HOW TO make certificates using your own Certification Authority and use them in WebSphere MQ

Layout:
A. Problem description
B. Solution
C. Further discussion

A. Problem description

You want to

- create your own Certification Authority (CA).
- use certificates, signed by this CA, for authentication of WebSphere MQ queue managers and clients.

You are running

- WebSphere MQ version 6.0
- Windows XP or similar

B. Solution

Following the steps in this document, you will use the command-line utilities supplied with WebSphere MQ to:

1. Create your own CA.
2. Issue a signed certificate for a queue manager
3. Issue a signed certificate for a C or C++ application
4. Issue a signed certificate for a Java application
5. Run a C application, to prove that it works
6. Run a Java application, to prove that it works

Steps 1.x: Creating a certification authority (CA)

These steps describe how to create a CA using the command-line utilities supplied with WebSphere MQ.

Step 1.1: Create a key repository for the CA

Create a directory, and in that directory create a *key repository* file, by entering the text shown below:

```
C:\> mkdir \myCAdir
C:\> cd \myCAdir
C:\myCAdir> runmqckm -keydb -create -db myCA.kdb -type cms
```

You will be prompted to create a password. Wait for the prompt, then type the password you want to use for the CA's key repository.

Notes:

- This GSKit key repository is needed to create and store the self-signed CA certificate, along with an associated private key.
- Keep the key repository password safe, as it protects the private key for your self-signed CA certificate. The password is required for all commands in this section.
- Three files are created by this command, in the current working directory. These are called `myCA.kdb`, `myCA.crl` and `myCA.rdb`.
- All commands that specify `-db myCA.kdb` later in this document will be issued from this directory.

Step 1.2: Create your self-signed CA certificate

Create a *self-signed CA certificate*, which will be used to identify your CA:

```
C:\myCAdir> runmqckm -cert -create -db myCA.kdb -type cms -label
"myCAcertificate" -dn
"CN=myCAName,O=myOrganisation,OU=myDepartment,L=myLocation,C=UK"
-expire 1000 -size 1024
```

Notes:

- The CA certificate is created *inside the key repository*. It is not yet extracted into a file – we will do this in Step 1.3.

- The `-expiry` parameter specifies the number of days, from the current date, until the CA certificate expires. This also limits the maximum expiry date of any certificate signed using the CA certificate.
- The distinguished name (DN), as specified in the `-dn` parameter, should be tailored to your organisation, and usage of the CA.
- Information about the components that make up the distinguished name is provided in the WebSphere MQ Security manual.

Step 1.3: Extract the CA certificate

Extract the CA certficate into a file called `myCAcertfile.cer`, which we will later transfer to the key repositories of the queue manager and client application:

```
C:\myCAdir> runmqckm -cert -extract -db myCA.kdb -type cms
-label "myCAcertificate" -target myCAcertfile.cer -format ascii
```

Notes:

- In this Step, we have extracted only the public certificate of the CA. The private key associated with this certificate is held securely in the key repository.
- In later Steps, we show how the CA certificate is added into the key repositories of other entities in the infrastructure.
- The CA certificate is added into the key repositories of other entities, in order to make those entities trust the CA.

Steps 2.x: Issuing a certificate to a queue manager

Each queue manager in your infrastructure should have its own certificate, with an appropriate distinguished name (DN). The DN should be unique within the WebSphere MQ network.

Step 2.1: Set the pathname for the queue manager's key repository file

If you have not already done so previously, create and start the queue manager.

```
C:\myCAdir> crtmqm myqmgr
C:\myCAdir> strmqm myqmgr
```

In this example, the following filename is used for the *key repository*: `C:\REPOS\myqmgr.kdb` Set this in the queue manager:

```
C:\myCAdir> runmqsc myqmgr
ALTER QMGR SSLKEYR('C:\REPOS\myqmgr')
```

Notes:

- Each queue manager requires a *key repository* in order to store its certificate, and the associated private key.
- The default **location** in which a queue manager looks for its *key repository* is (assuming a default installation directory) `C:\Program Files\IBM\WebSphere MQ\Qmgrs\myqmgr\ssl`
- The default **filename** a queue manager looks for when locating its *key repository* is `key.kdb`.
- The queue manager's SSLKEYR attribute contains the full path to the key repository, followed by the name of the key repository, up to, but not including, the `.kdb`.

Step 2.2: Create the queue manager's key repository

Change directory to the location where the key repository file is to be created (`C:\REPOS`)

```
C:\myCAdir> mkdir \REPOS
C:\myCAdir> cd \REPOS
```

and issue the following command to create a key database for the queue manager:

```
C:\REPOS> runmqckm -keydb -create -db myqmgr.kdb -type cms
-stash
```

You will be prompted to create a password. Wait for the prompt, then type the password you want to use for the queue manager's key repository.

Notes:

- It is important that this password is kept safe, as it protects the private key of the signed certificate. This password is required for later commands in this section.
- The `-stash` option is important, as it causes a 'stash file' to be created. This file is called `myqmgr.sth` . It allows the queue manager to open the key repository without requesting a password from the user.

Deployment note:

- Access to the `C:\REPOS` directory should be granted only to the 'mqm' group, to protect the integrity of your security policy. **IMPORTANT:** Set the permissions appropriately for your environment. If necessary, to help you implement an appropriate security policy, consult with the team that decides security policy in your organisation.

Step 2.3: Create a certificate request

Generate a *certificate request file* for the queue manager, along with a private key:

```
C:\REPOS> runmqckm -certreq -create -db myqmgr.kdb -type cms -dn
"CN=QMNAME,O=IBM,OU=WMQ,L=Hursley,C=UK" -label
"ibmwebspheremqmyqmgr" -file myqmgr.req
```

Notes:

- This command will prompt you for a password. Supply the password for the queue manager key repository. This was chosen and first used in Step 2.2.
- The distinguished name (DN), as specified in the -dn parameter, should be tailored to your organisation and uniquely identify the

queue manager within the infrastructure (see Step 1.2 for more information about DN strings).

- The label, as specified with `-label` parameter, must be of the form `ibmwebspheremq`**`myqmgr`** , all in lower case. This is important, as otherwise the queue manager will fail to find the certificate.

Step 2.4: Transfer the request

Transfer the certificate request file, `myqmgr.req` , to the directory where the CA files are located. Then change to this directory:

```
C:\REPOS> copy myqmgr.req \myCAdir
C:\REPOS> cd \myCAdir
```

Step 2.5: Sign the queue manager's certificate

Run the following command:

```
C:\myCAdir> runmqckm -cert -sign -db myCA.kdb -label
"myCAcertificate" -expire 365 -format ascii -file myqmgr.req
-target myqmgr.cer
```

Notes:

- This command will prompt you for a password. Supply the CA key repository's password. This was chosen and first used in Step 1.1.
- The `-expiry` parameter specifies the number of days, from the current date, until the signed certificate expires. The date of expiry cannot be later than the date of expiry of the CA certificate.

Step 2.6: Transfer the CA-signed certificate

Transfer the signed certificate, `myqmgr.cer` , and the public certificate of the CA, `myCAcertfile.cer` , back to `C:\REPOS`

```
C:\myCAdir> copy myqmgr.cer \REPOS
C:\myCAdir> copy myCAcertfile.cer \REPOS
C:\myCAdir> cd \REPOS
```

Step 2.7: "Add" the CA certificate

Add the public certificate of the CA to the *key repository* of the queue manager:

```
C:\REPOS> runmqckm -cert -add -db myqmgr.kdb -type cms -file
myCAcertfile.cer -label "theCAcert"
```

Notes:

- This command will prompt you for a password. Supply the queue manager key repository's password. This was chosen and first used in Step 2.2.
- In order to use a certificate signed by the CA, and to trust other certificates signed by that CA, the key repository of the queue manager must contain the public certificate of the CA.
- The label we used for the CA certificate, `theCAcert` , was different from the label it had in the CA's own key repository. It is not necessary to do this, but this example shows that it is possible.
- Take note of the terminology used here. We have just **added** the public certificate of the CA to the *key repository* of the queue manager. This is a different action from **receiving** a certificate.

Step 2.8: "Receive" the signed certificate

Receive the certificate (now signed by the CA) into the queue manager's key repository:

```
C:\REPOS> runmqckm -cert -receive -db myqmgr.kdb -type cms -file
myqmgr.cer
```

Notes:

- This command will prompt you for a password. Supply the queue manager key repository's password. This was chosen and first used in Step 2.2.
- The signed certificate must be **received** back into the same *key repository* that was used to create the certificate request.
- This key repository is the only place where the private key exists. The private key is uniquely partnered with the signed certificate.

Steps 3.x: Issuing a certificate to a C or C++ client application

Each application in your infrastructure should have its own certificate, with an appropriate distinguished name (DN). The DN should be unique within the WebSphere MQ network.

Step 3.1: Create a key repository for the application

Change directory to the location where the key repository is to be created (`C:\MYAPP`)

```
C:\myCAdir> mkdir \MYAPP
C:\myCAdir> cd \MYAPP
```

and issue the following command to create a key database for the application

```
C:\MYAPP> runmqckm -keydb -create -db myapp.kdb -type cms -stash
```

You will be prompted to create a password. Wait for the prompt, then type the password you want to use for the application's key repository.

Notes:

- It is important that this password is kept safe, as it protects the private key of the signed certificate. This password is required for later commands in this section.
- The `-stash` option is important, as it causes a 'stash file' to be created. This file is called `myapp.sth` . It allows the MQ library code in the application to open the key repository without requesting a password from the user.

Deployment note:

- Only the user under which the application runs should have access to the files in `C:\MYAPP` , to protect the integrity of your security policy. **IMPORTANT:** Set the permissions appropriately for your environment. If necessary, to help you implement an appropriate security policy, consult with the team in your organisation that decides security policy.

Step 3.2: Create a certificate request

Enter this command:

```
C:\MYAPP> runmqckm -certreq -create -db myapp.kdb -type cms -dn
"CN=myAppName,O=IBM,OU=myDepartment,L=Hursley,C=UK" -label
"ibmwebspheremqmyuserid" -file myapp.req
```

Notes:

- This command will prompt you for a password. Supply the application key repository's password. This was chosen and first used in Step 3.1.
- The label specified must be of the form `ibmwebspheremq`**`myuserid`** , in lower case, where *myuserid* is the user identifier under which the application runs on the client machine.

Step 3.3: Transfer the request

Transfer the certificate request file, `myapp.req` , to the directory where the CA files are located. Then change to this directory:

```
C:\MYAPP> copy myapp.req \myCAdir
C:\MYAPP> cd \myCAdir
```

Step 3.4: Sign the application's certificate

Run the following command:

```
C:\myCAdir> runmqckm -cert -sign -db myCA.kdb -label
"myCAcertificate" -expire 365 -format ascii -file myapp.req
-target myapp.cer
```

Notes:

- This command will prompt you for a password. Supply the CA key repository's password. This was chosen and first used in Step 1.1.
- The `-expiry` parameter specifies the number of days, from the current date, until the signed certificate expires. The date of expiry cannot be later than the date of expiry of the CA certificate.

Step 3.5: Transfer the CA-signed certificate

Transfer the signed certificate, `myapp.cer` , and the public certificate of the CA, `myCAcertfile.cer` , back to `C:\MYAPP`

```
C:\myCAdir> copy myapp.cer \MYAPP
C:\myCAdir> copy myCAcertfile.cer \MYAPP
C:\myCAdir> cd \MYAPP
```

Step 3.6: "Add" the CA certificate

Add the CA certificate to the *key repository* of the application:

```
C:\MYAPP> runmqckm -cert -add -db myapp.kdb -type cms -file
myCAcertfile.cer -label "theCAcert"
```

Notes:

- This command will prompt you for a password. Supply the application key repository's password. This was chosen and first used in Step 3.1.
- In order to use a certificate signed by the CA, and to trust other certificates signed by that CA, the key repository of the application must contain the CA certificate.
- The label we used for the CA certificate, `theCAcert` , was different from the label it had in the CA's own key repository. It is not necessary to do this, but it has been done here to show that it is possible.
- Take note of the terminology used here. We have just **added** the public certificate of the CA to the *key repository* of the application. This is a different action from **receiving** a certificate.

Step 3.7: "Receive" the signed certificate

Receive the certificate (now signed by the CA) into the application's key repository:

```
C:\MYAPP> runmqckm -cert -receive -db myapp.kdb -type cms -file
myapp.cer
```

Notes:

- This command will prompt you for a password. Supply the application key repository's password. This was chosen and first used in Step 3.1.
- The signed certificate must be **received** back into the same key repository that was used to create the certificate request.
- This key repository is the only place where the private key exists. The private key is uniquely partnered with the signed certificate.

Steps 4.x: Issuing a certificate to a Java client application

Each application in your infrastructure should have its own certificate, with an appropriate distinguished name (DN). The DN should be unique within the WebSphere MQ network.

This sequence of steps enables you to create signed certificates usable from Java Secure Socket Extension (JSSE), allowing you to secure each Java client.

The WebSphere MQ Explorer, which connects to queue managers using an MQI channel from a Java Virtual Machine, can also be secured using this method.

Step 4.1: Creating the Java keystore

Change directory to the location where the key repository is to be created (`C:\MYAPPJ`)

```
C:\myCAdir> mkdir \MYAPPJ
C:\myCAdir> cd \MYAPPJ
```

and issue the following command to create a *Java keystore* for the application:

```
C:\MYAPPJ> runmqckm -keydb -create -db myappj.jks -type jks
```

You will be prompted to create a password. Wait for the prompt, then type the password you want to use for the Java application's key repository.

Notes:

- It is important that this password is kept safe, as it protects the private key of the signed certificate. This password is required for later commands in this section.
- The certificate repositories used by Java client applications are of a different format to those used by queue managers, C and C++ client applications.
- The type of key repository is a *Java keystore*. This is a file format that can be used with Java Secure Socket Extension (JSSE). A

JSSE implementation is supplied with recent versions of the Java Runtime Environment.

- Because the WebSphere MQ Explorer connects to queue managers using the same classes as a normal MQ Java client application, this method is equally applicable to both.
- By using `-type jks`, the GSKit iKeycmd interface can be used to administer Java keystore key repositories, as described in this section.

Deployment note:

- Only the user under which the application runs should have access to the files in `C:\MYAPPJ`, to protect the integrity of your security policy. **IMPORTANT:** Set the permissions appropriately for your environment. If necessary, to help you implement an appropriate security policy, consult with the team in your organisation that decides security policy.

Step 4.2: Create a certificate request

Enter the following command:

```
C:\MYAPPJ> runmqckm -certreq -create -db myappj.jks -type jks
-dn "CN=Client Identifier,O=IBM,OU=WMQ,L=Hursley,C=UK" -label
"ibmwebspheremqmyuserid" -file myappj.req
```

Notes:

- The distinguished name (DN), as specified in the -dn parameter, should be tailored to your organisation and uniquely identify the Java client application within the infrastructure.

Step 4.3: Transfer the request

Transfer the certificate request file, `myappj.req`, to the directory where the CA files are located. Then change to this directory:

```
C:\MYAPPJ> copy myappj.req \myCAdir
C:\MYAPPJ> cd \myCAdir
```

Step 4.4: Sign the application's certificate

Run the following:

```
C:\myCAdir> runmqckm -cert -sign -db myCA.kdb -label
"myCAcertificate" -expire 365 -format ascii -file myappj.req
-target myappj.cer
```

Notes:

- This command will prompt you for a password. Supply the CA key repository's password. This was chosen and first used in Step 1.1.
- The `-expiry` parameter specifies the number of days, from the current date, until the signed certificate expires. The date of expiry cannot be later than the date of expiry of the CA certificate.

Step 4.5: Transfer the CA-signed certificate

Transfer the signed certificate, `myappj.cer` , and the public certificate of the CA, `myCAcertfile.cer` , back to `C:\MYAPPJ`

```
C:\myCAdir> copy myappj.cer \MYAPPJ
C:\myCAdir> copy myCAcertfile.cer \MYAPPJ
C:\myCAdir> cd \MYAPPJ
```

Step 4.6: "Add" the CA certificate

Add the CA certificate to the Java client application's keystore:

```
C:\MYAPPJ> runmqckm -cert -add -db myappj.jks -type jks -file
myCAcertfile.cer -label "theCAcertificate"
```

Notes:

- This command will prompt you for a password. Supply the Java application's keystore password. This was chosen and first used in Step 4.1.
- In order to use a certificate signed by the CA, and to trust other certificates signed by that CA, the keystore of the application must contain the CA certificate.
- The label we used for the CA certificate, `theCAcertificate` , was different from the label it had in the CA's own key repository. It is

not necessary to do this, but it has been done here to show that it is possible.

- Take note of the terminology used here. We have just **added** the public certificate of the CA to the key repository of the queue manager. This is a different action from **receiving** a certificate.

Step 4.7: "Receive" the signed certificate

Receive the certificate (now signed by the CA) into the Java client application's keystore:

```
C:\MYAPPJ> runmqckm -cert -receive -db myappj.jks -type jks
-file myappj.cer
```

Notes:

- This command will prompt you for a password. Supply the Java application's keystore password. This was chosen and first used in Step 4.1.
- The signed certificate must be **received** back into the same keystore that was used to create the certificate request.
- This keystore is the only place where the private key exists. The private key is uniquely partnered with the signed certificate.

Steps 5.x: Run a sample C application to show it works

The steps outlined here assume that you are logged in as a user in either the Administrators group, or the mqm group. This is necessary because some administration steps are performed (start queue manager, define channel, start listener).

IMPORTANT: Running applications as a user in the mqm group is **not** a typical way to operate your WebSphere MQ system. It is done here simply to provide the important information in the minimum number of steps.

In Step 3.2 the certificate label chosen was `ibmwebspheremqmyuserid`. In order to use this certificate, you must be logged in as `myuserid` when running the application.

Step 5.1: Create a SVRCONN channel definition that uses SSL

Start the queue manager if it is not already started.

```
C:\REPOS> strmqm myqmgr
```

An appropriate SVRCONN channel definition must be created. This can be done from any directory. In this example the directory `C:\REPOS` is used.

```
C:\REPOS> runmqsc myqmgr
DEFINE CHANNEL(MY.SEC.SVRCONN.CHL) CHLTYPE(SVRCONN) TRPTYPE(TCP)
SSLCAUTH(REQUIRED) SSLCIPH(RC4_MD5_US)
```

Step 5.2: Start a listener

If a listener is not already running for the queue manager, start one in a separate window:

```
C:\REPOS> start runmqlsr -m myqmgr -t tcp -p 1414
```

Note:

- If you start a listener in this way, it will stop running when you log out from your machine. This command is simply an example of how it *can* be done.

Step 5.3: Set up the application environment

Enter the following, to set up the environment:

```
C:\REPOS> cd \MYAPP
C:\MYAPP> set MQSSLKEYR=C:\MYAPP\myapp
```

Notes:

- For C or C++ client applications, the location and name of the certificate repository accessed by the client application can be configured in any of the following ways:
  - Outside the program, by setting the MQSSLKEYR environment variable.
  - Within a C program using the MQI, by setting the contents of the KeyRepository field in an MQSCO structure passed into an MQCONNX call.
  - Within a C++ program using the WebSphere MQ C++ classes, by using the setKeyRepository method on the imqQueueManager object.
- The user running the client application must have read access to the `.kdb`, `.crl`, `.rdb` and `.sth` files. Other users should be denied any access to these files.

Step 5.4: Build a small C program

Build (in a combined compile/link) the program below, using the following command.

```
C:\>MYAPP> cl.exe PutSample.c /Ic:\mqm\tools\c\include /link
c:\mqm\tools\lib\mqic32.lib
```

Notes:

- This command assumes that the WebSphere MQ installation is in `C:\mqm`, but a more common location is `C:\Program Files\IBM\WebSphere MQ`.

- The `cl.exe` program is the C/C++ compiler that is part of the Microsoft Visual Studio product. You will need this compiler installed in order to execute the above command.

Put the following program in the file `C:\MYAPP\PutSample.c` to make the above `cl.exe` command work.

```
#include <windows.h>
#include <cmqc.h>
#include <cmqxc.h>
/* usage: PutSample.exe qname qmgrname */

void exitOnError(char * apiName, MQLONG reason)
{
  if ( reason != MQRC_NONE )
  {
    printf( "Did not expect reason %d from %s\n"
          , reason, apiName);
    exit(1);
  }
}

int main(int argc, char ** argv)
{
  /* setup CLNTCONN channel details */
  MQCD  cd  = { MQCD_CLIENT_CONN_DEFAULT };
  MQSCO sco = { MQSCO_DEFAULT };
  MQCNO cno = { MQCNO_DEFAULT };
  MQOD  od  = { MQOD_DEFAULT };
  MQMD  md  = { MQMD_DEFAULT };
  MQPMO pmo = { MQPMO_DEFAULT };
  MQLONG  compcode = MQCC_OK;
  MQLONG  reason   = MQRC_NONE;
  MQHCONN hConn    = MQHC_UNUSABLE_HCONN;
  MQHOBJ  hObj     = MQHO_UNUSABLE_HOBJ;
  char buf[16]     = "Hello, World!";

  /* check we have enough program arguments */
  if ( argc != 3 )
    exit(2);

  /* CLNTCONN channel details */
  strncpy( cd.ChannelName
         , "MY.SEC.SVRCONN.CHL"
         , sizeof(cd.ChannelName) );
  strncpy( cd.ConnectionName
         , "localhost(1414)"
         , sizeof(cd.ConnectionName) );
  strncpy( cd.SSLCipherSpec
         , "RC4_MD5_US"
         , sizeof(cd.SSLCipherSpec) );

  /* connect */
  cno.Version = MQCNO_CURRENT_VERSION;
  cd.Version = MQCD_CURRENT_VERSION;
  cno.ClientConnPtr = &cd;
  MQCONNX( argv[2]
```

```
        , &cno
        , &hConn
        , &compcode
        , &reason );
  exitOnError("MQCONNX", reason);
  printf("MQCONNX successful\n");

  /* open */
  strncpy( od.ObjectName
         , argv[1]
         , sizeof(od.ObjectName) );
  MQOPEN( hConn
        , &od
        , MQOO_OUTPUT
        , &hObj
        , &compcode
        , &reason );
  exitOnError("MQOPEN", reason);
  printf("MQOPEN successful\n");

  memcpy( md.Format
        , MQFMT_STRING
        , sizeof(md.Format) );

  /* put */
  MQPUT( hConn
       , hObj
       , &md
       , &pmo
       , sizeof(buf)
       , buf
       , &compcode
       , &reason );
  exitOnError("MQPUT", reason);
  printf("MQPUT successful\n");

  /* cleanup (1) */
  MQCLOSE( hConn
         , &hObj
         , MQCO_NONE
         , &compcode
         , &reason );
  exitOnError("MQCLOSE", reason);

  /* give user 30 seconds to run DISPLAY CHSTATUS */
  printf("Sleeping for 30 seconds\n");
  Sleep(30000);

  /* cleanup (2) */
  MQDISC( &hConn
        , &compcode
        , &reason );
  exitOnError("MQDISC", reason);
  printf("Success\n");
}
```

This is some sample output from the combined compile/link:

```
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version
13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights
reserved.

PutSample.c
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:PutSample.exe
c:\mqm\tools\lib\mqic32.lib
PutSample.obj
```

Step 5.5: Run the application

```
C:\MYAPP> PutSample.exe SYSTEM.DEFAULT.LOCAL.QUEUE myqmgr
```

Note:

- If any of the previous steps have been incorrectly followed, it's likely an error message will be output from the application. For example, if you see an error message "`Did not expect reason 2381 from MQCONNX`" then one possible reason is that you omitted Step 5.3. For this application, it is vital for the MQSSLKEYR environment variable to be set correctly.

Step 5.6: Verify the channel status while the application runs

Notice that the application source code contains a `Sleep(30000)` call to give a delay of thirty seconds.

To verify that the channel is running with SSL capability:

- In one command window, run the PutSample.exe program again (see Step 5.5, above).
- In a second command window, enter the commands in bold text below, and notice the output:

```
C:\mydir> runmqsc myqmgr
5724-H72 (C) Copyright IBM Corp. 1994, 2004.  ALL RIGHTS
RESERVED.
Starting MQSC for queue manager myqmgr.


DISPLAY CHSTATUS(*) ALL
     1 : DISPLAY CHSTATUS(*) ALL
AMQ8417: Display Channel Status details.
   CHANNEL(MY.SEC.SVRCONN.CHL)             CHLTYPE(SVRCONN)
   BUFSRCVD(7)                             BUFSSENT(6)
```

```
   BYTSRCVD(1556)                          BYTSSENT(1376)
   CHSTADA(2006-04-25)                     CHSTATI(22.05.03)
   COMPHDR(NONE,NONE)                      COMPMSG(NONE,NONE)
   COMPRATE(0,0)                           COMPTIME(0,0)
   CONNAME(127.0.0.1)                      CURRENT
   EXITTIME(0,0)                           HBINT(300)
   JOBNAME(00000C4C00000788)               LOCLADDR( )
   LSTMSGDA(2006-04-25)                    LSTMSGTI(22.05.03)
   MCASTAT(RUNNING)                        MCAUSER(myuserid)
   MONCHL(OFF)                             MSGS(4)
   RAPPLTAG(C:\MYAPP\tmp.exe)              RQMNAME( )
SSLCERTI(CN=myCAName,OU=myDepartment,O=myOrganisation,L=myLocati
on,C=UK)
   SSLKEYDA( )                             SSLKEYTI( )
   SSLPEER(CN=myAppName,OU=myDepartment,O=IBM,L=Hursley,C=UK)
   SSLRKEYS(0)                             STATUS(RUNNING)
   STOPREQ(NO)                             SUBSTATE(RECEIVE)
   XMITQ( )
```

Notes:

- The SSL settings on the channel are evident in the channel status output.
- If you wait more than 30 seconds before running DISPLAY CHSTATUS, you will probably see the following message: "`AMQ8420: Channel Status not found.`" If you need more time to switch between windows and run commands, change the code to say: `Sleep(60000)` and repeat the above steps from Step 5.4. This will give you 60 seconds in which to switch between windows and run the command.

Steps 6.x Run a sample Java application to show it works

The steps outlined here assume that you are logged in as a user in either the Administrators group, or the mqm group. This is necessary because some administration steps are performed (start queue manager, define channel, start listener).

Running applications as a user in the mqm group is not a typical way to operate your WebSphere MQ system. It is done here simply to provide the important information in the minimum number of steps.

In Step 4.2 the certificate label chosen was `ibmwebspheremqmyuserid`. In order to use this certificate, you must be logged in as `myuserid` when running the application.

Step 6.1: Create a SVRCONN channel definition that uses SSL

Start the queue manager if it is not already started.

```
C:\REPOS> strmqm myqmgr
```

An appropriate SVRCONN channel definition must be created. This can be done from any directory. In this example the directory `C:\REPOS` is used.

```
C:\REPOS> runmqsc myqmgr
DEFINE CHANNEL(MY.SEC.SVRCONN.CHL) CHLTYPE(SVRCONN) TRPTYPE(TCP)
SSLCAUTH(REQUIRED) SSLCIPH(RC4_MD5_US)
```

Step 6.2: Start a listener

If a listener is not already running for the queue manager, start one in a separate window:

```
C:\REPOS> start runmqlsr -m myqmgr -t tcp -p 1414
```

Note:

- If you start a listener in this way, it will stop running when you log out from your machine. This command is simply an example of how it *can* be done.

Step 6.3: Compile a small Java program

Compile the program below, using the following command.

```
C:\>MYAPPJ> javac -classpath C:\mqm\Java\lib\com.ibm.mq.jar
PutSample.java
```

Note:

- This command assumes that the WebSphere MQ installation is in `C:\mqm` , but a more common location is `C:\Program Files\IBM\WebSphere MQ` .

Put the following program in the file `C:\MYAPPJ\PutSample.java` to make the above `javac` command work.

```
import com.ibm.mq.*;

public class PutSample
{
  // usage: java PutSample qname qmgrname
  //
  public static void main(String args[])
  {
    try
    {
      // setup CLNTCONN channel details
      MQEnvironment.hostname = "localhost";
      MQEnvironment.port     = 1414;
      MQEnvironment.channel  = "MY.SEC.SVRCONN.CHL";
      MQEnvironment.sslCipherSuite = "SSL_RSA_WITH_RC4_128_MD5";

      // connect
      MQQueueManager qMgr = new MQQueueManager(args[1]);
      System.out.println("qmgr construction successful");

      // open
      int OpenOptions = MQC.MQOO_OUTPUT;
      MQQueue queue = qMgr.accessQueue(args[0], OpenOptions);
      System.out.println("qmgr.accessQueue() successful");

      // create message
      MQMessage msg = new MQMessage();
      msg.format = MQC.MQFMT_STRING;
      msg.writeString("Hello, World!");
      MQPutMessageOptions pmo = new MQPutMessageOptions();

      // put
      queue.put(msg, pmo);
      System.out.println("queue.put() successful");

      // cleanup (1)
```

```
      queue.close();

      // give user 30 seconds to run DISPLAY CHSTATUS
      System.out.println("Sleeping for 30 seconds");
      try
      {
        java.lang.Thread.sleep(30000);
      }
      catch (InterruptedException ie)
      {
        System.out.println("InterruptedException: " + ie);
      }

      // cleanup (2)
      qMgr.disconnect();
      System.out.println("Success");
    }
    catch (MQException ex)
    {
      System.out.println("MQException: " + ex);
      if ( ex.reasonCode == MQException.MQRC_JSSE_ERROR )
        System.out.println("JSSE Exception: " + ex.getCause());
    }
    catch (java.io.IOException ex)
    {
      System.out.println("IOException: " + ex);
    }
  }
}
```

Note:

- The following properties have been used to specify the details of the Java keystore:

Property	**Notes**
javax.net.ssl.keyStore	The full path of the Java keystore; used to access the personal certificate
javax.net.ssl.keyStorePassword	The password required to access the Java keystore
javax.net.ssl.trustStore	The full path of the Java keystore containing the CA certificate. NB. It is possible to specify different Java keystores for the application and CA certificates, but in this example all required certificates are contained in a single repository

Step 6.4: Run the application

```
C:\MYAPPJ> set CLASSPATH=C:\mqm\Java\lib\com.ibm.mq.jar;.
C:\MYAPPJ> java -Djavax.net.ssl.keyStore=C:\MYAPPJ\myappj.jks
-Djavax.net.ssl.keyStorePassword=myappjpwd
-Djavax.net.ssl.trustStore=C:\MYAPPJ\myappj.jks PutSample
SYSTEM.DEFAULT.LOCAL.QUEUE myqmgr
```

Notes:

- In the above command, `CLASSPATH` is set up assuming the WebSphere MQ installation is at `C:\mqm`
- With the system set up as described throughout this document, all connections in a Java application must use the same Java keystore. It is possible for different connections to use different Java keystores, by creating their own SSLSocketFactory objects using the JSSE API. However, the details of this are beyond the scope of this document.
- In the WebSphere MQ Explorer, the preferences dialog is used to specify the location and password for the Java keystore.
- Please review APAR IC47466 before attempting to secure a connection from the WebSphere MQ Explorer to a queue manager using SSL. Use the following URL:
  `http://www.ibm.com/support/docview.wss?rs=171&uid=swg1IC47466`

Step 6.5: Verify the channel status while the application runs

Notice that the application source code contains a `java.lang.Thread.sleep(30000)` call to give a delay of thirty seconds.

To verify that the channel is running with SSL capability:

- In one command window, run the PutSample program again (see Step 6.4, above).
- In a second command window, enter the commands in bold text below, and notice the output:

```
C:\mydir> runmqsc myqmgr
5724-H72 (C) Copyright IBM Corp. 1994, 2004.  ALL RIGHTS
RESERVED.
Starting MQSC for queue manager myqmgr.


DISPLAY CHSTATUS(*) ALL
     1 : DISPLAY CHSTATUS(*) ALL
AMQ8417: Display Channel Status details.
   CHANNEL(MY.SEC.SVRCONN.CHL)             CHLTYPE(SVRCONN)
   BUFSRCVD(8)                             BUFSSENT(7)
```

```
   BYTSRCVD(1669)                          BYTSSENT(1632)
   CHSTADA(2006-04-25)                     CHSTATI(22.31.48)
   COMPHDR(NONE,NONE)                      COMPMSG(NONE,NONE)
   COMPRATE(0,0)                           COMPTIME(0,0)
   CONNAME(127.0.0.1)                      CURRENT
   EXITTIME(0,0)                           HBINT(300)
   JOBNAME(00000C4C00000988)               LOCLADDR( )
   LSTMSGDA(2006-04-25)                    LSTMSGTI(22.31.48)
   MCASTAT(RUNNING)                        MCAUSER( )
   MONCHL(OFF)                             MSGS(5)
   RAPPLTAG(WebSphere MQ Client for Java)
   RQMNAME( )
SSLCERTI(CN=myCAName,OU=myDepartment,O=myOrganisation,L=myLocati
on,C=UK)
   SSLKEYDA( )                             SSLKEYTI( )
   SSLPEER(CN=Client Identifier,OU=WMQ,O=IBM,L=Hursley,C=UK)
   SSLRKEYS(0)                             STATUS(RUNNING)
   STOPREQ(NO)                             SUBSTATE(RECEIVE)
   XMITQ( )
```

Notes:

- The SSL settings on the channel are evident in the channel status output.
- If you wait more than 30 seconds before running DISPLAY CHSTATUS, you will probably see the following message: "`AMQ8420: Channel Status not found.`" If you need more time to switch between windows and run commands, change the code to say: `java.lang.Thread.sleep(60000)` and repeat the above steps from Step 6.3. This will give you 60 seconds in which to switch between windows and run the command.

C. Further discussion

1. Why sign a certificate?

A certificate signed by your CA can be used to identify a single *entity* in your infrastructure, such as a queue manager, a client program, or a user.

By signing a certificate, the CA is certifying that it is trustworthy.

If you are acting as a CA, you must satisfy yourself somehow that the person (or workgroup, etc.) presenting you with a certificate request is indeed who they say they are. How you satisfy yourself of this is a subject beyond the scope of this document.

2. How to sign a certificate

Steps 2.x, Steps 3.x and Steps 4.x show how each entity can request a certificate. In each case, a file called a *certificate request file* (or PKCS #10 file) is created. This file contains an unsigned copy of the entity's certificate.

As the CA, you can then *sign* the certificate by issuing the necessary command. The output from the command is a *response file* that contains the signed certificate.

The certificate request file and the response file contain the *public certificate* of the entity, not its associated *private key*. The private key remains only in the key repository (or Java keystore) of the entity.

3. When to use an external CA

It isn't always necessary to manage your own CA within your enterprise. Another option is to purchase signed certificates from a trusted vendor. Some advantages of taking this approach include:

- The vendor manages expiry and renewal of certificates, to reduce risk associated with theft of certificates
- The vendor public certificate(s) are very likely to be installed on all systems in your enterprise. This means you might save any costs associated with transferring your CA's public certificate to all

systems in your enterprise – for example when a client system is re-installed after a major hardware or software failure.

4. How to transfer the certificate request file to the CA

The *certificate request file* contains Base64 encoded ASCII data. If you plan to use FTP to transfer this file between machines, then you should use the "ascii" command in the FTP client program before the "put" or "get" commands to ensure the file is transferred in the correct format.

Keep all *certificate request files*, and the CA-signed *response files*, in a safe place – for example, a single directory. This allows the CA to keep a record of the certificates that have been signed.

IMPORTANT: Set the permissions on these files and their directories appropriately for your environment. If necessary, to help you implement an appropriate security policy, consult with the team in your organisation that decides security policy.

5. Viewing the certificate files

Before transferring it back to the entity that signed it, you might want to view the details of the signed certificate. Windows has a built-in graphical viewer for certificates. A certificate can be viewed using this viewer, by issuing the following command:

```
C:\MYAPPJ> myappj.cer
```

6. How to transfer the certificates back from the CA

Two files should be transferred back to the entity (whether a queue manager, or an application):

- The CA certificate. (See Step 1.3 for details).
- The signed `.cer` certificate file. (See Steps 2.x, 3.x and 4.x for details).

Both of these X.509 certificates contain Base64 encoded ASCII data. The same mechanism can be used to transfer the certificate back to the entity, as was used to transfer the certificate request from the entity to the CA.