empty on this example*/
      "urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500", //NameSpace
      "urn:oasis:names:tc:SAML:2.0:attrname-format:uri",  //format
      "Address");

ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(sattribute);
credentialConfig.setSAMLAttributes(al);
```

3. Specifying the CredentialConfig as a parameter, use the com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory newSAMLToken method to create a SAML token containing the attributes. This step assumes that a RequesterConfig `reqData` object and a ProviderConfig `samlIssuerCfg` object have already been created. For more information on these objects, read about RequesterConfig and ProviderConfig.
   a. Obtain an instance of the SAMLTokenFactory.
   b. Create a SAML token using the newSAMLToken method from the SAMLTokenFactory, for example:

```
SAMLTokenFactory samlFactory =
  SAMLTokenFactory.getInstance("http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1");

SAMLToken aSamlToken = samlFactory.newSAMLToken(credentialConfig, reqData, samlIssuerCfg);
```

4. Optional: Extract SAML attributes from an existing SAML token. This step is useful to extract the SAML attributes from a received SAML token. You can use this step when a SAML assertion is received and the attributes contained in the assertion need to be processed.
   a. Invoke the getSAMLAttributes() method with the token as a parameter to obtain a list of the SAML attributes in the token. This method requires the Java security permission wssapi.SAMLToken.getSAMLAttributes.

b. Apply an iterator to the list.
c. Iterate through the list and perform any additional processing required for your application.

See the following example:

```
List<SAMLAttribute> aList = aSAMLToken.getSAMLAttributes();
java.util.Iterator<SAMLAttribute> i = aList.iterator();

while(i.hasNext()){

   SAMLAttribute anAttribute = i.next();

   //do something with namespace
   String namespace = anAttribute.getAttributeNamespace();

   //do something with name
   String name = anAttribute.getName();

   //do something with friendly name
   String friendlyName = anAttribute.getFriendlyName();

   //process sring attribute values
   String[] stringAttributeValues = anAttribute.getStringAttributeValue();

   //process XML attribute values
   XMLStructure[] xmlAttributeValues = (XMLStructure[]) anAttribute.getXMLAttributeValue();

   }
```

SAML user attributes:

A SAML assertion can contain user attributes relating to the principal of the SAML token. A SAML assertion can contain multiple user attributes.

You can include user attributes in the token to communicate the address of the person who is the SAML assertion principal. This example shows a SAML assertion containing a user attribute:

```
<saml:AttributeStatement>
  <saml:Attribute  xmlns:x500=
      "urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500"
    NameFormat=
      "urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
    Name="urn:oid:2.5.4.20"
    FriendlyName="Address">
          <saml:AttributeValue xsi:type="xs:string">
                      11111 Parker Lane, Austin, Texas, 78758
          </saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

This table describes the parameters used in the assertion:

Parameter	Description
NameFormat	Specifies how the attribute is interpreted.
Name	Indicates the formal name of the attribute.
FriendlyName	Provides a user-friendly name for an attribute when the Name parameter is cryptic.
AttributeValue	The value of the user attribute. The value can be a string, or a complex XML type.

Establishing security context for web services clients using SAML security tokens

WebSphere Application Server supports two policy set caller binding configuration options to establish client security context using SAML security tokens in web services SOAP request messages. The two configuration options are mapping SAML tokens to a user entry in a local user repository and, asserting SAML tokens based on a trust relationship.

Before you begin

This task assumes that you are familiar with WebSphere Application Server SAML technology.

About this task

This task describes setting the WebSphere Application Server policy set caller binding configuration option to establish client security context using SAML security tokens in web services SOAP request messages. You can either map SAML tokens to a user entry in a local user repository or assert SAML tokens based on a trust relationship. The second configuration option does not require accessing the local user repository. Instead, the WS-Security runtime environment populates the client security context entirely using the contents of SAML security tokens. This process is based on a trust relationship to the SAML token issuer. If a SAML tokens specifies the sender-vouches subject confirmation method. the process is based on a trust relationship to the message sender.

Procedure

1. Configure a policy set caller binding and select the SAML token type to represent a web services client request.
   a. Click **WebSphere enterprise applications** > ***application_name*** > **Service provider policy sets and bindings** > ***binding_name*** > **WS-Security** > **Callers**.
   b. Click **New** to create the caller configuration.
   c. Specify a **Name**, such as `caller`.
   d. Enter a value for the **Caller identity local part**. For example, `http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0`, which must match the local part of the `CustomToken` element in the attached WS-Security policy.
   e. Click **Apply** and **Save**.
2. Optional: Map SAML security tokens to a user entry in a local user repository. Mapping to a user entry is the default behavior when you configure a caller binding without specifying a configuration option. Alternatively and optionally, you can select this configuration option explicitly using the following steps:
   a. On the caller binding configuration page, add a Callback handler: `com.ibm.websphere.wssecurity.callbackhandler.SAMLIdAssertionCallbackHandler`.
   b. Add a Callback handler custom property, `crossDomainIdAssertion`, and set its value to `false`.
3. Optional: Assert SAML security tokens based on trust relationship.
   a. On the caller binding configuration page, add a Callback handler: `com.ibm.websphere.wssecurity.callbackhandler.SAMLIdAssertionCallbackHandler`.
   b. Add a Callback handler custom property, `crossDomainIdAssertion`, and set its value to `true`.

   In WebSphere Application Server Version 7.0 Fix Pack 7 and later releases, the WS-Security runtime environment takes a SAML token Issuer name to represent the foreign security realm name. WS-Security takes the `NameID` element in the case of SAML 2.0 security tokens or the `NameIdentifier` element in the case of SAML 1.1 security tokens to represent user security name. Alternatively, you can explicitly specify which SAML token attribute to use to represent user security name. Moreover, you can also specify which SAML token attribute to use to represent user group membership. Read about SAML assertions across WebSphere Application Server security domains for a detailed discussion of the SAML token assertion trust model and binding configuration.

   Version 8.x supports propagating select security context data in SAML tokens. You must set a `tokenRequest` custom property with an `issueByWSCredential` property value in the WS-Security binding configuration of the web services client. Read about propagating SAML tokens for a detailed description of this binding option. When the `crossDomainIdAssertion` property is set to `true` in Version 8.x, WS-Security checks whether a SAML token contains a SAML Attribute `UniqueSecurityName` with a `NameFormat` element with a value of `com.ibm.websphere.security.cred.WSCredential`. If found, WS-Security uses the `NameQualifier` attribute value of the `NameID` element or `NameIdentifier` element to represent the user security realm name. WS-Security also uses the `UniqueSecurityName` attribute

value and the `GroupIds` attribute value to represent a unique user name and group membership. This default behavior is different between Version 7 and Version 8.x of the product. You can add a `CallbackHandler` property, `IssuerNameForRealm`, and set its value to `true` to configure Version 8.x to preserve the Version 7 behavior. Alternatively, you can add a `CallbackHandler` property, `NameQualifierForRealm`, and set its value to `true` to configure Version 8.x to always use the `NameQualifier` attribute to represent the user security realm name.

Results

You have configured a web service to establish a client security context using the SAML security token in the web services SOAP request messages.

Example

The following example illustrates the `NameIdentifier` and `Attribute` elements from a self-issued SAML 1.1 assertion based on `WSCredential`:

```
<saml:AttributeStatement>
    <saml:Subject>
        <saml:NameIdentifier NameQualifier="ldap.example.com:9080">uid=alice,dc=example,dc=com</saml:NameIdentifier>
        <saml:SubjectConfirmation>
            <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:bearer</saml:ConfirmationMethod>
        </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Attribute AttributeName="UniqueSecurityName" AttributeNamespace="com.ibm.websphere.security.cred.WSCredential">
        <saml:AttributeValue>uid=alice,dc=example,dc=com</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute AttributeName="GroupIds" AttributeNamespace="com.ibm.websphere.security.cred.WSCredential">
        <saml:AttributeValue>cn=development,dc=example,dc=com</saml:AttributeValue>
        <saml:AttributeValue>cn=deployment,dc=example,dc=com</saml:AttributeValue>
        <saml:AttributeValue>cn=test,dc=example,dc=com</saml:AttributeValue>
    </saml:Attribute>
</saml:AttributeStatement>
```

The following example illustrates the `NameID` and `Attribute` elements from a self-issued SAML 2.0 assertion based on `WSCredential`:

```
<saml2:AttributeStatement>
    <saml2:Attribute Name="UniqueSecurityName" NameFormat="com.ibm.websphere.security.cred.WSCredential" />
    <saml2:AttributeValue>uid=alice,dc=example,dc=com</saml2:AttributeValue>
    <saml2:Attribute>
        <saml2:Attribute AttributeName="GroupIds" NameFormat="com.ibm.websphere.security.cred.WSCredential" />
        <saml2:AttributeValue>cn=development,dc=example,dc=com</saml2:AttributeValue>
        <saml2:AttributeValue>cn=deployment,dc=example,dc=com</saml2:AttributeValue>
        <saml2:AttributeValue>cn=test,dc=example,dc=com</saml2:AttributeValue>
    </saml2:Attribute>
<saml2:AttributeStatement>

<saml2:NameID NameQualifier="ldap.example.com:9060">alice</saml2:NameID>
```

Tuning Web Services Security

When using Web Services Security for message-level protection of SOAP message in WebSphere Application Server, the choice of configuration options can affect the performance of the application.

Tuning Web Services Security for Version 8.5 applications

The Java Cryptography Extension (JCE) is integrated into the software development kit (SDK) Version 1.4.*x* and later. This is no longer an optional package. However, the default JCE jurisdiction policy file shipped with the SDK enables you to use cryptography to enforce this default policy. In addition, you can modify the web services security configuration options to achieve the best performance for web services security protected applications.

About this task

Using the unrestricted JCE policy files

Due to export and import regulations, the default JCE jurisdiction policy file shipped with the SDK enables you to use strong, but limited, cryptography only. To enforce this default policy, WebSphere Application Server uses a JCE jurisdiction policy file that might introduce a performance impact. The default JCE jurisdiction policy might have a performance impact on the cryptographic functions that are supported by Web Services Security. If you have web services applications that use transport level security for XML encryption or digital signatures, you might encounter performance degradation over previous releases of WebSphere Application Server. However, IBM and Oracle Corporation provide versions of these jurisdiction policy files that do not have restrictions on cryptographic strengths. If you are permitted by your governmental import and export regulations, download one of these jurisdiction policy files. After downloading one of these files, the performance of JCE and Web Services Security might improve.

Attention: Fix packs that include updates to the Software Development Kit (SDK) might overwrite unrestricted policy files. Back up unrestricted policy files before you apply a fix pack and reapply these files after the fix pack is applied.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

For WebSphere Application Server platforms using IBM Developer Kit, Java Technology Edition Version 6, you can obtain unlimited jurisdiction policy files by completing the following steps:

1. Go to the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html
2. Click **Java SE 6**
3. Scroll down and click **IBM SDK Policy files**.

   The Unrestricted JCE Policy files for the SDK website is displayed.
4. Click **Sign in** and provide your IBM intranet ID and password or register with IBM to download the files.
5. Select the appropriate Unrestricted JCE Policy files and then click **Continue**.
6. View the license agreement and then click **I Agree**.
7. Click **Download Now**.

Using configuration options to tune WebSphere Application Server

When using WS-Security for message-level protection of SOAP message in WebSphere Application Server, the choice of configuration options can affect the performance of the application. The following guidelines will help you achieve the best performance for your WS-Security protected applications.

1. Use WS-SecureConversation when appropriate for JAX-WS applications. The use of symmetric keys with a Secure Conversation typically performs better than asymmetric keys used with X.509.

   Note: The use of WS-SecureConversation is supported for JAX-WS applications only, not JAX-RPC applications.
2. Use the standard token types provided by WebSphere Application Server. Use of custom tokens is supported, but higher performance is achieved with the use of the provided token types.
3. For signatures, use only the exclusive canonicalization transform algorithm. See the W3 Recommendation web page (http://www.w3.org/2001/10/xml-exc-c14n#) for more information.
4. Whenever possible, avoid the use of the XPath expression to select which SOAP message parts to protect. The WS-Security policies shipped with WebSphere Application Server for JAX-WS applications use XPath expressions to specify the protection of some elements in the security header, such as Timestamp, SignatureConfirmation, and UsernameToken. The use of these XPath expressions is optimized, but other uses are not.

5. Although there are Websphere Application Server extensions to WS-Security that can be used to insert nonce and timestamp elements into SOAP message parts before signing or encrypting the message parts, you should avoid the use of these extensions for improved performance.
6. There is an option to send the base-64 encoded CipherValue of WS-Security encrypted elements as MTOM attachments. For small encrypted elements, the best performance is achieved by avoiding this option. For larger encrypted elements, the best performance is achieved by using this option.
7. When signing and encrypting elements in the SOAP message, specify the order as sign first, then encrypt.
8. When adding a timestamp element to a message, the timestamp should be added to the security header before the signature element. This is accomplished by using the `Strict` or `LaxTimestampFirst` security header layout option in the WS-Security policy configuration.
9. For JAX-WS applications, use the policy-based configuration rather than WSS API-based configuration.

What to do next

In IBM WebSphere Application Server Version 6.1 and later, Web Services Security supports the use of cryptographic hardware devices. There are two ways in which to use hardware cryptographic devices with Web Services Security. See “Hardware cryptographic device support for Web Services Security” on page 242 for more information.

Tuning Web Services Security for Version 5.x applications

The Java Cryptography Extension (JCE) policy is integrated into the IBM Software Development Kit (SDK) Version 1.4.*x* and is no longer an optional package. However, due to export and import regulations, the default JCE jurisdiction policy file shipped with the SDK enables you to use strong, but limited, cryptography only.

About this task

To enforce this default policy, WebSphere Application Server uses a JCE jurisdiction policy file that might introduce a performance impact. The default JCE jurisdiction policy might have a performance impact on the cryptographic functions that are supported by Web Services Security. If you have web services applications that use transport level security for XML encryption or digital signatures, you might encounter performance degradation over previous releases of WebSphere Application Server. However, IBM and Sun Microsystems provide versions of these jurisdiction policy files that do not have restrictions on cryptographic strengths. If you are permitted by your governmental import and export regulations, download one of these jurisdiction policy files. After downloading one of these files, the performance of JCE and Web Services Security might improve.

Securing WSIF

The steps to be taken to enable the Web Services Invocation Framework (WSIF) to interact with a security manager.

About this task

WSIF interacts with a security manager in the following ways:
- WSIF runs in the Java Platform, Enterprise Edition (Java EE) security context without modification.
- When WSIF is run under a Java EE container, port implementations can use the security context to pass on security tokens or credentials as necessary.
- WSIF implementations can automatically convert Java EE security context into appropriate context for onward services.

For WSIF to interact effectively with the WebSphere Application Server security manager, complete the following step:

Procedure

To load the Web Services Description Language (WSDL) file, enable the **FilePermission** attribute in the `was.policy` file. This permission is required when a WSDL file is referred to using the `file://` protocol.

Configuring UDDI registry security

The UDDI Version 3 registry uses the advantages of WebSphere Application Server security. The UDDI registry also supports the UDDI Version 3 Security API and the UDDI Version 1 and Version 2 security features.

About this task

For production use, it is advisable to configure the UDDI Version 3 registry to use WebSphere Application Server security. However, it is possible to configure the UDDI registry to use the UDDI security features if this is a requirement.

You can configure the UDDI registry to use the UDDI Version 3 security API or the UDDI Version 1 and Version 2 publish security features. For production use, you can configure the UDDI registry to use the UDDI security features with WebSphere Application Server security enabled. For test use, you can configure the UDDI registry to use the UDDI security features with WebSphere Application Server security disabled. Do not configure the UDDI registry to use the UDDI security features with WebSphere Application Server security disabled for production use.

To configure UDDI registry security, complete the following steps:

Procedure

1. Use one of the following procedures, depending on the type of configuration that you want to set up:
   - Configure the UDDI registry to use WebSphere Application Server security.
   - Configure UDDI Security with WebSphere Application Server security enabled.
   - Configure UDDI Security with WebSphere Application Server security disabled.
2. Review the "UDDI registry security and UDDI registry settings" on page 1028.

Configuring the UDDI registry to use WebSphere Application Server security

You can configure the UDDI registry to determine whether users are allowed access to services, and to determine security of data at the transport level.

Before you begin

- WebSphere Application Server administrative security must be enabled. For details, see the topic about enabling security.
- WebSphere Application Server must be configured to use HyperText Transport Protocol Secure Sockets Layer (HTTPS), to support secure access with the UDDI registry. By default, WebSphere Application Server is configured to accept Secure Sockets Layer (SSL) requests on port 9443. To make additional SSL configuration changes, see the topic about SSL configurations for selected scopes.

About this task

The UDDI registry uses two aspects of WebSphere Application Server security:

Authorization

Authorization determines whether users are allowed access to services. WebSphere Application Server determines authorization by mapping users, or groups of users, to roles. UDDI uses two

WebSphere Application Server special subjects: *Everyone* (all users are allowed access) and *AllAuthenticatedUsers* (only valid WebSphere Application Server registered users are allowed access).

Data confidentiality
Data confidentiality determines security at the transport level. Data confidentiality for WebSphere Application Server services can be either none, where HTTP is used as the transport protocol, or confidential, where the use of SSL is required and HTTPS is used as the transport protocol.

When WebSphere Application Server security is enabled, the default settings in the UDDI Version 3 Application and web deployment descriptors produce the following results:

- Publish, Custody Transfer, and Security services are mapped to the AllAuthenticatedUsers special subject, and data confidentiality is enforced through HTTPS. Authentication uses the standard WebSphere Application Server security facilities and the UDDI registry does not have a separate registration function. To use publish functions, users must supply their WebSphere Application Server user name and password (unless you modified the supplied publish role), and must also be registered UDDI publishers. By registering users as UDDI publishers, you control which users in the AllAuthenticatedUsers subject can update the UDDI registry.
- Inquiry services are mapped to the Everyone special subject, data confidentiality is not enforced, and HTTP is used. To use inquiry services, users do not have to supply a user name or password, and do not have to be registered UDDI publishers.

You can use the default settings, as described previously. To change the defaults, you map roles to different users or user groups. If you do this, enable the **Automatically register UDDI publishers** property of the UDDI node settings so that you do not have to use two mechanisms to give access to a subset of users. If you have a role that is not mapped to any users or user groups, all access to that role is disabled.

For more information about UDDI role mappings, and a list of UDDI registry services and roles, see the topic about access control for UDDI registry interfaces.

To change the default settings, use the following steps:

Procedure

- To change the role mappings by using the administrative console, complete the following steps:
  1. In the navigation pane, click **Applications** > **Application Types** > **WebSphere enterprise applications**.
  2. In the content pane, click the UDDI registry application.
  3. Under **Detail Properties** click **Security role to user/group mapping**.
  4. Make the changes you require, then click **OK**.
- To change the role mappings by using the wsadmin command, complete the following step:
  1. Use the **edit** command of the AdminApp object and the `MapRolesToUsers` option of this command to map the roles that are defined in the UDDI registry application to the special subjects Everyone or AllAuthenticatedUsers, to users, or to user groups. For example, the following command maps the Version 3 GUI Publish role to Everyone, and the Version 3 SOAP Publish role to user user1 and group group1. *UDDI_Registry_Application* is a variable that represents the name of the UDDI registry application.

     Using Jython:

```
AdminApp.edit(UDDI_Registry_Application, ["-MapRolesToUsers",
[["GUI_Publish_User", "Yes", "No", "", ""],
["V3SOAP_Publish_ User_Role", "No", "No", "user1", "group1"]]] )
```

Using Jacl:

```
$AdminApp edit $UDDI_Registry_Application {-MapRolesToUsers {
{"GUI_Publish_User" Yes No "" ""}
{"V3SOAP_Publish_User_Role" No No "user1" "group1"} }}
```

- Optional: To change the data confidentiality settings, see the topic about configuring SOAP API and GUI services for the UDDI registry.

Configuring UDDI security with WebSphere Application Server security enabled

You can configure the UDDI registry to use the UDDI Version 3 security API or the UDDI Version 1 and Version 2 publish security features. Because WebSphere Application Server security is enabled, WebSphere Application Server data confidentiality management is independent of UDDI security.

Before you begin

WebSphere Application Server security must be enabled.

About this task

You can configure the UDDI registry to use the UDDI security features if this is a requirement. However, for production use, another option is to configure the UDDI Version 3 registry to use WebSphere Application Server security.

The UDDI Version 1 and Version 2 publish security features involve the use of authentication tokens.

To configure the UDDI registry to use the UDDI security features, you use the administrative console.

Procedure

1. In the navigation pane of the administrative console, click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. In the content pane, click the UDDI registry application.
3. Under **Detail Properties**, click **Security role to user/group mapping**.
4. Set the WebSphere Application Server security role mappings to `Everyone` for the following UDDI services:
   - Versions 1 and 2 SOAP publish service (SOAP_Publish _User)
   - Version 3 publish service (V3SOAP_Publish_User_Role)
   - Version 3 custody transfer service (V3SOAP_CustodyTransfer_User_Role)
   - Version 3 security service (V3SOAP_Security_User_Role)

   This change to the role mappings ensures that WebSphere Application Server security cannot override UDDI security.
5. For the UDDI Version 3 Publish and Custody Transfer services, ensure that the UDDI Policy is set to require the use of authentication tokens. The use of authentication tokens is already required for Version 1 and Version 2 Publish services.
   a. Click **UDDI** > **UDDI Nodes** > ***uddi_node_name*** > **[Policy Groups] API policies**.
   b. Select **Authorization for publish** and **Authorization for custody transfer** .
   c. Optional: If you require authentication for UDDI Inquiry services, select **Authorization for inquiry** .
   d. Click **OK**.

Results

After the configuration is complete, WebSphere Application Server authenticates the credentials (user name and password) that are associated with the authentication token. No Security Role authentication restriction is imposed.

For details of WebSphere Application Server data confidentiality management, see the topic about configuring the UDDI registry to use WebSphere Application Server security.

Configuring UDDI Security with WebSphere Application Server security disabled

You can configure the UDDI registry to use the UDDI Version 3 security API or the UDDI Version 1 and Version 2 publish security features. When WebSphere Application Server security is disabled, WebSphere Application Server security roles and data confidentiality constraints do not apply.

Before you begin

WebSphere Application Server security must be disabled.

About this task

You can configure the UDDI registry to use the UDDI security features with WebSphere Application Server security disabled for test purposes. It is not advisable to use this type of configuration for production purposes.

For UDDI Version 1 and Version 2 security, the following features are active and do not require further configuration:

- UDDI Version 1 and Version 2 publish requests require UDDI Version 1 and Version 2 authentication tokens respectively. Publishers that request or use an authentication token must be registered WebSphere Application Server users.
- UDDI Version 1 and Version 2 inquiry requests do not require authentication tokens.

For UDDI Version 3, you must configure the UDDI registry to use the UDDI Version 3 Security API, or to use authentication tokens with the Version 3 Publish and Custody Transfer APIs.

To configure the UDDI registry to use the UDDI Version 3 security features, you use the administrative console.

Procedure

1. Click **UDDI** > **UDDI Nodes** > ***uddi_node_name***.
2. In the **General Properties** section, select **Use authInfo credentials if provided** to specify that the authInfo contents in UDDI API requests are used to validate users.
3. Click **OK**.

Results

Authentication tokens are required for publish requests and custody transfer requests, but not for inquiry requests. Publishers that request or use an authentication token must be registered WebSphere Application Server users.

Access control for UDDI registry interfaces

Access to UDDI registry interfaces is controlled by a combination of Java™ Platform, Enterprise Edition (Java EE) declarative security that uses role mappings, and UDDI properties and policies, such as registering users as UDDI publishers.

Each UDDI registry interface is represented by a security role. The interfaces and their corresponding roles are as follows:

Table 192. Security roles for UDDI registry interfaces. The table lists the different UDDI registry interfaces then the associated security roles.

UDDI registry interface	Security role
Version 3 SOAP inquiry	V3SOAP_Inquiry_User_Role
Version 3 SOAP publish	V3SOAP_Publish_User_Role
Version 3 SOAP custody transfer	V3SOAP_CustodyTransfer_User_Role
Version 3 SOAP security	V3SOAP_Security_User_Role
Version 3 GUI inquiry	GUI_Inquiry_User
Version 3 GUI publish	GUI_Publish_User
Versions 1 and 2 SOAP inquiry	SOAP_Inquiry_User
Versions 1 and 2 SOAP publish	SOAP_Publish_User
EJB inquiry	EJB_Inquiry_Role
EJB publish	EJB_Publish_Role

By default, the inquiry roles are mapped to the *Everyone* special subject and the non inquiry roles are mapped to the *AllAuthenticatedUsers* special subject. With these default settings, after you enable WebSphere Application Server security, you do not need access control to use the UDDI registry inquiry interfaces. However, to use the publish roles and the Version 3 custody transfer role, you must be authenticated using a WebSphere Application Server user ID and password. The Version 3 security role is a special case, because it uses UDDI registry security instead of WebSphere Application Server security, and it must be specially configured.

Roles that are mapped to the *AllAuthenticatedUsers* special subject are further protected, because the user must also be registered as a UDDI publisher to publish data to the UDDI registry. If the user is not registered, an E_unknownUser error is returned in the disposition report. You can register users as UDDI publishers in one of two ways:

- Create a new UDDI publisher by using the administrative console or the Java Management Extensions (JMX) interface.
- Set the **Automatically register UDDI publishers** property of the UDDI node settings so that users are automatically registered as a publisher on their first publish request.

An additional access control, in accordance with the UDDI specification, is that for an entity that is published to the UDDI registry, only the user who originally published that entity can update or delete it.

The UDDI registry also provides some management interfaces that are protected because they require administrative permissions for certain operations.

UDDI registry security and UDDI registry settings

In addition to the configuration of UDDI registry security, other UDDI registry settings can affect the security of the UDDI registry.

Some UDDI property and policy settings can affect the security of a UDDI registry. Other UDDI settings are not specific to security, but can place restrictions on the successful completion of publish requests.

Security settings

UDDI registry interfaces are protected, as detailed in Access control for UDDI registry interfaces.

The UDDI registry supports the use of XML Digital Signatures to sign UDDI entities. See the topic about digital signatures and the UDDI registry.

Some UDDI property and policy settings can affect the security of a UDDI registry.

To review or change the following property settings, click **UDDI** > **UDDI Nodes** > ***uddi_node_name***.

Key space requests require digital signature
: Specifies whether all tModel:keyGenerator requests for key space must be digitally signed. To understand key space, see the topic about UDDI registry Version 3 entity keys.

Use authInfo credentials if provided
: Specifies that the UDDI registry uses the UDDI Version 3 security features. This setting applies only when WebSphere Application Server security is disabled. See Configuring UDDI Security with WebSphere Application Server security disabled.

Authentication token expiry period
: Specifies the length of idle time (in minutes) allowed before an authentication token is no longer valid.

Default user name
: Specifies the name to use for publish operations when WebSphere Application Server security is disabled and no authentication token data is supplied.

To review or change the following policy settings, click **UDDI** > **UDDI Nodes** > ***uddi_node_name*** > **[Policy Groups] API policies**.

Authorization for inquiry
: Specifies whether authorization that uses authentication tokens is required for inquiry API requests.

Authorization for publish
: Specifies whether authorization that uses authentication tokens is required for publish API requests.

Authorization for custody transfer
: Specifies whether authorization that uses authentication tokens is required for custody transfer API requests.

These policy settings apply when UDDI security features are used and WebSphere Application Server security is enabled. If the UDDI service is mapped to the AllAuthenticatedUsers security role, these settings are overridden. See Configuring UDDI Security with WebSphere Application Server security enabled.

Additional settings

The publish-related actions that a registered UDDI publisher can undertake are defined by their entitlements, as described in UDDI registry user entitlements.

Some UDDI property and keying policy settings influence publish behavior. These settings are not specific to security, but you must consider them because they place restrictions on the successful completion of publish requests.

To review or change the following property settings, click **UDDI** > **UDDI Nodes** > ***uddi_node_name***.

Automatically register UDDI publishers
: Specifies that the UDDI registry requires that publisher entitlements are set before allowing any publish requests. This option automatically registers users with default entitlements.

: If you do not select this option, you can register users as UDDI publishers, and specify their entitlements, by using the UDDI publisher settings.

Use tier limits
: Specifies that publication tier limits are enforced.

If you select this option, one or more tiers must be configured by using the UDDI Tier settings. Also, ensure that registered UDDI Publishers are assigned to a tier by using the UDDI publisher settings.

To review or change the following property setting, click **UDDI** > **UDDI Nodes** > ***uddi_node_name*** > **[Policy Groups] Keying policies**.

Registry key generation
Specifies that publishers can request key space and, if successful, publish with publisher-assigned keys.

UDDI registry user entitlements

UDDI registry user entitlements define the set of publish-related actions that registered UDDI users are entitled to perform.

An important entitlement is the number of entities of each type that a UDDI user can publish. You control this entitlement by assigning the user to a UDDI publisher tier. You can define any number of tiers to the UDDI registry, and some predefined tiers are supplied when the UDDI registry is deployed. A UDDI publisher tier specifies the maximum number of each entity (business, service, binding, tModel, and publisher assertion) that a user assigned to that tier can publish. To define UDDI publisher tiers, you use the UDDI Tier settings.

Other entitlements relate to the entitlement of a user to allocate key spaces, where they can specify publisher assigned keys when publishing UDDI entities. A key space is allocated by publishing a keyGenerator tModel, and there are a number of entitlements relating to different kinds of key generator in the UDDI Publisher settings. For more information about key generators and publisher assigned keys, see UDDI registry Version 3 entity keys.

You can set the entitlements for a UDDI user by using the UDDI node collection page in the administrative console, or through Java Management Extensions (JMX) by using the UDDI registry administrative interface.

Securing bus-enabled web services

Service integration technologies provides a range of facilities for secure communication between the service requester and the service integration bus, and between the bus and the target service.

About this task

By default, bus-enabled web services are available when WebSphere Application Server security is enabled and your service integration buses are secured. However this level of security does not impose any restrictions on the users of individual web services. To control how your web services are used by each group of your colleagues or customers, you can further configure your web services to work with password-protected components and servers, with WS-Security and with HTTPS.

Procedure

- “Overriding the default security configuration between bus-enabled web services and a secure bus” on page 1031.
- “Configuring secure transmission of SOAP messages by using WS-Security” on page 1034.
- “Working with password-protected components” on page 1035.
- “Invoking outbound services over HTTPS” on page 1043.

Overriding the default security configuration between bus-enabled web services and a secure bus

To override the default configuration through which the bus-enabled web services component accesses a secure service integration bus, you configure an authentication alias that the service integration resource adapter uses to access the bus.

Before you begin

Note: To use bus-enabled web services when bus security is enabled, your web services clients must provide suitable credentials when making requests. Your clients can provide credentials either by using WS-Security or by using HTTP basic authentication, as described in "Authenticating web services clients using HTTP basic authentication" on page 190. For HTTP basic authentication, application security must also be enabled and, depending on which of these authentication schemes you use, the endpoint listener application must be appropriately configured as described in "Password-protecting inbound services" on page 1036. When you use HTTP basic authentication, you map the **AuthenticatedUsers** role to the special "AllAuthenticatedUsers" group (or to some other suitable authenticated group or user); when you use WS-Security you do not need to map the endpoint listener **AuthenticatedUsers** role unless Application Security is enabled, in which case you map the **AuthenticatedUsers** role to the special "Everyone" group. For more information, see "Assigning users and groups to roles" on page 106.

About this task

The default configuration that the bus-enabled web services component uses to access a secure bus is as follows:

- Access to a bus is configured through the *bus connector role*. By default, every bus connector role includes a group called *server*. Members of this group are authorized to connect to the bus.
- The service integration resource adapter uses a J2C activation specification to communicate with the bus. By default, this activation specification has a Boolean custom property **useServerSubject** that is set to true. This property allows the service integration resource adapter to connect to the bus as a subject (a member) of the server group.

For more information, see "Bus-enabled web services default configuration for accessing a secure bus" on page 1032.

You can override this default configuration by defining an authentication alias that the service integration resource adapter uses to access the bus. Using an authentication alias does not make your configuration more secure. However, you might want to use an alias for consistency of approach if you have other application servers running under WebSphere Application Server Version 6.0.x, or to support your internal business controls for use of IDs and passwords.

Procedure

1. In the navigation pane, click **Service integration -> Buses -> *security_value* -> [Related Items] JAAS - J2C authentication data**.
2. Create a J2C authentication alias.
3. Configure authentication for the resource adapter by completing the following steps:
   a. In the administrative console navigation pane, click **Resources -> Resource Adapters -> J2C activation specifications -> *activation_specification_name***, where ***activation_specification_name*** is **SIBWS_OUTBOUND_MDB**.
   b. In the **Authentication alias** drop-down list, select the authentication alias that you created.
   c. Click **Apply**.
4. Optional: Disable the default authentication configuration.

If you configure an authentication alias you need not also disable the default configuration. If an authentication alias exists, it overrides the default configuration. This means that if you use an authentication alias that is authorized to access the bus then the communication will succeed, and if you use an authentication alias that is not authorized to access the bus then the communication will fail, irrespective of the default settings. However if you subsequently remove the authentication alias from the activation specification, the default configuration will again take control and (if not disabled) will allow the service integration resource adapter to continue to access the bus. For more information, see "Bus-enabled web services default configuration for accessing a secure bus."

To disable the default authentication configuration, complete the following steps:

a. In the administrative console navigation pane, click **Resources -> Resource Adapters -> J2C activation specifications -> *activation_specification_name* -> [Additional Properties] J2C activation specification custom properties**, where ***activation_specification_name*** is `SIBWS_OUTBOUND_MDB`.

b. In the list of custom properties, click **`useServerSubject`**

c. Change the `Value` for the **`useServerSubject`** property from "`true`" to "`false`".

d. Click **OK**.

5. Save your changes to the master configuration.
6. Close the administrative console.

Bus-enabled web services default configuration for accessing a secure bus

By default, the bus-enabled web services component can access a secure service integration bus. This means that your Web services clients, if they provide suitable credentials when making requests, can use bus-enabled web services when bus security is enabled. You can modify or override the default configuration, for example by defining an authentication alias that the service integration resource adapter uses to access the bus.

Note: To use bus-enabled web services when bus security is enabled, your web services clients must provide suitable credentials when making requests. Your clients can provide credentials either by using WS-Security or by using HTTP basic authentication, as described in "Authenticating web services clients using HTTP basic authentication" on page 190. For HTTP basic authentication, application security must also be enabled and, depending on which of these authentication schemes you use, the endpoint listener application must be appropriately configured as described in "Password-protecting inbound services" on page 1036. When you use HTTP basic authentication, you map the **`AuthenticatedUsers`** role to the special "`AllAuthenticatedUsers`" group (or to some other suitable authenticated group or user); when you use WS-Security you do not need to map the endpoint listener **`AuthenticatedUsers`** role unless Application Security is enabled, in which case you map the **`AuthenticatedUsers`** role to the special "`Everyone`" group. For more information, see "Assigning users and groups to roles" on page 106.

The default configuration that the bus-enabled web services component uses to access a secure bus is as follows:

- Access to a bus is configured through the *bus connector role*. By default, every bus connector role includes a group called *server*. Members of this group are authorized to connect to the bus.
- The service integration resource adapter uses a J2C activation specification to communicate with the bus. By default, this activation specification has a Boolean custom property **`useServerSubject`** that is set to `true`. This property allows the service integration resource adapter to connect to the bus as a subject (a member) of the server group.

The server group in the bus connector role

This group controls whether a user is authorized to connect to the bus. The server group can be added or removed by using the administrative console:

Service integration -> Buses -> *security_value* -> [Authorization Policy] Users and groups in the bus connector role

This group can also be set by using the following wsadmin command scripts:

```
addGroupToBusConnectorRole
removeGroupFromBusConnectorRole
```

The useServerSubject property

This boolean property is found in the custom properties panel of the J2C activation specification associated with the inbound, outbound or gateway service:

Resources -> Resource Adapters -> J2C activation specifications -> *activation_specification_name* -> [Additional Properties] J2C activation specification custom properties

This property can also be set by using wsadmin command scripts.

Disabling and overriding the default configuration

To disable the default configuration, set the **`useServerSubject`** property to "`false`" rather than removing the server group, because the service integration resource adapter is not the only system resource that uses the server subject. If you remove the server group from the bus connector role, then no system resources can use the server subject.

You can also override the default configuration by defining an authentication alias that the service integration resource adapter uses to access the bus. Using an authentication alias does not make your configuration more secure. However, you might want to use an alias for consistency of approach if you have other application servers running under WebSphere Application Server Version 6.0.x, or to support your internal business controls for use of IDs and passwords.

If you configure an authentication alias you need not also disable the default configuration. If an authentication alias exists, it overrides the default configuration. However if you subsequently remove the authentication alias from the activation specification, the default configuration will again take control and (if not disabled) will allow the service integration resource adapter to continue to access the bus.

The following table shows whether the service integration resource adapter can connect to the secured bus, depending on the state of the different properties:

Table 193. Summary of expected behavior for accessing a secure service integration bus. The first column of this table shows whether or not the secure service integration bus has a valid authentication alias, indicated by Yes or No as appropriate. The second column indicates whether or not the useServerSubject property is selected, indicated by Yes or No as appropriate. The third column shows whether or not the server group has been added to the bus connector role, indicated by Yes or No as appropriate. The fourth column shows, for each of the combinations of Yes and No settings given in the first three columns, whether or not the resource adapter can connect to the bus, indicated by Yes or No as appropriate.

Valid authentication alias	useServerSubject	Server group on bus connector role	Resource adapter can connect?
Yes	No	No	Yes
No	Yes	Yes	Yes
No	No	Yes	No
No	No	No	No
No	Yes	No	No

Table 193. Summary of expected behavior for accessing a secure service integration bus (continued). The first column of this table shows whether or not the secure service integration bus has a valid authentication alias, indicated by Yes or No as appropriate. The second column indicates whether or not the useServerSubject property is selected, indicated by Yes or No as appropriate. The third column shows whether or not the server group has been added to the bus connector role, indicated by Yes or No as appropriate. The fourth column shows, for each of the combinations of Yes and No settings given in the first three columns, whether or not the resource adapter can connect to the bus, indicated by Yes or No as appropriate.

Valid authentication alias	useServerSubject	Server group on bus connector role	Resource adapter can connect?
Yes	Yes	Yes	Yes (using the authentication alias)

Configuring secure transmission of SOAP messages by using WS-Security

Configure service integration technologies for secure transmission of SOAP messages by using tokens, keys, signatures and encryption in accordance with the Web Services Security (WS-Security) specification.

Before you begin

You can configure the service integration bus for secure transmission of SOAP messages by using tokens, keys, signatures and encryption in accordance with the Web Services Security (WS-Security) 1.0 specification.

Alternatively, you can configure the bus in accordance with the previous WS-Security specification, WS-Security Draft 13 (also known as the Web Services Security Core Specification).

Note: Use of WS-Security Draft 13 was deprecated in WebSphere Application Server Version 6.0. Use of WS-Security Draft 13 is deprecated, and you should only use it to allow continued use of an existing web services client application that has been written to the WS-Security Draft 13 specification.

You can only use WS-Security with web service applications that comply with the *Web Services for Java Platform, Enterprise Edition (Java EE)* or *Java Specification Requirements (JSR) 109* specification. For more information, see "Web Services Security and Java Platform, Enterprise Edition security relationship" on page 325. For information about how to make your web service applications JSR-109 compliant, see Implementing JAX-RPC web services clients or Implementing static JAX-WS web services clients .

About this task

To protect a service integration bus-deployed web service, you can apply the following types of WS-Security resource to the inbound or outbound ports that the service uses:
- WS-Security bindings.
- WS-Security configurations.

The *configurations* resource type specifies the level of security that you require (for example "The body must be signed"), and the *bindings* resource type provides the information that the run-time environment needs to implement the configuration (for example "To sign the body, use this key"),

When you associate a WS-Security resource with a port, you choose from a list of WS-Security resources that you have previously configured as described in the following topics:

Procedure

- Creating a new WS-Security binding.
- Creating a new WS-Security configuration.

What to do next

Note: You can associate any binding with any configuration, so you must ensure that you choose a valid combination. You can also configure various WS-Security binding objects at the cell level, as described in “Default bindings and runtime properties for Web Services Security” on page 224. You can then use these binding objects when configuring bindings for use with your inbound and outbound ports. For example you can use a trust anchor that is defined at cell level when you are defining the signing information for a service integration binding object.

For an overview of how WS-Security is applied to service integration bus-deployed web services, see Service integration technologies and WS-Security. For detailed information about how WS-Security is implemented in WebSphere Application Server, see “Overview of standards and programming models for web services message-level security” on page 298. For more information about the WS-Security standard, see the Web Services Security (WS-Security) 1.0 specification.

Getting WS-Security information from the owning parties

Use this task to learn how to obtain the WS-Security configurations for the client (for an inbound service) and the target web service (for an outbound service).

About this task

You get, from the owning parties, the WS-Security configurations for the client (for an inbound service) and the target web service (for an outbound service). This information is found in the following files on the owners systems:

- Key stores (`.ks`, `.jks` and `.jceks` files).
- Certificate stores (`.cer` files).
- Security settings (the `ibm-webservicesclient-ext.xmi` file for the client, and the `ibm-webservices-ext.xmi` file for the web service).
- Binding information - for example the location of a keystore file on the file system (the `ibm-webservicesclient-bnd.xmi` file for the client, and the `ibm-webservices-bnd.xmi` file for the Web service).

If the client is hosted on WebSphere Application Server, and the Web Service Security settings are created by using IBM web services tooling (for example IBM Rational Application Developer), then the files that contain the security settings and binding information have the exact file names (`*.xmi`) noted previously. For clients and web services from other vendors, these files have different file names.

You must copy the key store and certificate store files to the WebSphere Application Server file system, and enter and configure (as WS-Security bindings and configurations) the security settings that are contained in the `.xmi` files. There are tools available (for example IBM Rational Application Developer) that can parse the `.xmi` files for you.

Working with password-protected components

Configure user ID and password authentication and authorization for inbound services, and for individual operations within a web service. Invoke password-protected outbound services, and access password-protected proxy servers.

About this task

In addition to the security options described in Configuring secure transmission of SOAP messages by using WS-Security, you can also use the broader security features of WebSphere Application Server to work with password-protected components.

Procedure

- "Password-protecting inbound services."
- "Password-protecting a web service operation" on page 1037.
- "Invoking a password-protected outbound service" on page 1041.
- "Accessing a password-protected proxy server" on page 1041.

Password-protecting inbound services

Password-protect a set of inbound services by requiring user authentication for access to the associated HTTP endpoint listener, or (for JMS) to the associated JMS queue destination.

Before you begin

This topic covers the two main areas in which you might want to change the HTTP endpoint listener authentication settings:

- Changing the HTTP endpoint listener security role.
- Mapping the HTTP endpoint listener security role to users or groups.

If you want to change the HTTP endpoint listener security role, do so before you create the HTTP endpoint listener configuration.

For a SOAP over JMS endpoint listener, you can achieve similar results by securing the underlying destination for each JMS queue.

About this task

When WebSphere Application Server administrative security is enabled, clients that access an HTTP endpoint listener can be prompted for a user ID and password, which are authenticated against the registry defined within the security configuration. The HTTP endpoint listeners that are supplied with WebSphere Application Server are configured with a security role named `AuthenticatedUsers`. By default this role is mapped to the special group **Everyone**, so even if security is enabled all users can access any inbound service deployed to the HTTP endpoint listener.

You need not change the default security role. You would only choose to do so if you wanted to use a role name that is more specific, or more meaningful in the context of your organization. To change the security role, you modify the endpoint listener application EAR file before you configure the endpoint listener.

After you configure the endpoint listener application, you can map the security role to specific users or groups so that, when WebSphere Application Server security and service integration bus security are enabled, access to the HTTP endpoint listener is restricted. For more information about why you might want to do this, see Endpoint listeners and inbound ports: Entry points to the service integration bus.

To configure HTTP endpoint listener authentication, complete the following steps:

Procedure

1. Optional: If you want to change the HTTP endpoint listener security role, use an assembly tool to modify the endpoint listener application by completing the following steps:
   a. In the endpoint listener enterprise application, edit the Web application deployment descriptor to add a new role with a name of your choice.

b. Remove the existing role (for example `AuthenticatedUsers`) from the authorized roles within the security constraint, then add the role you created in the previous step.

c. Save the modified endpoint listener application.

2. Create the HTTP endpoint listener configuration.
3. Map the HTTP endpoint listener security role to users or groups by completing the following steps:

Note: The default security role `AuthenticatedUsers` is mapped to the special group **Everyone**. That is, even if WebSphere Application Server security is enabled all users can access any inbound service deployed to the HTTP endpoint listener. To restrict access to just authenticated users, map the role to the special group named **All authenticated**.

a. Enable WebSphere Application Server security.

b. Start the WebSphere Application Server administrative server.

c. Start the administrative console.

d. In the navigation pane, click **Applications -> Application Types -> WebSphere enterprise applications ->** ***application_name***

where *application_name* is the name of the EAR file for this listener. For example `soaphttpchannel1`. In the additional properties for this listener application, an option to map security roles to users and groups is displayed.

e. Assign users and groups to the security role. For example, map the `AuthenticatedUsers` role to the **All authenticated** group.

f. Click **OK**.

g. Save your changes to the master configuration.

Password-protecting a web service operation

Password-protect individual operations (methods) in a Web service by creating an enterprise bean with methods matching the Web service operations, then applying WebSphere Application Server authentication mechanisms to the enterprise bean so that, before a web service operation is invoked, a call is made to the EJB method for authorization.

Before you begin

As well as password-protecting a web service operation as described in this topic, you must also configure the service as either an inbound or outbound service, and select the option to **Enable operation-level security** as described in Modifying an existing inbound service configuration or Modifying an existing outbound service configuration.

In order for an application deployed to the service integration bus to use operation-level security, you must set the application server class-loader policy to “`single`”, as described in Configuring class loaders of a server.

About this task

For operation-level authorization you create an enterprise bean with methods matching the web service operations. These EJB methods perform no operation and are just entities for applying security. You then apply existing WebSphere Application Server authentication mechanisms to the enterprise bean. Before any web service operation is invoked, a call is made to the EJB method. If authorization is granted, the web service is invoked.

Your target web service is protected by wrapping it in an EAR file (*your_webservice*`.ear`), then applying role-based authorization to the EAR file. This process is explained in general terms in Operation-level security: Role-based authorization. The *your_webservice*`.ear` file is then imported into the

sibwsauthbean.ear file and the sibwsauthbean.ear file is modified to set the roles and assign them to methods. The modified sibwsauthbean.ear file is then deployed in WebSphere Application Server, and users are assigned to the previously-defined roles.

The installation version of the sibwsauthbean.ear file is in the *app_server_root*/installableApps directory, where *app_server_root* is the root directory for the installation of WebSphere Application Server.

The sibwsauthbean.ear file contains an EAR file for each web service that you protect. For the first web service that you protect through operation-level authorization, you copy the installation version of the sibwsauthbean.ear file and store your copy outside of the application server file system. For each subsequent web service that you protect, you further modify the same copy of the sibwsauthbean.ear file.

To enable operation-level authorization, you use the **sibwsAuthGen** command, and an assembly tool. You can only use these tools on a Windows system, so you must copy (in binary) to a Windows system all the files you need for this task, then create and modify the EAR files on the Windows system, then copy (in binary) the modified sibwsauthbean.ear file back to your z/OS system.

To password-protect web service operations, complete the following steps for each web service that you want to protect:

Procedure

1. For the first web service that you protect, complete the following steps:
   a. Make your own copy of the *app_server_root*/installableApps/SIBWSauthbean.ear file in a convenient location outside of the application server file system.
   b. On your Windows system, install the assembly tool.
   c. On your Windows system, create a directory with a name of your own choosing (for example /your_dir) and in that directory create a subdirectory called lib.
   d. Use File Transport Protocol (FTP) to copy (in binary) the following files from your target application server under z/OS to your Windows system:

      Copy the following files into your new directory (for example /your_dir):
      - *app_server_root*/util/SIBWSAuthGen.bat

      Copy the following files into your new lib subdirectory (for example /your_dir/lib):
      - *app_server_root*/lib/commons-logging-api.jar
      - *app_server_root*/lib/qname.jar
      - *app_server_root*/lib/wsdl4j.jar
      - *app_server_root*/lib/wsif.jar
      - *app_server_root*/lib/xerces.jar
2. Use File Transport Protocol (FTP) to copy (in binary) your own copy of the sibwsauthbean.ear file from your z/OS system into your directory (for example /your_dir) on your Windows system.
3. To create the *your_webservice*.ear file, complete the following steps:
   a. Open a command prompt on the Windows system.
   b. Go to your directory (for example your_dir).
   c. Enter one of the following commands to set the *WAS_HOME* environment variable to point to your new directory:

      ```
      set WAS_HOME=path_to_new_directory
      ```

      or

      ```
      set WAS_HOME=.
      ```

      where *path_to_new_directory* is the full path to your new directory .
   d. Set the path to point to the Java virtual machine that is supplied with the assembly tool.
   e. Enter the following command to update the class path:

```
set classpath=lib\commons-logging-api.jar;lib\j2ee.jar;lib\qname.jar;lib\wsdl4j.jar;lib\wsif.jar;lib\xerces.jar;
```

f. Enter the following command:

```
sibwsAuthGen location your_webservice_name
```

where

- *location* is the service WSDL location. For an outbound service, you need the target WSDL file that is located at an external web address. For an inbound service, you need the template WSDL file that is located at the endpoint listener endpoint for the service.
- *your_webservice_name* is the name of the service that you are securing, as defined in the location field of the WSDL file. This is case-sensitive.

Note: To get the location details for a given inbound service WSDL file, publish the WSDL file to a compressed file as described in Modifying an existing inbound service configuration, then look up the location within the exported WSDL file. Alternatively, you can retrieve the inbound service WSDL file by using the following standard web services query:

```
http://host_name:port_number/epl_context_root/soaphttpengine/bus_name/inbound_service_name/inbound_port_name?wsdl
```

where *host_name* and *port_number* are the host name and port number for this application server, and *epl_context_root* is the context root of the endpoint listener application as described in Modifying an existing endpoint listener configuration.

Examples of using the **sibwsAuthGen** command:

(outbound service):

```
sibwsAuthGen http://www.somecompany.com/targetservice/wsdl/targetservice.wsdl targetServiceName
```

(inbound service):

```
sibwsAuthGen http://your.server.name:9080/wsgwsoaphttp1/soaphttpengine/yourbus/yourservice/inboundport1?wsdl yourservicename
```

The *your_webservice*.ear file is created in the current directory. There is also a temporary directory *current_directory*/ejb that you can delete.

4. To finish assigning roles and protecting methods, complete the steps given in the topic Using assembly tools to Password-protect a web service operation.
5. To install the modified copy of the sibwsauthbean.ear file, complete the following steps:
   a. Use FTP to copy (in binary) the modified sibwsauthbean.ear file back to the convenient location on your z/OS system that you chose in step 1. Store the modified sibwsauthbean.ear file in this location for subsequent reuse and further modification.
   b. Start the WebSphere Application Server administrative console.
   c. In the navigation pane, click **Applications -> New Application**.
   d. Use the **New Enterprise Application** option to install the modified copy of the sibwsauthbean.ear file. Select the users or groups to assign to the roles when prompted.

Using assembly tools to password-protect a web service operation:

Use this task to learn how to protect a web service operation by using the sibwsauthbean.ear file.

Before you begin

This task assumes that you have already completed the initial steps for Password-protecting a web service operation.

About this task

As is explained in general terms in Operation-level security: Role-based authorization, your target web service is protected by wrapping it in an EAR file and applying role-based authorization to the EAR file. In this task, the EAR file that contains your web service (*your_webservice*.ear) is imported into the

`sibwsauthbean.ear` file (which contains all of the protected web services) and the `sibwsauthbean.ear` file is modified to set the roles and assign them to methods. This modified `sibwsauthbean.ear` file is then deployed in WebSphere Application Server and users are assigned to the previously defined roles.

Use an assembly tool to complete the following steps:

Procedure

1. Start the assembly tool, then open the Java EE perspective.
2. From the File menu select **File > Import > EAR**, then browse to select your copy of the `sibwsauthbean` EAR file. On the Project Explorer tab these projects are created:
   - An enterprise application project called `sibwsauthbean`
   - An EJB project called `Authorization`
3. From the File menu select **File > Import > EAR**, specify a new EAR project name, then browse to select the *your_webservice* EAR file. On the Project Explorer tab these projects are created:
   - An enterprise application project called *your_webservice*.
   - An EJB project called *your_webservice* `ejb`.
4. Select the EJB project *your_webservice* `ejb`, then edit the **EJB Deployment Descriptor**. For every security role that you want to create, repeat the following steps:
   a. On the Assembly tab, add the required security role (for example READER).
   b. Use the Add Method Permission wizard to add one or more method permissions to the security role.
   c. Save your changes.
5. To import the enterprise application *your_webservice* into the `sibwsauthbean` EAR file, complete the following steps:
   a. Select the enterprise application project `sibwsauthbean`, then edit the **EAR Deployment Descriptor**.
   b. On the Module tab, add the *your_webservice* `ejb` enterprise bean from the EJB project *your_webservice* `ejb`.
   c. Save your changes.
6. To ensure that the authorization enterprise bean can reference the newly-imported enterprise bean, complete the following steps to add an EJB reference:
   a. Select the EJB project `Authorization`, then edit the **EJB Deployment Descriptor**.
   b. On the Reference tab, select the `Authorization` reference then click **Add**. The Add Reference wizard is displayed.
   c. Select **EJB Reference > Next**.
   d. Select the **Enterprise beans in the workspace** radio button, then browse to select the *your_webservice* `ejb` enterprise bean.
   e. Save your changes.
7. To assign users to roles, complete the following steps:
   a. Select the enterprise application project `sibwsauthbean`, then edit the **EAR Deployment Descriptor**.
   b. On the Security tab, select **Gather**. For every security role that you want to assign, repeat the following steps:
      1) Select a security role.
      2) Under **WebSphere Bindings**, select the required access level from the following choices:
         - Everyone
         - All authenticated
         - Users/Groups

8. Export the enterprise application project `sibwsauthbean` as an EAR file.

What to do next

You are now ready to install the modified copy of the `sibwsauthbean` EAR file as described in the final step of Password-protecting a web service operation.

Invoking a password-protected outbound service

Invoke a password-protected external web service by configuring and deploying a JAX-RPC handler to set the associated user ID and password.

About this task

Providers of external web services can use HTTP basic authentication to secure their services. When you configure an outbound service to invoke an external web service that requires HTTP basic authentication, you configure and deploy a JAX-RPC handler at the outbound port to provide the required user ID and password in the form of an HTTP Basic Authentication header. To configure and deploy this handler, complete the following steps.

Procedure

1. Create a new JAX-RPC handler class that sets the properties `javax.xml.rpc.security.auth.username` and `javax.xml.rpc.security.auth.password`. For example:

```
public class BasicAuthHandler extends GenericHandler {

    public QName[] getHeaders() {
        return null;
    }

    public boolean handleRequest(MessageContext mc) {

        // Insert basic auth properties
        mc.setProperty("javax.xml.rpc.security.auth.username", "bob");
        mc.setProperty("javax.xml.rpc.security.auth.password", "xy129bge");
        return super.handleRequest(mc);
    }
}
```

2. Create a new JAX-RPC handler configuration for the handler.
3. Create a new JAX-RPC handler list, then select the handler that sets the HTTP basic authentication properties for this service and add it to the handler list.
4. Use the instructions given in Modifying an existing outbound service configuration to navigate to the administrative console page **Service integration -> Buses -> *bus_name* -> [Services] Outbound Services -> *service_name* -> Outbound Ports -> *port_name***, where `service` and `port` indicate the outbound port at which you apply the HTTP basic authentication properties.
5. Set the **JAX-RPC Handler list** property by selecting, from the drop-down list, the handler list that sets the HTTP basic authentication properties for this service.
6. Save your changes to the master configuration.

Accessing a password-protected proxy server

Configure access to an external web service or WSDL file through a password-protected proxy server.

About this task

Service integration technologies requires access to the Internet for invoking outbound services and for retrieval of external WSDL files. Many enterprise installations use a proxy server in support of Internet routing, and many proxy servers require authentication before they grant access to the Internet. This requirement is supported in HTTP messaging by a Proxy-Authorization message header that contains encoded user ID and password credentials.

To enable service integration technologies to invoke an outbound service you configure, for each outbound port, a proxy host, port and J2C authentication alias.

When you create or modify inbound or outbound services, the service integration bus might also have to pass messages through an authenticating proxy server to retrieve WSDL documents. Consequently you must configure the proxy host and port that are used.

Note: Neither the administrative console panels used to create a new web service configuration, nor the **Reload WSDL** option provided in the panels used to modify an existing web service configuration, allow you to enter an authentication alias for WSDL retrieval. If the bus needs to pass messages through an authenticating proxy server to retrieve WSDL documents, then you must use command-line tools to retrieve the WSDL.

Procedure

1. Start the WebSphere Application Server administrative server.
2. Start the administrative console.
3. To enable invocation of an outbound service through a password-protected proxy server, complete the following steps:
   a. In the administrative console navigation pane, click **Service integration -> Buses -> *security_value* -> [Related Items] JAAS - J2C authentication data**.
   b. Create a J2C authentication alias, providing an alias name, and the user ID and password required by the authenticating proxy server.
   c. Click **OK**.
   d. In the administrative console navigation pane, click **Service integration -> Buses -> *bus_name* -> [Services] Outbound Services -> *service_name* -> Outbound Ports -> *port_name***.
   e. Type into the appropriate fields the authenticating proxy host name, port, and the authentication alias you created.
   f. Click **OK**.
4. To enable the service integration bus to pass messages through an authenticating proxy server to retrieve WSDL documents, complete the following steps:
   a. In the administrative console navigation pane, select **Servers -> Server Types -> WebSphere application servers -> *server_name* -> [Server Infrastructure] Java and Process Management -> Process Definition** > **[Additional Properties] Java Virtual Machine -> [Additional Properties] Custom Properties**.
   b. Set the following properties:
      - **http.proxySet** - Set this to `true` to tell the application server that it is required to work with an authenticating proxy.
      - **http.proxyHost** - Set this to the machine name of the authenticating proxy.
      - **http.proxyPort** - Set this to the port through which the authenticating proxy is accessed. For example `8080`.
      - **http.nonProxyHosts** - List the internal machines for which authentication is not required for routing through the proxy. Separate each machine name in the list with a vertical bar (“|”).
      - This list must include the machine on which the bus is installed.

   Note: If the bus needs to pass messages through an authenticating proxy server to retrieve WSDL documents, then you must use command-line tools to retrieve the WSDL.
5. Save your changes to the master configuration.
6. Stop then restart the application server.
7. Close the administrative console.

Invoking outbound services over HTTPS

Use Secure Sockets Layers (SSL) to allow the service integration bus to invoke external web services that include `https://` in their addresses.

About this task

There are two ways to set the bus to use SSL with SOAP over HTTPS messages:

- Configure SSL certificate and key management for a managed endpoint.
- Use a JAX-RPC handler to set the SSL configuration.

By default, each managed endpoint is already configured to use SSL. However you will have to modify the default configuration, for example to add information about the keys and keystores that the external web service uses.

Alternatively, you can use a JAX-RPC handler to set the SSL configuration. You might want to do this because you are upgrading from a previous version of WebSphere Application Server and your configuration is already set to work in this way, or because you have to target an SSL configuration very precisely; for example to secure each service or each invocation.

To configure SSL certificate and key management for a managed endpoint, see Creating a Secure Sockets Layer configuration.

To use a JAX-RPC handler to set the SSL configuration, complete the following steps:

Procedure

1. Start the administrative console.
2. Create a new Secure Sockets Layer repertoire configuration entry.
3. Create a new JAX-RPC handler class that sets the property `ssl.configName` to a value that is the name of the SSL repertoire configuration that you have just created. For example:

```
public class SSLHandler extends GenericHandler {

    public QName[] getHeaders() {
        return null;
    }

    public boolean handleRequest(MessageContext mc) {

        // Insert SSL property
        mc.setProperty("ssl.configName", "myNode/SSLConfig");
        return super.handleRequest(mc);
    }
}
```

4. Create a new JAX-RPC handler configuration for the handler.
5. Create a new JAX-RPC handler list, then select the handler that sets the SSL configuration name property and add it to the handler list.
6. Use the instructions given in Modifying an existing outbound service configuration to navigate to the administrative console page **Service integration -> Buses -> *bus_name* -> [Services] Outbound Services -> *service_name* -> Outbound Ports -> *port_name***, where `service` and `port` indicate the outbound port that is to use SSL.
7. Set the **JAX-RPC Handler list** property by selecting, from the drop-down list, the handler list that sets the SSL configuration name property.
8. Save your changes to the master configuration.

Securing WS-Notification

The WS-Notification security implementation requires that a user identity is flowed in requests for WS-Notification services. This identity is used to authenticate the client application and check that the client is authorized to invoke the requested operation, and to access the underlying service integration bus topic spaces and topic resources.

About this task

WS-Notification uses the same mechanisms as other Web services to provide an authenticated identity. For example WS-Security or HTTP Basic Authentication.

There are three parts to configuring secure access to WS-Notification:
- Securing the communication channel between the application and the server.
- Authorizing the application to invoke the NotificationBroker.
- Authorizing the application to access the resources of the service integration bus.

If messaging security is enabled, and the WS-Security or HTTP Basic Authentication components are not configured to flow a user identity in WS-Notification requests, then all such requests are treated as unauthenticated and can only access messaging resources that are accessible by the WebSphere Application Server “everyone” group.

Procedure

1. Secure the communication between the application and the server:
   a. Provide security for inbound requests and associated responses by configuring the WS-Notification service point.
      - For JAX-WS based Version 7.0 WS-Notification service points, attach security-enabled policy sets to the application associated with the service point. For JAX-RPC based Version 6.1 WS-Notification service points, configure security for the inbound ports associated with the service point.
      - If you are using SOAP over HTTP as the binding for your WS-Notification service point, modify it to use SOAP over HTTPS as described in “Configuring secure access to WS-Notification service points by using SOAP over HTTPS” on page 1045.
      - If you are using SOAP over JMS as the binding (for Version 6.1 WS-Notification services), configure the JMS connection factory used by the client application to use a secure communication protocol to communicate with the JMS provider. Exactly how you do this depends upon the JMS provider. If you are using the service integration bus as the JMS provider, configure the client to use SSL to communicate with the server by setting the **target inbound transport chain** to `InboundSecureMessaging` as described in How JMS applications connect to a messaging engine on a bus and its related tasks.

   b. Provide security for outbound requests (for example notifications from the server to subscribed consumers) by configuring the WS-Notification service.
      - For JAX-WS based Version 7.0 WS-Notification services, the steps involved are similar to those for applying security to JAX-WS web service clients except that any binding or configuration created is applied to the WS-Notification service. For more information, see “Securing JAX-WS web services using message-level security” on page 191.
      - For JAX-RPC based Version 6.1 WS-Notification services, the steps involved are similar to those for applying security to service integration bus-enabled web services outbound ports except that any binding or configuration created is applied to the WS-Notification service. For more information, see “Securing bus-enabled web services” on page 1030 and its sub-topics, notably “Invoking outbound services over HTTPS” on page 1043.

   c. You can also use WS-Security to sign or encrypt SOAP messages.

- For JAX-WS based Version 7.0 WS-Notification services, see “Signing and encrypting message parts using policy sets” on page 145.
- For JAX-RPC based Version 6.1 WS-Notification services, see “Configuring secure transmission of SOAP messages by using WS-Security” on page 1034.

2. Authorize the application to invoke the NotificationBroker:
   a. Configure the client application to provide an appropriate identity.

      To authorize a web service application to communicate with the server, the application must identify itself as running as a particular authenticated identity. The mechanism for doing this depends upon the type of web service binding you are using:
      - If you are using SOAP over HTTP web service bindings, use either HTTP Basic Authentication or WS-Security to provide the authenticated identity.
      - If you are using SOAP over JMS web service bindings (for Version 6.1 WS-Notification services), use WS-Security to provide an authenticated identity.

   b. Configure the server to authorize the client application identity to carry out the required operations.
      - For JAX-WS based Version 7.0 WS-Notification services, you can use Web services policy sets such as the “Username WS-I RSP default” or “Username WSSecurity default” policy sets to apply authorization to the Web services that are deployed in the enterprise application associated with a service point. See also the IBMdeveloperWorks article Configuring JAX-WS applications with WS-Security for WS-Notification.
      - For JAX-RPC based Version 6.1 WS-Notification services, you can apply authorization to the whole of an inbound service (for example the NotificationBroker endpoint of a WS-Notification service point) as described in “Password-protecting inbound services” on page 1036, or configure authorization constraints independently for each Web service operation as described in “Password-protecting a web service operation” on page 1037.

3. Authorize the application to access the resources of the service integration bus.

   Service integration bus security uses role-based authorization. When a user is assigned to a role, the user is granted all of the permissions that the role contains. By administering authorization permissions, you can control user access to a bus and to its resources when messaging security is enabled.
   a. Authorize the application identity to be able to connect to the service integration bus, as described in “Administering the bus connector role” on page 57.
   b. When the application can connect to the bus, grant the application access to the appropriate destinations on the bus.

      You can determine which service integration bus topic spaces are required, by checking which WS-Notification topic namespaces are used by the application then looking at the appropriate WS-Notification permanent topic namespace to find the service integration bus topic space to which it maps. You can then grant authorization (for example the Sender or Receiver roles) for the authenticated identity to access that topic space as described in “Administering destination roles” on page 62.
   c. After the client application has been authorized to access the appropriate topic space destination, you might also need to authorize the client application to access the individual topics within the topic space destination as described in “Administering topic roles” on page 77.

   For general information about configuring access to the service integration bus, see Securing service integration.

Configuring secure access to WS-Notification service points by using SOAP over HTTPS

Modify the configuration of existing WS-Notification service points to require use of SOAP over HTTPS instead of SOAP over HTTP as the binding for inbound requests from notification providers.

Before you begin

This task assumes that you have an existing WS-Notification service and service points, and that you are using the SOAP over HTTP binding for inbound requests.

About this task

By default the SOAP over HTTP endpoints used by service points accept both HTTP and HTTPS requests. HTTPS can be used by changing the endpoint URL prefix to `https://` on each WS-Notification service point for the service, and modifying the port to the HTTPS port used by the server (default of 9443).

For Version 7.0 WS-Notification services, enterprise applications are used to expose the web services associated with the WS-Notification service. For Version 6.1 WS-Notification services, service integration endpoint listeners are used.

Procedure

1. Start the administrative console.
2. Navigate to **Service integration -> WS-Notification -> Services ->** ***service_name*** **-> [Additional Properties] WS-Notification service points** or **Service integration -> Buses ->** ***bus_name*** **-> [Services] WS-Notification services ->** ***service_name*** **-> [Additional Properties] WS-Notification service points**, then identify the WS-Notification service points for the WS-Notification service you want to secure.
3. Configure HTTPS access individually on each of the WS-Notification service points by repeating the following sub-steps:

   For Version 7.0 WS-Notification services:

   a. In the content pane, click the name of a Version 7.0 WS-Notification service point in the list.

   b. Navigate to the associated enterprise application by clicking **[Additional Properties] Service point application**. The enterprise application settings panel is displayed.

      Note: You can also reach this panel by clicking **Applications -> Application Types -> WebSphere enterprise applications ->** ***application_name***.

   c. In the enterprise application settings panel, click **[Web Services Properties] Provide HTTP endpoint URL information**.

   d. Specify the endpoint URL prefix (that is the protocol (HTTPS), host name, and port number) to use in the endpoint URL. You can select the default HTTPS prefix (`https://your_host_name:9443`) from a predefined list, or you can create and use your own custom HTTPS prefix. For more information, see Configuring endpoint URL information for HTTP bindings.

   For Version 6.1 WS-Notification services:

   a. Create a new endpoint listener with an `https` URL as the URL root.

   b. Modify this WS-Notification service point to associate the inbound port for the new endpoint listener with this service point. The `https` URL appears in the published WSDL file.

   c. Prevent the new endpoint listener from accepting HTTP connections by modifying the virtual host settings. For more information, see Virtual hosts and Creating a Secure Sockets Layer configuration.

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

APACHE INFORMATION. This information may include all or portions of information which IBM obtained under the terms and conditions of the Apache License Version 2.0, January 2004. The information may also consist of voluntary contributions made by many individuals to the Apache Software Foundation. For more information on the Apache Software Foundation, please see http://www.apache.org. You may obtain a copy of the Apache License at http://www.apache.org/licenses/LICENSE-2.0.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Intellectual Property & Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Requests

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

© Copyright IBM Corp. 2012

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. For a current list of IBM trademarks, visit the IBM Copyright and trademark information Web site (www.ibm.com/legal/copytrade.shtml).

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

© Copyright IBM Corp. 2012

Index

A

algorithm mapping 926
algorithm URIs 924
APIs
  JavaMail
    security 35
    security permissions 35
  propagation
    SAML tokens 405
  SAML
    overview 306
  SAML token library 391
  token creation
    SAML sender-vouches token 403
  web services encryption
    WSSEncryptionPart 423, 466
  web services security 589
    consumer signing verification 562
    encryption configuration 411, 454
    generator security tokens 448, 491
    generator signing information configuration 425, 468
    generator token attachment 441, 484
    message authenticity 441, 484
    message confidentiality 550
    message protection 549
    message-level security 408
    programming model 379
    signature verification 566
    signing information configuration 426, 469
    signing information verification 563
    SOAP messages 411, 453
    WS-Trust client 386
    WSS 591
    WSSDecryption 552
    WSSDecryptPart 558
    WSSEncryption 414, 456
    WSSSignature 430, 438, 473, 481
    WSSSignPart 440, 482
    WSSSignPart API 434, 476
    WSSVerfyPart 569
    WSSVerification 574
    WSSVerifyPart 576
audits
  web services security
    runtime environment 695
authentication
  client configuration
    for signatures 657
  HTTP basic usage
    web services clients 190
  Kerberos
    message protection 781
    tokens 284
    web services clustering 290
    web services configuration models 289
    web services message protection 286
authentication *(continued)*
  Kerberos *(continued)*
    web services usage overview 286
  Kerberos tokens
    web services security 778
  LTPA tokens
    web services security 712
  server configuration
    basic authentication handling 647
    identity assertion handling 652
    LTPA tokens 668
    signature validation 661
  Username tokens
    web services security 712
  web services security 193

B

basic authentication
  client configuration 642
    authentication information collection 644
  server configuration
    validation 648
  web services security
    Version 5.x applications 641
bindings
  client bindings configuration 801
  client configurations 801
  client security configuration
    assembly tool 624
  configuration 819
    clients 1002
    servers 1002
  migration
    clients 370
    servers 368
  policy set management 138
  provider configurations 801
  server security configuration
    administrative console 1004
    assembly tool 627
  STS 791

C

cache
  distributed nonce 297
  web services client tokens
    SAML 407
callers
  configuration
    general and default bindings 150
  order change
    token or message parts 151
certificate revocation list 969

certificate stores
  configuration
    generator bindings 961
certificates 155
  collection certificate stores 349
  revocation list 273
collection certificate stores 272, 963
  client configuration
    administrative console 995
    assembly tool 605
  configuration
    consumer bindings 971
  server configuration
    assembly tool 605
collection certificates
  configuration
    cells 973
    servers 973
consumer signing
  information verification
    message protection 562
cryptographic keys
  enabling
    hardware devices 982
  hardware devices
    web services security 980
  web services security
    hardware devices 982
custom properties
  inbound 154
  outbound 154
  web services security 675, 677

D

data sources
  security 5
decryption
  methods 560
  methods for consumer bindings 555
  web services security
    WSSDecryptPart 558
default bindings 346
default collection certificate stores
  configuration
    administrative console 997
default configurations
  ws-security.xml file 347
deployment descriptors
  web services
    clients 371
    servers 375
digital signature
  XML signatures 342
directory
  installation
    conventions 129
distributed cache
  synchronous update 730
  token recovery 730
distributed cache *(continued)*
  web services security
    administrative console 774

E

encryption
  adding parts
    SOAP messages 423, 466
  methods 421, 464
  methods for generator bindings 417, 460
  SOAP headers 592
  XML 273
encryption information 890
enterprise applications 37
Enterprise JavaBeans (EJB)
  security 27

F

files
  rrdSecurity.props file 978
  SAMLIssuerConfig.properties file 806
  ws-security.xml file 347

H

HTTP basic authentication 495, 674
HTTP sessions
  security 121
HTTP SSL configuration 672

I

interfaces
  web services security
    default service providers 263
    provider programming 599

J

JAAS
  Kerberos login modules
    system updates 788
JAAS Subject
  token retrieval 597, 598
JAS-WS
  client applications
    security tokens 595
JAX-RPC
  administration
    message-level security 821
  application migration 363
  consumer binding
    applications 839
  default sample configurations 257
  development
    message-level security 597
  encryption methods
    message confidentiality protection 888, 912

JAX-RPC *(continued)*
generator signing configuration
message integrity 822
key information
consumer bindings 911
generator bindings 909
key information configuration
applications 844
key locators
cell 952
configuration 943
consumer bindings 950
server 952
message authenticity protection
token consumers 877
message integrity protection
consumer signing 839
message-level security 192
cells 903
for applications 822
servers 903
migration
client-side extension 367
server-side extension 364
secure messages
request consumers 822
request generators 821
response consumers 822
response generators 821
signing information
consumer bindings 906
generator bindings 823, 903
token consumers
message authenticity protection 929
token generators
message authenticity protection 916
web services security
configuration 939
JAX-RPC web services
HTTP basic authentication
administrative console 673
assembly tool 602
programmatic configuration 379
JAX-RS
secure applications 137
web container 123
secure clients
using SSL 134
secure downstream resources 133
secure resources
using annotations 129
web applications security 123
JAX-WAS web services
message-level security 379
JAX-WS
administration
message-level security 694
application development
token retrieval 596
default bindings
web services security 820
JAX-WS *(continued)*
Kerberos token
policy sets 779
message-level security 191
security binding migration 362
JAX-WS applications
sample bindings 244

K

key
locators 237, 945
key information 847
consumer bindings
applications 855
key information references 842
keys 236, 350, 871
locator configuration 606
administrative console 999
locators 349
cells 1000
servers 1000

L

login mappings 351, 983
LTPA 670
server configuration
authentication information validation 669
single sign-on
enablement 979
token authentication
client configuration 666
method information collection 667

M

message-level security
development
JAX-WAS web services 379
web services applications 298
messages
authenticity protection
for applications 858
encryption
policy sets usage 145
SOAP 360
methods
authentication
identity assertion 646, 651
web services security 337
BasicAuth 643
decryption 560
client configuration 639, 640
encryption 421, 464
message confidentiality protection 900, 912, 914
identity assertion 650
response signature verification
clients 572
server configuration
response encryption 637

methods *(continued)*
signature authentication 658
XML encryption 354
web services security 630

N

nonce 295
cache distribution 942
configuration 940
applications 991
cells 992
servers 990
web services security tokens 989
randomly generated tokens 341
NotificationBroker endpoints 1044

O

OASIS applications
supported functions 201
optimized local adapters
security 21

P

part references 832
platform configurations
bindings 232
overview 232
policy sets
Kerberos
sample bindings 788
managing
secure policy set bindings 138
STS 791
system 743
system configuration
administrative console 746
system definition
administrative console 747
trust service
request protection 713
web services security 698
policy sets 1044
portlet URL security 39
properties
default bindings 928
rrdSecurity.props file 978
SAMLIssuerConfig.properties file 806
security runtime environment 928
web services security 948

R

request decryption
server configuration
decryption methods 634
message parts 633
request digital signature
server configuration 615
request digital signatures
server configuration 613
request encryption
client configuration
message parts 631
method information collection 632
request receiver bindings 1008
request receivers 358
request sender bindings 1007
request senders 357
request signing
client configuration 609, 611
client methods 436, 479
response digital signatures
client configuration 620, 622
response encryption
server configuration 636
response receiver bindings 1010
response receivers 359
response sender bindings 1010
response senders 358
response signing
server configuration 617
digital signature methods 619

S

SAML
assertions
token profile standard 301
concepts 301
default policy sets 304
limitations 315
message-level tokens 789
sample bindings 304
secure messages 789
token propagation 1014
tokens
attribute creation 1018
wsadmin commands 817
usage scenarios 308
user attributes 1019
web services
client token cache 407
web services security 193
SAML applications
deployment 1014
development 385
secure messages
message parts
administrative console 143
SSL transport policy 138
security
JAX-RS web applications 123
SIP 93, 94
security models 230
security profiles
compliance tips 295

servers
identity assertion validation 654
service endpoints
attachments
administrative console 755
signature authentication
client configuration
authentication information collection 659
server configuration 660
Version 5.x web services 656
signature confirmation 594
signatures
XML digital signatures 271
signing information 826
SIP
development
trust association interceptor 97
security 93, 94
trust association interceptor 97
SOAP
secure conversation 714
SPIs 384
system policy sets 748

T

targets
trust service
administrative console 769
trust service endpoint configuration
administrative console 768
time stamps 292
token consumers 934
token generators 921
token providers
security context
administrative console 760, 761
tokens 361
binary 341
web services security 282
consumer configuration
secure conversation 731
consumers
configuration 584
message protection 578
derived key 722
generator configuration
secure conversation 731
generic security token login modules 315
token consumers 320
token generators 317
Kerberos
realm environments 291
LTPA 277
nonce 295
overview 339
pluggable configuration 663
administrative console 1012
Version 5.x web services 663
pluggable support 361
propagation 328
tokens *(continued)*
SAML 292
SAML bearer
API 396
SAML bearer tokens
external STS 520
self-issued 495
SAML holder-of-keys
API 399
asymmetric keys 516
external STS 539, 540
symmetric keys 514
SAML sender-vouches
external STS 526, 533
message-level protection 502
SSL transport protection 509
security context 736
disabling submission draft levels 763
Username 279, 340
web services security 277, 541
propagation 297
XML 281
XML tokens 342
transforms 835
trust
ID evaluators 351
trust anchors 238, 348, 956
configuration
assembly tool 603
cells 960
consumer bindings 959
generator bindings 954
servers 960
trust service 733
attachment configuration
administrative console 752
attachments 756
targets 770
token custom properties 763
token providers 766
trusted IDs
configuration
cells 975
servers 975
evaluators 239, 976

U

UDDI publishers 1027
UDDI registry
configuring 1024
interfaces 1027, 1028
security 1024, 1026, 1028
UDDI registry settings 1028
UDDI registry user entitlements 1030

W

web applications
security 101

web services
  applications
    transport communications 189
  secure conversation
    standard 732
  secure conversations 717, 718
    client cache 720
  security 188, 194
  settings
    actor roles 188
    authentication generator 173
    callback handler 176
    caller 183, 186
    certificate stores 159
    consumer token 173
    custom keystores 181
    key information 156
    message expiration 187
    protection tokens 169
    signed or encrypted message parts 147
    SSL transport security 140
    SSL transport security policy 139
    trust anchors 161
  trust service
    configurations 720
  trust standard 745
web services security
  administration 671
  application development 379
  architecture 228
  authentication 226
  binary tokens 282
  binding configuration 819
  client security bindings 874
  collection certificate stores 998
  concepts 194
    Version 5.x applications 321
  confidentiality 226
  configuration 587
    during application assembly 601
    JAX-RPC 939
  configuration considerations 222
  considerations 293
  constraints 329
  cryptographic device
    hardware 242
  cryptographic device enablement 980
  default bindings 224
  default configurations 243
  enhancements 197
  hardware cryptographic devices
    configuration 980
  HTTP outbound transport communications
    administrative console 671
    assembly tool 601
    Java properties 673
  identity assertion
    Version 5.x web services 649
  Java EE security 325
  keys 236
  message integrity 226
web services security *(continued)*
  migration 362
  mixed cluster environment 723
  new features 194
  properties 948
  runtime configuration updates 773
  runtime properties 224
  SAML 193
  SAML bearer tokens
    self-issued 495
  sample configuration 330
  secure conversations
    distributed cache 724
    security context tokens 726
    session affinity 724
  security context tokens
    reliable messaging 728
  security model 326
  server security bindings 875
  server-side collection certificates
    administrative console 996
  settings
    algorithm mapping configurations 927
    algorithm URI configuration 925
    callback handler 866
    certificate revocation list configurations 970
    collection certificate store configurations 965
    encryption information configuration 891, 897
    JAAS configuration 885
    key configuration 872
    key information configuration 848
    key information reference configuration 843
    key locator configurations 946
    login bindings configuration 1005
    part reference configuration 834
    request consumer (receiver) binding configuration 882
    request generator (sender) binding configuration 863
    response consumer (receiver) binding configuration 884
    response generator (sender) binding configuration 865
    security cache 775
    signing information configuration 827
    signing parameter configurations 343
    system policy sets 750
    token consumers 935
    token generator configurations 921
    transforms 836
    trust anchors 957
    trust service attachments 759
    trust service targets 772
    trust service token providers 764
    trusted ID evaluator configurations 977
    web services runtime updates 774
    web services security property configurations 949
    X.509 certificate configurations 968
  specification 216, 321
  support 322

web services security *(continued)*
transport communications
web services applications 189
tuning 1021
Version 5.x applications 1023
Version 5.x applications 663
identity assertion authentication 649
signature authentication 656
XML digital signatures 271
WS-Notification
service points 1046
WS-Security
authentication 163
application-specific bindings 166
general bindings 164
policy sets
bindings 153
protection 163
application-specific bindings 166
general bindings 164
WS-Security standard 1044
wsadmin commands
self-issued SAML tokens 817
WSDL
binding assertions
transformation 141
policy assertions
transformation 141

X

x.509 certificates 966
XML basic authentication
configuration
Version 5.x web services 641
XML digital signatures
configuration
Version 5.x web services 602, 983
web services security 607
XML encryption
configuration
Version 5.x web services 629, 1005