IBM KeyWorks Toolkit

Application Programming Interface (API) Specification

001.001.003

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Recently cryptography has come into widespread use in meeting multiple security needs, such as confidentiality, integrity, authentication and non-repudiation. In order to address these requirements in the emerging Internet, Intranet, and Extranet application domains, the IBM KeyWorks Toolkit was developed. The IBM KeyWorks Toolkit is a comprehensive set of layered security services suitable for use in operating systems such as IBM AIX, MVS, and OS/400. Windows NT/95, Solaris, and HP-UX are middleware that are embedded in applications, or are provided as a component of cryptographic security toolkits. The IBM KeyWorks Toolkit focuses on security in peer-to-peer, store-and-forward, and archival applications. It is designed to be compliant with industry standards such as OpenGroup, and is applicable to a broad range of hardware and operating system platforms. IBM KEYWORKS is intended to include full lifecycle key management and portable credentials. The definition of such a set of layered security services and an open architecture protects the investment made in implementation of security applications by facilitating the reuse of core components of the architecture for different products.

The security services available in the IBM KeyWorks Toolkit are defined by the categories of service provider modules that the architecture accommodates. These service providers are:

- Cryptographic Service Providers
- Key Recovery Service Providers
- Trust Policy Libraries
- Certificate Libraries
- Data Storage Libraries

The central component of this architecture is the IBM KeyWorks Framework, which is a layer of middleware that lies between application code and the service provider modules. The KeyWorks Framework is based on the Common Data Security Architecture (CDSA) and the Common Security Services Manager (CSSM) application programming interface (API) developed by Intel. The existing interfaces of CSSM have been enhanced in the IBM KeyWorks Toolkit to include key recovery features. Unlike basic security features such as cryptographic functions, certificates, and trust policies, key recovery is a relatively new field and is the focus of innovations related to the IBM KeyWorks Toolkit.

1.1 IBM KeyWorks Toolkit Architecture

The IBM KeyWorks Toolkit Architecture consists of a set of layered security services and associated programming interfaces designed to furnish an integrated set of information and communication security capabilities. Each layer builds on the more fundamental services of the layer directly below it.

These layers start with fundamental components such as cryptographic algorithms, random numbers, and unique identification information in the lower layers, and build up to digital certificates, key management and recovery mechanisms, and secure transaction protocols in higher layers. The IBM KeyWorks Architecture is intended to be the multiplatform security architecture that is both horizontally broad and vertically robust.

Figure 1 shows a simplified view of the layered architecture of an IBM KeyWorks-based system. There are four major layers in the IBM KeyWorks Toolkit Architecture: Application Domains, System Security Services, KeyWorks Framework, and Service Providers.

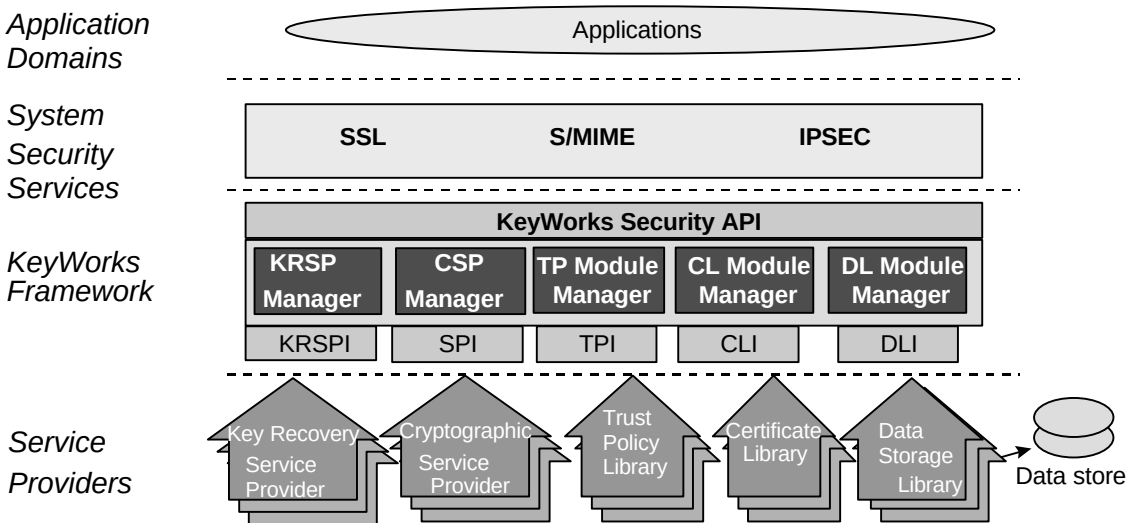

Figure 1. IBM KeyWorks Toolkit Architecture

The Application Domains layer implements the application domain services such as Secure Electronic Transactions (SET) and E-Wallet, E-mail services, or file archival services. The System Security Services layer is between the Application Domains layer and the IBM KeyWorks Framework layer. It implements security protocols that are used by the Application Domains layer. Software at this layer may implement cryptographic system security services such as Security Sockets Layer (SSL), Internet Protocol Security (IPSEC), Secure/Multipurpose Internet Mail Extensions (S/MIME), and Electronic Data Interchange (EDI). The System Security Services layer also includes tools and utilities for installing, configuring, and maintaining the KeyWorks Framework and service provider modules.

The KeyWorks Framework is the central component of this extensible architecture that provides mechanisms to dynamically manage service provider modules. The KeyWorks Framework defines a common security API that must be used to access services of service provider modules. Applications request security services through the KeyWorks Security API or through system security services implemented over the KeyWorks API. The service provider modules actually perform the requested security services. IBM provides a number of service provider modules. Additional service provider modules may be available from other Independent Software Vendors (ISVs) and hardware vendors. Applications may direct their requests to modules from specific vendors or to any module that performs the required services. Both the KeyWorks Framework and the service provider interfaces (SPIs) are discussed in detail in this document.

1.2 Intended Audience

This document provides an overview of the IBM KeyWorks Toolkit for ISVs who develop their own operating systems or other security products either as complete applications or as plug-ins to extensible platforms. This document is intended for use by:

- Advanced programmers
- Experienced software designers
- Security architects who work in high-end cryptography
- Sophisticated integrators familiar with numerous forms of network computing
- Vendors of customizable service providers for cryptographic, trust, and database services

This audience understands the requirements for a ubiquitous security infrastructure upon which they can build security-aware application products.

1.3 Documentation Set

The IBM KeyWorks Toolkit documentation set consists of the following manuals. These manuals are provided in electronic format and can be viewed using the Adobe Acrobat Reader distributed with the IBM KeyWorks Toolkit. Both the electronic manuals and the Adobe Acrobat Reader are located in the IBM KeyWorks Toolkit `doc` subdirectory.

- *IBM KeyWorks Toolkit Developer's Guide*
  Document filename: kw_dev.pdf
  This document presents an overview of the IBM KeyWorks Toolkit. It explains how to integrate KeyWorks into applications and contains a sample KeyWorks application.

- *IBM KeyWorks Toolkit Application Programming Interface Specification*
  Document filename: kw_api.pdf
  This document defines the interface that applications developers employ to access security services provided by the IBM KeyWorks Framework and service provider modules.

- *IBM KeyWorks Toolkit Service Provider Module Structure & Administration*
  Document filename: kw_mod.pdf
  This document describes the features common to all IBM KeyWorks service provider modules. It should be used in conjunction with the individual IBM KeyWorks service provider interface specifications in order to build a service provider module.

- *IBM KeyWorks Toolkit Cryptographic Service Provider Interface Specification*
  Document filename: kw_spi.pdf
  This document defines the interface to which cryptographic service provider modules must conform in order to be accessible through IBM KeyWorks.

- *Key Recovery Service Provider Interface Specification*
  Document filename: kr_spi.pdf
  This document defines the interface to which key recovery service provider modules must conform in order to be accessible through IBM KeyWorks.

- *Key Recovery Server Installation and Usage Guide*
  Document filename: krs_gd.pdf
  This document describes how to install and use key recovery solutions using the components in the IBM Key Recovery Server.

- *IBM KeyWorks Toolkit Trust Policy Interface Specification*
  Document filename: kw_tp_spi.pdf
  This document defines the interface to which policy makers, such as certificate authorities (CAs), certificate issuers, and policy-making application developers, must conform in order to extend IBM KeyWorks with model or application-specific policies.

- *IBM KeyWorks Toolkit Certificate Library Interface Specification*
  Document filename: kw_cl_spi.pdf
  This document defines the interface to which certificate library developers must conform to provide format-specific certificate manipulation services to numerous IBM KeyWorks applications and trust policy modules.

- *IBM KeyWorks Toolkit Data Storage Library Interface Specification*
  Document filename: kw_dl_spi.pdf
  This document defines the interface to which library developers must conform to provide format-specific or format-independent persistent storage of certificates.

1.4 References

Cryptography	*Applied Cryptograph*, Schneier, Bruce, 2nd Edition, John Wiley and Sons, Inc., 1996.
	Handbook of Applied Cryptography, Menezes, A., Van Oorschot, P., and Vanstone, S., CRC Press, Inc., 1997.
	SDSI - A Simple Distributed Security Infrastructure, R. Rivest and B. Lampson, 1996.
	Microsoft CryptoAPI, Version 0.9, Microsoft Corporation, January 17, 1996.
CDSA Spec	*Common Data Security Architecture Specification,* Intel Architecture Labs, 1997.
CSSM API	*Common Security Services Manager Application Programming Interface Specification*, Intel Architecture Labs, 1997.
Key Escrow	*A Taxonomy for Key Escrow Encryption Systems*, Denning, Dorothy E. and Branstad, Dennis, Communications of the ACM, Vol. 39, No. 3, March 1996.
PKCS	*The Public-Key Cryptography Standards*, RSA Laboratories, Redwood City, CA: RSA Data Security, Inc.
IBM KeyWorks CLI	*Certificate Library Interface Specification,* Intel Architecture Labs, 1997.
IBM KeyWorks DLI	*Data Storage Library Interface Specification,* Intel Architecture Labs, 1997.
IBM KeyWorks KRI	*Key Recovery Service Provider Interface Specification,* Intel Architecture Labs, 1997.
IBM KeyWorks SPI	*Cryptographic Service Provider Interface Specification*, Intel Architecture Labs, 1997.
IBM KeyWorks TPI	*Trust Policy Interface Specification,* Intel Architecture Labs, 1997.
X.509	*CCITT. Recommendation X.509: The Directory – Authentication Framework,* 1988. CCITT stands for Comite Consultatif Internationale Telegraphique et Telephonique (International Telegraph and Telephone Consultative Committee)

Chapter 2. Core Services API

The IBM KeyWorks provides the following set of services:

- Module Management
- Memory Management Support
- Security Context Management
- Integrity Verification Services

These APIs are implemented by the IBM KeyWorks, not by service provider modules.

2.1 Module Management Services

The IBM KeyWorks module management functions support module installation, dynamic selection and loading of modules, and querying of module features and status. System administration utilities use KeyWorks install and uninstall functions to maintain service provider modules on a local system.

Applications select the particular security services they will use by selectively attaching service provider modules. These modules are provided by Independent Software Vendors (ISVs). Each module has an assigned Globally Unique ID (GUID) and a set of descriptive attributes to assist applications in selecting appropriate modules for their use. A module can implement a range of services across the KeyWorks APIs (e.g., cryptographic functions, data storage functions) or a module can restrict its services to a single KeyWorks category of service (e.g., Certificate Library (CL) services only). Modules that span service categories are called multiservice modules.

Applications use a module's GUID to specify the module to be attached. The CSSM_ModuleAttach function returns a handle representing a unique pairing between the caller and the attached module. Applications must provide this handle as an input parameter when requesting services from the attached module. IBM KeyWorks uses the handle to match the caller with the appropriate service module.

The calling application uses the handle to obtain all types of services implemented by the attached module. Figure 2 shows how the handle for an attached Public-Key Cryptographic Standard (PKCS#11) service provider is used to perform cryptographic operations and persistent storage of certificates. The single handle value can be used as the CSPHandle in cryptographic operations and as the DLHandle in data storage operations.

Multiple calls to attach are viewed as independent requests. Each attach request returns separate, independent handles that do not share execution state.

Before attaching a service module, an application can query the KeyWorks registry to obtain information on:

- Modules installed on the system
- Capabilities (and functions) implemented by those modules
- GUID associated with a given module

Applications use this information to select a module for use. A multiservice module has multiple capability descriptions associated with it. At least one per functional area supported by the module. Some areas, such as Cryptographic Service Provider (CSP) and Trust Policy (TP), may have multiple independent capability descriptions for a single functional area. There is one KeyWorks registry entry for a multiservice module. That entry records all service types for the module. KeyWorks returns all information about a module's capabilities when queried by the application.

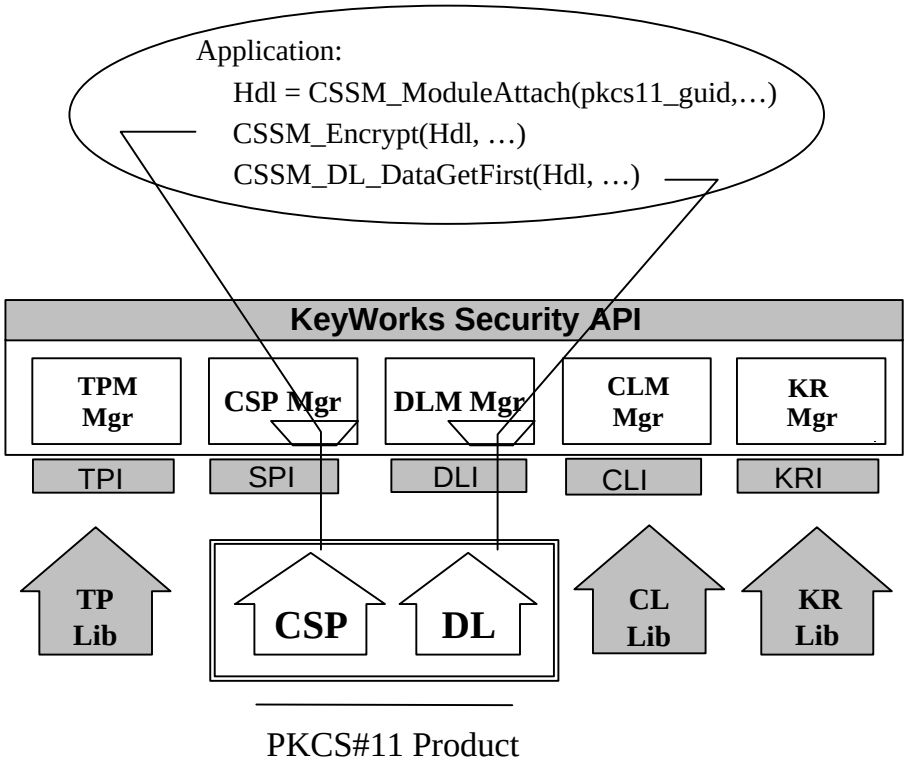

Figure 2. Application using cryptographic services and persistent storage services of a class 2, PKCS#11 device.

Applications can query about KeyWorks themselves. KeyWorks provides several functions to assist applications in ensuring that the current KeyWorks version meets the application's needs. One function returns version information about KeyWorks. Another function verifies whether the application's expected KeyWorks version is compatible with the currently running KeyWorks version. (The general function to query service provider module information also returns the module's version information.)

2.2 Memory Management Support

The KeyWorks memory management functions are a class of routines for reclaiming memory allocated by KeyWorks on behalf of an application from the KeyWorks memory heap. When KeyWorks allocates objects from its own heap and returns them to an application, the application must inform KeyWorks when it no longer requires the use of that object. Applications use specific APIs to free KeyWorks-allocated memory. When an application invokes a free function, KeyWorks can choose to retain or free the indicated object depending on other conditions known only to KeyWorks. In this way, KeyWorks and applications work together to manage these objects in the KeyWorks memory heap.

2.3 Security Context Management

The KeyWorks framework is responsible for maintaining data that may be required to perform cryptographic and security operations. The internal context structure maintains information pertaining to the parameters of the cryptographic operation, such as the type of algorithm to be performed, and maintains a list of attributes to customize the information stored in the context. These attributes can be of different types, including keys, dates, and raw data buffers. When the application creates a context, it supplies a set of parameters based on what type of context it is, and the framework returns a handle to that

context. The application can then use that handle to add additional attributes to the framework, and update the contents of the existing attributes. The context handle is passed to the functions that perform the actual cryptographic operations. The data and attributes are retrieved from the context management system for use by the addin performing the operations. When the application is done with a context, it should pass the handle to the CSSM_ContextDelete function in order to free up the memory used by that context.

2.4 Integrity Verification Services

As a security framework, IBM KeyWorks provides each application with additional assurance of the integrity of the KeyWorks environment in which the application is running. With dynamic link-loading of service provider modules, viruses and other forms of impersonation are real threats. KeyWorks reduces the risk of these threats by requiring digitally signed modules and by dynamically checking the identity and integrity of each module at attach time. Verification improves the chances that any modification, whether accidental or malicious, may be detected prior to performing trusted operations.

Module verification has the following three aspects:

- Verification of module identity based on a digitally signed certificate
- Verification of object code integrity based on a signed hash of the object
- Tightly binding the verified module identity with the verified set of object code

2.5 Data Structures for Core Services

2.5.1 Basic Data Types

```
typedef unsigned char uint8;
typedef unsigned short uint16;
typedef short sint16;
typedef unsigned int uint32;
typedef int sint32;

#define CSSM_MODULE_STRING_SIZE  64
typedef char CSSM_STRING [CSSM_MODULE_STRING_SIZE + 4];
```

2.5.2 CSSM_ALL_SUBSERVICES

```
#define  CSSM_ALL_SUBSERVICES      (-1)
```

2.5.3 CSSM_BOOL

This data type is used to indicate a true or false condition.

```
typedef uint32 CSSM_BOOL;

#define CSSM_TRUE 1
define CSSM_FALSE 0
```

Definitions:

CSSM_TRUE - Indicates a true result or a true value.

CSSM_FALSE - Indicates a false result or a false value.

2.5.4 CSSM_COUNTRY_ORIGIN

```
typedef enum cssm_country_origin {
   CSSM_COUNTRY_US = 1,
   CSSM_COUNTRY_NONUS = 2
} CSSM_COUNTRY_ORIGIN;
```

2.5.5 CSSM_CRYPTO_TYPE

```
typedef enum cssm_crypto_type {
   CSP_TYPE_NONE = 0,
   CSP_TYPE_EXPORT = 1,
   CSP_TYPE_SSL = 2,
   CSP_TYPE_FINANCIAL = 3,
   CSP_TYPE_EXPORTVERIFY = 4,
   CSP_TYPE_AUTHENTICATE = 5
} CSSM_CRYPTO_TYPE;
```

2.5.6 CSSM_CSP_MANIFEST

```
typedef struct cssm_csp_manifest {
   char *Vendor;
   CSSM_COUNTRY_ORIGIN CountryOrigin;
   CSSM_CRYPTO_TYPE CryptoType;
   uint32 NumberCapabilities;
   CSSM_CSP_CAPABILITY_PTR Capabilities;
} CSSM_CSP_MANIFEST, * CSSM_CSP_MANIFEST_PTR;
```

2.5.7 CSSM_CSSMINFO

This data structure represents the information associated with an installation of KeyWorks.

```
typedef struct cssm_cssminfo {
   CSSM_VERSION Version;
   char *Description
   char *Vendor
   CSSM_BOOL ThreadSafe;
   char *Location;
   CSSM_GUID GUID;
}CSSM_CSSMINFO, *CSSM_CSSMINFO_PTR
```

2.5.8 CSSM_DATA

The CSSM_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory. This memory must be allocated and freed using the memory management routines provided by the calling application through KeyWorks. TP modules and CLs use this structure to hold certificates and Certificate Revocation Lists (CRLs). Other service modules, such as CSPs, use this same structure to hold general data buffers. DL modules use this structure to hold persistent security-related objects.

```
typedef struct cssm_data{
   uint32 Length;/* in bytes */
   uint8 *Data;
} CSSM_DATA, *CSSM_DATA_PTR
```

Definitions:

Length - Length of the data buffer in bytes.

Data - Points to the start of an arbitrary length data buffer.

2.5.9 CSSM_EVENT_TYPE

```
typedef uint32 CSSM_EVENT_TYPE, *CSSM_EVENT_TYPE_PTR;

#define CSSM_EVENT_ATTACH          (0)
#define CSSM_EVENT_DETACH          (1)
#define CSSM_EVENT_INFOATTACH      (2)
#define CSSM_EVENT_INFODETACH      (3)
#define CSSM_EVENT_CREATE_CONTEXT(4)
```

```
#define CSSM_EVENT_DELETE_CONTEXT (5)
```

2.5.10 CSSM_GUID

This structure designates a Globally Unique ID (GUID) that distinguishes one service provider module from another. All GUID values should be computer-generated to guarantee uniqueness. (The GUID generator in Microsoft Developer Studio and the RPC UUIDGEN/uuid_gen program can be used on a number of UNIX-based platforms.)

```
typedef struct cssm_guid{
   uint32 Data1;
   uint16 Data2;
   uint16 Data3;
   uint8Data4[8];
} CSSM_GUID, *CSSM_GUID_PTR
```

Definitions:

Data1 - Specifies the first 8 hexadecimal digits of the GUID.

Data2 - Specifies the first group of 4 hexadecimal digits of the GUID.

Data3 - Specifies the second group of 4 hexadecimal digits of the GUID.

Data4 - Specifies an array of 8 elements that contains the third and final group of 8 hexadecimal digits of the GUID in elements 0 and 1, and the final 12 hexadecimal digits of the GUID in elements 2 through 7

2.5.11 CSSM_INFO_LEVEL

This enumerated list defines the levels of information detail that can be retrieved about the services and capabilities implemented by a particular module. Modules can implement multiple KeyWorks service types. Modules can also present their services as subservices. Modules can also be dynamic with respect to the services and features they provide.

```
typedef enum cssm_info_level {
   CSSM_INFO_LEVEL_MODULE = 0,
                /* values from XXinfo struct */
   CSSM_INFO_LEVEL_SUBSERVICE  = 1,
                /* values from XXinfo and XXsubservice struct */
   CSSM_INFO_LEVEL_STATIC_ATTR       = 2,
                /* values from XXinfo and XXsubservice and all
                 static-valued attributes of a subservice */
   CSSM_INFO_LEVEL_ALL_ATTR     = 3,
                /* values from XXinfo and XXsubservice and all attributes,
                 static and dynamic, of a subservice */
} CSSM_INFO_LEVEL;
```

2.5.12 CSSM_LIST

This structure is used to encapsulate an array of CSSM_LIST_ITEMs, where the array length is given by the Length variable.

```
typedef struct cssm_list{
   uint32 NumberItems;
   CSSM_LIST_ITEM_PTR Items;
} CSSM_LIST, *CSSM_LIST_PTR
```

Definitions:

NumberItems - The number of items in the list.

Items - An array of pointers to the item structures.

2.5.13 CSSM_LIST_ITEM

This structure is used to encapsulate the name and GUID of a service provider module.

```
typedef struct cssm_list_item{
   CSSM_GUID GUID;
   char *Name;
} CSSM_LIST_ITEM, *CSSM_LIST_ITEM_PTR
```

Definitions:

GUID - The global unique identifier of the module.

Name - The name of the module.

2.5.14 CSSM_MODULE_FLAGS

```
typedef uint32 CSSM_MODULE_FLAGS;

#define CSSM_MODULE_THREADSAFE      0x1
#define CSSM_MODULE_EXPORTABLE      0x2
```

2.5.15 CSSM_MODULE_HANDLE

This structure is a unique identifier for an attached service provider module.

```
typedef uint32 CSSM_MODULE_HANDLE
```

2.5.16 CSSM_MODULEINFO

```
typedef struct cssm_moduleinfo {
   CSSM_VERSION Version; /* Module version */
   CSSM_VERSION CompatibleCSSMVersion; /* Module written for CSSM version */
   CSSM_STRING Description; /* Module description */
   CSSM_STRING Vendor;/* Vendor name, etc */
   CSSM_MODULE_FLAGS Flags; /* Flags to describe and control module use */
   CSSM_SERVICE_MASK ServiceMask; /* Bit mask of supported services */
   uint32 NumberOfServices; /* Num of services in ServiceList */
   CSSM_SERVICE_INFO_PTR ServiceList; /* Pointer to list of service infos */
   void* Reserved;
} CSSM_MODULE_INFO, *CSSM_MODULE_INFO_PTR;
```

2.5.17 CSSM_RETURN

This data type is used to indicate whether a function was successful.

```
typedef enum cssm_return {
      CSSM_OK = 0,
      CSSM_FAIL = -1
} CSSM_RETURN
```

Definitions:

CSSM_OK - Indicates operation was successful.

CSSM_FAIL - Indicates operation was unsuccessful.

2.5.18 CSSM_SERVICE_FLAGS

This defines a bit-mask that categorizes the type of service provided by a service provider module. It can contain any combination of CSSM_SERVICE_MASK values.

```
typedef uint32 CSSM_SERVICE_FLAGS;
```

2.5.19 CSSM_SERVICEINFO

```
typedef struct cssm_serviceinfo {
   CSSM_STRING Description;/* Service description */
   CSSM_SERVICE_TYPE Type;/* Service type */
   CSSM_SERVICE_FLAGS Flags;/* Service flags */

   uint32 NumberOfSubServices;/* Number of sub services in SubServiceList */
   union { /* List of sub services */
      void *SubServiceList;
      CSSM_CSPSUBSERVICE_PTR CspSubServiceList;
      CSSM_DLSUBSERVICE_PTR DlSubServiceList;
      CSSM_CLSUBSERVICE_PTR ClSubServiceList;
      CSSM_TPSUBSERVICE_PTR TpSubServiceList;
      CSSM_KRSPSUBSERVICE_PTR KrSubServiceList;
   };
   void* Reserved;
} CSSM_SERVICE_INFO, *CSSM_SERVICE_INFO_PTR;
```

2.5.20 CSSM_SERVICE_MASK

This defines a bit-mask of the possible categories of KeyWorks services that may be implemented by a single service provider module.

```
typedef uint32 CSSM_SERVICE_MASK;
typedef CSSM_SERVICE_MASK CSSM_SERVICE_TYPE;
```

CSSM_SERVICE_CSSM	0x1
CSSM_SERVICE_CSP	0x2
CSSM_SERVICE_DL	0x4

CSSM_SERVICE_CL	0x8
CSSM_SERVICE_TP	0x10
CSSM_SERVICE_KR	0x20
CSSM_SERVICE_LAST	CSSM_SERVICE_KR
CSSM_DB_DATASTORES_UNKNOWN	(0xFFFFFFFF)
CSSM_ALL_SUBSERVICES	(0xFFFFFFFF)

2.5.21 CSSM_USER_AUTHENTICATION

This structure holds the user's credentials for authenticating to the data storage library module. The type of credentials required is defined by the DL module and specified as a CSSM_USER_AUTHENTICATION_MECHANISM.

```
typedef struct cssm_user_authentication {
   CSSM_DATA_PTR Credential;
   CSSM_CRYPTO_DATA_PTR MoreAuthenticationData;
} CSSM_USER_AUTHENTICATION, *CSSM_USER_AUTHENTICATION_PTR;
```

Definitions:

Credential - A certificate, a shared secret, a token, or whatever the service provider module requires for user authentication. The required credential type is specified as a CSSM_USER_AUTHENTICATION_MECHANISM.

MoreAuthenticationData - A passphrase or other data that can be provided as immediate data within this structure or via a callback function to the user/caller.

2.5.22 CSSM_USER_AUTHENTICATION_MECHANISM

This enumerated list defines different methods a service provider module can require when authenticating a caller. The module specifies which mechanism the caller must use for each subservice type provided by the module. For example, the DL modules may require password-based authentication, may require a login sequence, or may accept a certificate and passphrase.

```
typedef enum cssm_user_authentication_mechanism {
   CSSM_AUTHENTICATION_NONE = 0,
   CSSM_AUTHENTICATION_CUSTOM = 1,
   CSSM_AUTHENTICATION_PASSWORD = 2,
   CSSM_AUTHENTICATION_USERID_AND_PASSWORD = 3,
   CSSM_AUTHENTICATION_CERTIFICATE_AND_PASSPHRASE = 4,
   CSSM_AUTHENTICATION_LOGIN_AND_WRAP = 5,
} CSSM_USER_AUTHENTICATION_MECHANISM;
```

2.5.23 CSSM_VERSION

This structure is used to represent the version of KeyWorks components.

```
typedef struct cssm_version {
   uint32 Major;
   uint32 Minor;
} CSSM_VERSION, *CSSM_VERSION_PTR
```

Definitions:

Major - The major version number of the component.

Minor - The minor version number of the component.

2.5.24 CSSM_FreeInfo

CSSM_RETURN CSSMAPI CSSM_FreeInfo (CSSM_INFO_PTR CssmInfo)

This function frees the memory allocated for the CSSM_INFO structure by the CSSM_GetInfo function.

Parameters

CssmInfo (input/output)
A pointer to the CSSM_INFO structure to be freed.

Return Value

A CSSM_OK return value signifies the memory has been freed. When CSSM_FAIL is returned, an error occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_GetInfo

2.5.25 CSSM_GetInfo

CSSM_INFO_PTR CSSMAPI CSSM_GetInfo (void)

This function returns the version information of the KeyWorks Framework.

Parameters

None

Return Value

A pointer to a CSSM_INFO structure. If the pointer is NULL, an error occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_FreeInfo

2.5.26 CSSM_Init

CSSM_RETURN CSSMAPI CSSM_Init (const CSSM_VERSION_PTR Version,
const CSSM_API_MEMORY_FUNCS_PTR MemoryFuncs,
const void * Reserved)

This function initializes KeyWorks and verifies that the version of KeyWorks expected by the application is compatible with the version of KeyWorks on the system. This function should be called only once by each application.

Parameters

Version (input)
The major and minor version number of the KeyWorks release the application is compatible with.

MemoryFuncs (input)
Memory functions for KeyWorks to use when allocating data structures for the application.

Reserved (input)
A reserved input.

Return Value

A CSSM_OK return value signifies the initialization operation was successful. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

2.6 Module Management Functions

2.6.1 CSSM_FreeModuleInfo

CSSM_RETURN CSSMAPI CSSM_FreeModuleInfo (CSSM_MODULE_INFO_PTR ModuleInfo)

This function frees the memory allocated by CSSM_GetModuleInfo to hold the module info structures. All substructures within the info structure are freed by this function.

Parameters

ModuleInfo (input)
A pointer to the CSSM_MODULE_INFO structures to be freed.

Return Value

This function returns CSSM_OK if successful, and returns CSSM_FAIL if an error has occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_GetModuleInfo, CSSM_SetModuleInfo

2.6.2 CSSM_GetCSSMRegistryPath

CSSM_DATA_PTR CSSMAPI CSSM_GetCSSMRegistryPath (void)

This function gets the directory path of the KeyWorks registry.

Parameters

None

Return Value

A pointer to a CSSM_DATA structure containing the registry path information or a NULL, if an error occurred in getting the information. Use CSSM_GetError to determine the exact error.

2.6.3 CSSM_GetGUIDUsage

CSSM_SERVICE_MASK CSSMAPI CSSM_GetGUIDUsage
(const CSSM_GUID_PTR ModuleGUID)

Returns a bit-mask describing the KeyWorks function categories of service provided by the module identified by GUID.

Parameters

ModuleGUID (input)
Globally Unique Identifier for the module of interest.

Return Value

A CSSM_SERVICE_MASK from the info structure describing the services provided by the module referenced by the GUID.

See Also

CSSM_GetHandleUsage

2.6.4 CSSM_GetHandleUsage

CSSM_SERVICE_MASK CSSMAPI CSSM_GetHandleUsage
(CSSM_HANDLE ModuleHandle)

Returns a bit-mask describing the KeyWorks function categories of service provided by the module, which is identified by the specified handle for an attached module.

Parameters

ModuleHandle (input)
Handle of the module for which information should be returned.

Return Value

A CSSM_SERVICE_MASK from the info structure describing the services provided by the module referenced by the handle.

See Also

CSSM_GetGUIDUsage

2.6.5 CSSM_GetModuleGUIDFromHandle

CSSM_GUID_PTR CSSMAPI CSSM_GetModuleGUIDFromHandle
(CSSM_HANDLE ModuleHandle)

This function determines the GUID associated with a specific module handle.

Parameters

ModuleHandle (input)
The handle that describes the service provider module.

Return Value

A CSSM_GUID_PTR to a data structure containing the GUID associated with *ModuleHandle*. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

2.6.6 CSSM_GetModuleInfo

CSSM_MODULE_INFO_PTR CSSMAPI CSSM_GetModuleInfo
(const CSSM_GUID_PTR ModuleGUID,
CSSM_SERVICE_MASK ServiceMask,
uint32 SubserviceID,
CSSM_INFO_LEVEL InfoLevel)

This function returns descriptive information about the module identified by the *ModuleGUID*. The information returned can include: all of the capability information, information for each subservice, or information for each of the service types implemented by the selected module. The request for information can be limited to a particular set of services, as specified by the *ServiceMask* bit-mask. The request may be further limited to one or all of the subservices implemented in one or all of the service categories. Finally, the detail level of the information returned can be controlled by the *InfoLevel* input parameter. This is particularly important for a module with dynamic capabilities. *InfoLevel* can be used to request static attribute values only or dynamic values.

Parameters

ModuleGUID (input)
A pointer to the CSSM_GUID structure containing the GUID for the service provider module.

ServiceMask (input)
A bit-mask specifying the module service types used to restrict the capabilities information returned by this function. An input value of zero specifies all services for the specified module.

SubserviceID (input)
A single subservice ID or the value CSSM_ALL_SUBSERVICES must be provided. If a subservice ID is provided, the get operation is limited to the specified subservice. Note that a service mask may already limit the operation. If so, the subservice ID applies to all service categories selected by the service mask. If CSSM_ALL_SUBSERVICES is specified, information for all subservices (as limited by the service mask) is returned by this function.

InfoLevel (input)
Indicates the level of detail returned by this function. Information retrieval can be restricted as follows. Note that not all service provider modules support all of the following values.

- CSSM_INFO_LEVEL_MODULE - Returns only the information contained in the cssm_moduleinfo structure.
- CSSM_INFO_LEVEL_SUBSERVICE - Returns the information returned by CSSM_INFO_LEVEL_MODULE and the information contained in the cssm_XXsubservice structure, where XX corresponds to the module type, such as cssm_tpsubservice.
- CSSM_INFO_LEVEL_STATIC_ATTR - Returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically defined for the module.
- CSSM_INFO_LEVEL_ALL_ATTR - Returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically or dynamically defined for the module. Dynamic modules, whose capabilities change over time, support a query function used by KeyWorks to interrogate the module's current capability status.

Return Value

A CSSM_MODULE_INFO_PTR to an array of one or more info structures. Each structure contains type information identifying the capability description as representing CL capabilities, DL capabilities, etc. The capability descriptions can also be subclassed into subservices.

See Also

CSSM_GetModuleInfo, CSSM_FreeModuleInfo

2.6.7 CSSM_GetModuleLocation

CSSM_DATA_PTR CSSMAPI CSSM_GetModuleLocation (const CSSM_GUID_PTR GUID)

This function returns the directory path of the service provider module specified by the *GUID* input parameter.

Parameters

GUID (input)
A pointer to the CSSM_GUID structure containing the GUID for the service provider module.

Return Value

A pointer to a CSSM_DATA data structure containing the directory path of the module associated with *GUID*. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

2.6.8 CSSM_ListModules

CSSM_LIST_PTR CSSMAPI CSSM_ListModules (CSSM_SERVICE_MASK
ServiceMask,CSSM_BOOL MatchAll)

This function returns a list containing the GUID/name pair for each of the currently installed service provider modules that provide services in any of the KeyWorks functional categories selected in the service mask.

Parameters

ServiceMask (input)
A bit-mask selecting the KeyWorks functional categories. This information can be used to select information about potential service provider modules.

MatchAll (input)
A Boolean value defining how the multiple bits in the service mask are interpreted. CSSM_TRUE means the service modules selected must match all service areas specified by the service mask. CSSM_FALSE means the service module selected must specify one or more of the service areas specified by the service mask.

Return Value

A pointer to the CSSM_LIST structure containing the GUID/name pair for each of the modules. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_GetModuleInfo, CSSM_FreeModuleInfo

2.6.9 CSSM_ModuleAttach

CSSM_MODULE_HANDLE CSSMAPI CSSM_ModuleAttach
(const CSSM_GUID_PTR GUID,
const CSSM_VERSION_PTR Version,
const CSSM_API_MEMORY_FUNCS_PTR MemoryFuncs,
uint32 SubserviceID,
uint32 SubserviceFlags,
uint32 Application,
const CSSM_NOTIFY_CALLBACK Notification,
const void * Reserved)

This function attaches the service provider module and verifies that the version of the module expected by the application is compatible with the version on the system. The module can implement subservices (as described in the service provider's documentation). The caller can specify a specific subservice provided by the module. Subservice flags may be required to set parameters for the service.

Parameters

GUID (input)
A pointer to the CSSM_GUID structure containing the GUID for the service provider module.

Version (input)
The major and minor version number of the service provider module with which the application is compatible.

MemoryFuncs (input)
Memory functions for KeyWorks to use when allocating data structures for the application.

SubserviceID (input)
The number of a subservice provided by the module. This value should always be taken from the ServiceMask field of the CSSM_MODULE_INFO structure to insure that a compatible identifier is used. (Service provider modules that implement only one service can use zero as the subservice identifier.)

SubserviceFlags(input)
Bit-mask of service options defined by a particular subservice of the module. Valid values are described in module-specific documentation. A default set of flags is specified in the CSSM_MODULE_INFO structure for use by the caller.

Application(input/optional)
Nonce passed to the application when its callback is invoked allowing the application to determine the proper context of operation.

Notification (input/optional)
Callback provided by the application that is called by the service provider module when one of the following occurs: a parallel operation completes, a token running in serial mode surrenders control to the application, or the token is removed (hardware-specific).

Reserved (input)
A reserved input.

Return Value

A module handle for the attached service provider module is returned. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_ModuleDetach

2.6.10 CSSM_ModuleDetach

CSSM_RETURN CSSMAPI CSSM_ModuleDetach (CSSM_MODULE_HANDLE ModuleHandle)

This function detaches the application from the service provider module.

Parameters

ModuleHandle (input)
The handle that describes the service provider module.

Return Value

A CSSM_OK return value signifies that the application has been detached from the service provider module. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_ModuleAttach

2.6.11 CSSM_SetModuleInfo

CSSM_RETURN CSSMAPI CSSM_SetModuleInfo
(const CSSM_GUID_PTR ModuleGUID,
const CSSM_MODULE_INFO_PTR ModuleInfo)

This function replaces all of the currently registered descriptive information about the module identified by *GUID* with the new specified information. CSSM_SetModuleInfo replaces all information for all service categories and all subservices.

To retain any of the module information, use the CSSM_GetModuleInfo function to retrieve the current module information from the KeyWorks registry, make a private copy, and then use the *CSSM_SetModuleInfo* function to update the KeyWorks registry.

This function should be used to incrementally update descriptive information that is unspecified at installation time.

Parameters

ModuleGUID (input)
A pointer to the CSSM_GUID structure containing the GUID for the service provider module.

ModuleInfo (input)
A pointer to the complete structured set of descriptive information about the module.

Return Value

A CSSM_OK return value signifies that the application has been detached from the service provider module. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_GetModuleInfo, CSSM_FreeModuleInfo

2.6.12 CSSM_QueryModulePrivilege

CSSM_RETURN CSSM_QueryModulePrivilege
(const char *AppFileName,
const char *AppPathName,
CSSM_EXEMPTION_MASK *PrivilegeSet)

This function verifies the application's signed manifest credentials and extracts the set of CSSM privileges (exemptions) that it contains.

The application file name and application path name specifies the name and location of the application credentials.

Application may invoke this function to establish whether a module is privileged and what level of privilege the module carries in its manifest credentials.

Parameters

AppFileName (input)
The module file name for which the privilege vector is to be retrieved. This file name is used to locate the module's credentials.

AppPathName (input)
The path to the file that implements the module. This path name is used to locate the module's credentials, authenticate the module, and extract the privilege set, if any.

PrivilegeSet (output)
A bitmask specifying all the privileges that the module has.

Return Value

This function returns CSSM_OK if credential verification was successful and a privilege set was retrieved. On error CSSM_FAIL is returned. Use CSSM_GetError to obtain the error code.

2.7 Utility Functions

2.7.1 CSSM_Free

void CSSMAPI CSSM_Free (void *MemPtr, CSSM_HANDLE AddInHandle)

This function frees the memory allocated by a service provider module.

Parameters

MemPtr (input)
A pointer to the memory to be freed.

AddInHandle (input)
The handle to service provider module that wants to free memory.

Return Value

None

2.7.2 CSSM_FreeList

CSSM_RETURN CSSMAPI CSSM_FreeList (CSSM_LIST_PTR List)

This function frees the memory allocated to hold a list of strings.

Parameters

List (input)
A pointer to the CSSM_LIST structure containing the GUID/name pair of service provider modules.

Return Value

A CSSM_OK return value signifies that the application has been detached from the service provider module. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

2.7.3 CSSM_GetAPIMemoryFunctions

CSSM_API_MEMORY_FUNCS_PTR CSSMAPI CSSM_GetAPIMemoryFunctions
(CSSM_HANDLE AddInHandle)

This function retrieves the memory function table associated with the service provider module.

Parameters

AddInHandle (input)
The handle to the service provider module whose memory function table is being requested.

Return Value

A pointer to the CSSM_API_MEMORY_FUNCS table associated with the service provider module. If an error condition occurred, the function returns NULL. Use CSSM_GetError to obtain the error code.

Chapter 3. IBM KeyWorks Privilege Mechanism

The IBM KeyWorks privilege or exemption mechanism allows the IBM KeyWorks Toolkit to support various modes of operation, providing differing levels of services to different application layer modules. It may be noted that the words *privilege* and *exemption* are used synonymously and interchangeably throughout this document.

Application layer modules using the KeyWorks API may request and be granted special privileges with respect to the KeyWorks framework. The APIs specified in Section 3.2 may be used by an application to request a set of privileges, and to retrieve the current set of privileges. Privileges are granted per application thread, if threads are supported in the operating system environment. Privileges cannot be inherited by spawned processes or spawned or sibling threads. Each process or thread must obtain its own privilege status.

Applications that request and obtain privileges may obtain specialized services that are above and beyond the set of services provided by the KeyWorks framework to non-privileged application layer modules. Privileges are associated with an application module via a set of signed manifest credentials. The set of signed credentials for a privileged application includes a manifest file in which there is a *privilege vecto*r attribute. The value of this attribute describes the privileges for the related application module. At the time an application module is shipped, a determination is made by the development house in liaison with the relevant governmental agencies, regarding the set of privileges that may be granted to the application; the application module is then signed with the appropriate set of privileges.

The KeyWorks framework implements a number of built-in policy checks for controlled functioning of the security services (i.e., compliant with the U.S. export regulations). Applications may request exemption from these built-in checks. If the KeyWorks framework is operating with a set of U.S. domestic Key Recovery Policy Tables (KRPTs), then exemptions are granted "Exemption" if the calling application provides credentials that:

- Are successfully authenticated by the framework (i.e., the credentials pertain to the application module requesting privileges)
- Carry attributes that allow the requested exemptions (i.e., credentials carry the requested privilege attributes)

It may be noted that the privilege mechanism in its full form is relevant only for the exportable versions of the IBM KeyWorks Toolkit. When the KeyWorks Toolkit contains U.S. domestic key recovery policy files, the framework provides its full set of services to all applications and does not discriminate between privileged and non-privileged applications. It is not necessary to request or obtain privileges when working with U.S. domestic versions of the KeyWorks Toolkit. However, in order to maintain compatibility between U.S. domestic and U.S. exportable versions of the KeyWorks Toolkit, the privilege APIs work as expected in both cases. The only difference being that with a U.S. domestic version of the KeyWorks Toolkit, the actual credential files are not checked prior to granting of privileges to a requesting application. An application can request and be granted any set of privileges without the framework checking the application’s credentials.

3.1 Data Structures

3.1.1 CSSM_EXEMPTION_MASK

This data type defines a bit-mask of exemptions or privileges pertaining to the KeyWorks framework. Exemptions are defined to correspond to built-in checks performed by KeyWorks framework and the module managers. The caller must possess the necessary credentials to be granted the exemptions. At this time, the CSSM_EXEMPTION_MASK can hold a maximum of 32 distinct privileges. The mask data type may be changed in the future to allow expansion to support larger sets of privileges.

typedef uint32 CSSM_EXEMPTION_MASK;

```
#define CSSM_EXEMPT_NONE                        0x00    /* no privileges */

#define CSSM_EXEMPT_MULTI_ENCRYPT_CHECK         0x01    /* privilege that allows the caller      */
                                                        /* to perform repeated nested encryption */
                                                        /* of a data buffer */

#define CSSM_STRONG_CRYPTO_WITH_KR              0x02    /* privilege that allows the caller      */
                                                        /* to obtain any strength cryptography as */
                                                        /* long as key recovery operations are   */
                                                        /* performed based on key recovery policy */
                                                        /* tables */

#define CSSM_EXEMPT_LE_KR                       0x04.   /* privilege that allows the caller      */
                                                        /* to obtain any strength cryptography   */
                                                        /* without the need to perform law       */
                                                        /* enforcement key recovery operations.  */

#define CSSM_EXEMPT_ENT_KR                      0x08    /* privilege that allows the caller      */
                                                        /* to obtain any strength cryptography   */
                                                        /* without the need to perform enterprise */
                                                        /* key recovery operations.*/

#define CSSM_EXEMPT_ALL                         0xff    /* privilege that allows the caller      */
                                                        /* to obtain the services corresponding to */
                                                        /* the combination of all the privileges */
                                                        /* defined.*/
```

3.2 Operations

3.2.1 CSSM_CheckCssmExemption

CSSM_RETURN CSSMAPI CSSM_CheckCssmExemption
(CSSM_EXEMPTION_MASK *Exemptions)

This function returns the exemptions possessed by the current thread. If the exemptions returned is non-zero, it implies that the **CSSM_RequestCssmExemption** API had been called to request the specific set of exemptions

Parameters

Exemptions (output)
A bit-mask of all exemptions possessed by the calling thread.

Return Value

A CSSM_OK return value signifies the operation was successful and that the exemption returned is valid. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_ RequestCssmExemption

3.2.2 CSSM_RequestCssmExemption

CSSM_RETURN CSSMAPI CSSM_RequestCssmExemption
(CSSM_EXEMPTION_MASK ExemptionRequest,
const char *AppFileName,
const char *AppPathName,
const void * Reserved)

This function authenticates the application and verifies whether it is authorized to receive the requested exemptions. Authentication is based on successful verification of the application's signed manifest credentials. After the authentication step, the framework ensures that the credentials authorize the application to acquire the requested exemptions. The KeyWorks framework has built-in knowledge of the allowable roots of trust for authenticating application credentials.

The credential verification step is bypassed when the KeyWorks framework is operating with a set of U.S. domestic KRPTs. The requested exemptions are granted automatically in this case, and the *AppFileName* and *AppPathName* parameters may be left as NULL.

The exemption mask defines the requested exemptions. The application filename and application pathname specify the location of the application module, and allow the framework to locate the application's credentials.

Applications may invoke this function multiple times. Each successful verification replaces the previously granted exemptions. Exemptions are not inherited by spawned processes or spawned threads. If the *ExemptionRequest* parameter is zero, all privileges are dropped for that thread.

It may be noted that the *AppFileName* and *AppPathName* parameters may be left as NULL if it is known for sure that the requested exemptions are a subset of the currently possessed exemptions. In such cases, the actual credentials are not checked by the framework.

Parameters

ExemptionRequest (input)
A bit-mask of all exemptions being requested by the caller. If the value is CSSM_EXEMPT_ALL, the caller is requesting all possible privileges that may be granted according to the credentials that are presented and checked.

AppFileName (input)
The name of the file that implements the application (containing its main entry point). This filename is used to locate the application's credentials for purposes of application authentication by the framework. Note that the filename is expected not to have a leading or trailing pathname separator ("/" or "\" depending on the platform of use).

AppPathName (input)
The path to the file that implements the application (containing its main entry point). This pathname is used to locate the application's credentials for purposes of application authentication by the framework. Note that the pathname may be a fully qualified pathname or a relative pathname from the current working directory. In either case, it is expected to contain a trailing pathname separator ("/" or "\" depending on the platform of use).

Reserved (input/optional)
A reserved input.

Return Value

A CSSM_OK return value signifies the verification operation was successful and the exemption has been granted. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_ CheckCssmExemption

Chapter 4. Cryptographic Services API

Cryptographic Service Providers (CSPs) are service provider modules which perform cryptographic operations including encryption, decryption, digital signaturing, key and key pair generation, random number generation (RNG), message digest, key wrapping, key unwrapping, and key exchange. Cryptographic services can be implemented by a hardware-software combination or by software only. Besides the traditional cryptographic functions, CSPs may provide other vendor-specific services. The set of services provided can be dynamic even after a caller has attached the CSP for service. This means the capabilities registered when the CSP was installed can change during execution based on changes internal or external to the system.

The CSP is always responsible for the secure storage of private keys. Optionally, the CSP may assume responsibility for the secure storage of other object types, such as symmetric keys and certificates. The implementation of secured persistent storage for keys can use the services of a Data Storage Library (DL) module within the KeyWorks Framework or some approach internal to the CSP. Accessing persistent objects managed by the CSP, other than keys, is performed using KeyWorks's DL application programming interfaces (APIs).

CSPs optionally support a password-based login sequence. When login is supported, the caller is allowed to change passwords as deemed necessary. This is part of a standard user-initiated maintenance procedure. Some CSPs support operations for privileged CSP administrators. The model for CSP administration varies widely among CSP implementations. For this reason, KeyWorks does not define APIs for vendor-specific CSP administration operations. CSP vendors can makes these services available to CSP administration tools using the CSSM_CSP_Passthrough function.

The range and types of cryptographic services a CSP supports are at the discretion of the vendor. A registry and query mechanism is available through the KeyWorks for CSPs to disclose the services and details about the services. As an example, a CSP may register the following with the KeyWorks: Encryption is supported, algorithms present are Data Encryption Standard (DES) with cipher block chaining for key sizes 40 and 56 bits, and triple DES with three keys for key-size 168 bits.

All cryptographic services requested by applications will be channeled to one of the CSPs through KeyWorks. CSP vendors only need target their modules to KeyWorks for all security-conscious applications to have access to their product.

Calls made to a CSP to perform cryptographic operations occur within a framework called a *session*, which is established and terminated by the application. Applications must create a *session context* (simply referred to as the *context*) prior to starting CSP operations and delete it as soon as possible upon completion of the operation. Context information is not persistent; it is not saved permanently in a file or database.

Before an application calls a CSP to perform a cryptographic operation, the application uses the query services function to determine what CSPs are installed and what services they provide. Based on this information, the application then can determine which CSP to use for subsequent operations; the application creates a session with this CSP and performs the operation.

Depending on the class of cryptographic operations, individualized attributes are available for the cryptographic context. Besides specifying an algorithm when creating the context, the application may also initialize a session key, pass an initialization vector and/or pass padding information to complete the description of the session. A successful return value from the create function indicates the desired CSP is available. Functions are also provided to manage the created context.

When a context is no longer required, the application calls CSSM_DeleteContext. Resources that were allocated for that context can be reclaimed by the operating system.

There are two basic types of cryptographic operations – a single call to perform an operation and a staged method of performing the operation. For the single call method, only one call is needed to obtain the result. For the staged method, there is an initialization call followed by one or more update calls, and ending with a completion (final) call. The result is available after the final function completes its execution for most cryptographic operations – staged encryption/decryption are an exception in that each update call generates a portion of the result.

4.1 Data Structures

4.1.1 CSSM_CALLBACK

typedef CSSM_DATA_PTR (CSSMAPI *CSSM_CALLBACK) (void *allocRef, uint32 ID);

Definitions:

allocRef - Memory heap reference specifying which heap to use for memory allocation.

ID - Input data to identify the callback.

4.1.2 CSSM_CC_HANDLE

typedef uint32 CSSM_CC_HANDLE/* Cryptographic Context Handle */

4.1.3 CSSM_CONTEXT

```
typedef struct cssm_context {
   uint32 ContextType;
   uint32 AlgorithmType;
   uint32 Reserve;
   uint32 NumberOfAttributes;
   CSSM_CONTEXT_ATTRIBUTE_PTR ContextAttributes;
   CSSM_BOOL Privileged;
   uint32 EncryptionProhibited;
   uint32 WorkFactor;
} CSSM_CONTEXT, *CSSM_CONTEXT_PTR
```

Definitions:

ContextType - An identifier describing the type of services for this context, as shown in Table 1.

Table 1. Context Types

Value	Description
CSSM_ALGCLASS_NONE	Null Context type
CSSM_ALGCLASS_CUSTOM	Custom algorithms
CSSM_ALGCLASS_KEYXCH	Key Exchange algorithms
CSSM_ALGCLASS_SIGNATURE	Signature algorithms

Value	Description
CSSM_ALGCLASS_SYMMETRIC	Symmetric Encryption algorithms
CSSM_ALGCLASS_DIGEST	Message Digest algorithms
CSSM_ALGCLASS_RANDOMGEN	Random Number Generation algorithms
CSSM_ALGCLASS_UNIQUEGEN	Unique ID Generation algorithms
CSSM_ALGCLASS_MAC	Message Authentication Code (MAC) algorithms
CSSM_ALGCLASS_ASYMMETRIC	Asymmetric Encryption algorithms
CSSM_ALGCLASS_KEYGEN	Key Generation algorithms
CSSM_ALGCLASS_DERIVEKEY	Key Derivation algorithms
CSSM_ALGCLASS_KEY_RECOVERY_ENABLEMENT	Key Recovery Enablement algorithms
CSSM_ALGCLASS_KEY_RECOVERY_REGISTRATION	Key Recovery Registration algorithms
CSSM_ALGCLASS_KEY_RECOVERY_REQUEST	Key Recovery Request algorithms

AlgorithmType - An ID number describing the algorithm to be used (see Table 2).

Table 2. Algorithms for a Session Context

Value	Description
CSSM_ALGID_NONE	Null algorithm
CSSM_ALGID_CUSTOM	Custom algorithm
CSSM_ALGID_DH	Diffie-Hellman key exchange algorithm
CSSM_ALGID_PH	Pohlig-Hellman key exchange algorithm
CSSM_ALGID_KEA	Key Exchange algorithm
CSSM_ALGID_MD2	MD2hash algorithm
CSSM_ALGID_MD4	MD4hash algorithm
CSSM_ALGID_MD5	MD5hash algorithm
CSSM_ALGID_SHA1	Secure Hash algorithm
CSSM_ALGID_NHASH	N-Hash algorithm
CSSM_ALGID_HAVAL	HAVAL hash algorithm (MD5 variant)
CSSM_ALGID_RIPEMD	RIPE-MD hash algorithm (MD4 variant - developed for the European Community's RIPE project)
CSSM_ALGID_IBCHASH	IBC-Hash (keyed hash algorithm or MAC)
CSSM_ALGID_RIPEMAC	RIPE-MAC
CSSM_ALGID_DES	Data Encryption Standard block cipher

Value	Description
CSSM_ALGID_DESX	DESX block cipher (DES variant from RSA)
CSSM_ALGID_RDES	RDES block cipher (DES variant)
CSSM_ALGID_3DES_3KEY	Triple-DES block cipher (with 3 keys)
CSSM_ALGID_3DES_2KEY	Triple-DES block cipher (with 2 keys)
CSSM_ALGID_3DES_1KEY	Triple-DES block cipher (with 1 key)
CSSM_ALGID_IDEA	International Data Encryption Algorithm (IDEA) block cipher
CSSM_ALGID_RC2	RC2 block cipher
CSSM_ALGID_RC5	RC5 block cipher
CSSM_ALGID_RC4	RC4 stream cipher
CSSM_ALGID_SEAL	SEAL stream cipher
CSSM_ALGID_CAST	CAST block cipher
CSSM_ALGID_BLOWFISH	BLOWFISH block cipher
CSSM_ALGID_SKIPJACK	Skipjack block cipher
CSSM_ALGID_LUCIFER	Lucifer block cipher
CSSM_ALGID_MADRYGA	Madryga block cipher
CSSM_ALGID_FEAL	FEAL block cipher
CSSM_ALGID_REDOC	REDOC 2 block cipher
CSSM_ALGID_REDOC3	REDOC 3 block cipher
CSSM_ALGID_LOKI	LOKI block cipher
CSSM_ALGID_KHUFU	KHUFU block cipher
CSSM_ALGID_KHAFRE	KHAFRE block cipher
CSSM_ALGID_MMB	MMB block cipher (IDEA variant)
CSSM_ALGID_GOST	GOST block cipher
CSSM_ALGID_SAFER	SAFER K-40, K-64, K-128 block cipher
CSSM_ALGID_CRAB	CRAB block cipher
CSSM_ALGID_RSA	RSA public key cipher
CSSM_ALGID_DSA	Digital Signature algorithm
CSSM_ALGID_MD5WithRSA	MD5/RSA signature algorithm
CSSM_ALGID_MD2WithRSA	MD2/RSA signature algorithm
CSSM_ALGID_ElGamal	ElGamal signature algorithm
CSSM_ALGID_MD2Random	MD2-based random numbers
CSSM_ALGID_MD5Random	MD5-based random numbers
CSSM_ALGID_SHARandom	SHA-based random numbers

Value	Description
CSSM_ALGID_DESRandom	DES-based random numbers
CSSM_ALGID_SHA1WithRSA	SHA-1/RSA signature algorithm
CSSM_ALGID_RSA_PKCS	RSA as specified in PKCS#1
CSSM_ALGID_RSA_ISO9796	RSA as specified in International Organization for Standardization (ISO) 9796
CSSM_ALGID_RSA_RAW	Raw RSA as assumed in X.509
CSSM_ALGID_CDMF	CDMF block cipher
CSSM_ALGID_CAST3	Entrust's CAST3 block cipher
CSSM_ALGID_CAST5	Entrust's CAST5 block cipher
CSSM_ALGID_GenericSecret	Generic secret operations
CSSM_ALGID_ConcatBaseAndKey	Concatenate two keys, base key first
CSSM_ALGID_ConcatKeyAndBase	Concatenate two keys, base key last
CSSM_ALGID_ConcatBaseAndData	Concatenate base key and random data, key first
CSSM_ALGID_ConcatDataAndBase	Concatenate base key and data, data first
CSSM_ALGID_XORBaseAndData	XOR a byte string with the base key
CSSM_ALGID_ExtractFromKey	Extract a key from base key, starting at arbitrary bit position
CSSM_ALGID_SSL3PreMasterGen	Generate a 48-byte SSL 3 premaster key
CSSM_ALGID_SSL3MasterDerive	Derive an SSL 3 key from a premaster key
CSSM_ALGID_SSL3KeyAndMacDerive	Derive the keys and MACing keys for the SSL cipher suite
CSSM_ALGID_SSL3MD5_MAC	Performs SSL 3 MD5 MACing
CSSM_ALGID_SSL3SHA1_MAC	Performs SSL 3 SHA-1 MACing
CSSM_ALGID_MD5Derive	Generate key by MD5 hashing a base key
CSSM_ALGID_MD2Derive	Generate key by MD2 hashing a base key
CSSM_ALGID_SHA1Derive	Generate key by SHA-1 hashing a base key
CSSM_ALGID_WrapLynks	Spyrus LYNKS DES based wrapping scheme w/checksum
CSSM_ALGID_WrapSET_OAEP	SET key wrapping
CSSM_ALGID_BATON	Fortezza BATON cipher
CSSM_ALGID_ECDSA	Elliptic Curve DSA
CSSM_ALGID_MAYFLY	Fortezza MAYFLY cipher
CSSM_ALGID_JUNIPER	Fortezza JUNIPER cipher
CSSM_ALGID_FASTHASH	Fortezza FASTHASH
CSSM_ALGID_3DES	Generic 3DES

Value	Description
CSSM_ALGID_SSL3MD5	SSL3MD5
CSSM_ALGID_SSL3SHA1	SSL3SHA1
CSSM_ALGID_FortezzaTimestamp	FortezzaTimestamp
CSSM_ALGID_SHA1WithDSA	SHA1WithDSA
CSSM_ALGID_SHA1WithECDSA	SHA1WithECDSA
CSSM_ALGID_DSA_BSAFE	BSAFE Key format

Some of the algorithms above in Table 2 operate in a variety of modes. The desired mode is specified using an attribute of type CSSM_ATTRIBUTE_MODE. The valid values for the mode attribute are as follows in Table 3.

Table 3. Modes of Algorithms

Value	Description
CSSM_ALGMODE_NONE	Null algorithm mode
CSSM_ALGMODE_CUSTOM	Custom mode
CSSM_ALGMODE_ECB	Electronic Code Book
CSSM_ALGMODE_ECBPad	ECB with padding
CSSM_ALGMODE_CBC	Cipher Block Chaining
CSSM_ALGMODE_CBC_IV8	CBC with Initialization Vector of 8 bytes
CSSM_ALGMODE_CBCPadIV8	CBC with padding and Initialization Vector of 8 bytes
CSSM_ALGMODE_CFB	Cipher FeedBack
CSSM_ALGMODE_CFB_IV8	CFB with Initialization Vector of 8 bytes
CSSM_ALGMODE_CFBPadIV8	CFB with Initialization Vector of 8 bytes and padding
CSSM_ALGMODE_OFB	Output FeedBack
CSSM_ALGMODE_OFB_IV8	OFB with Initialization Vector of 8 bytes
CSSM_ALGMODE_OFBPadIV8	OFB with Initialization Vector of 8 bytes and padding
CSSM_ALGMODE_COUNTER	Counter
CSSM_ALGMODE_BC	Block Chaining
CSSM_ALGMODE_PCBC	Propagating CBC
CSSM_ALGMODE_CBCC	CBC with Checksum
CSSM_ALGMODE_OFBNLF	OFB with Nonlinear Function
CSSM_ALGMODE_PBC	Plaintext Block Chaining
CSSM_ALGMODE_PFB	Plaintext FeedBack

Value	Description
CSSM_ALGMODE_CBCPD	CBC of Plaintext Difference
CSSM_ALGMODE_PUBLIC_KEY	Use the public key
CSSM_ALGMODE_PRIVATE_KEY	Use the private key
CSSM_ALGMODE_SHUFFLE	Fortezza shuffle mode
CSSM_ALGMODE_ECB64	Electronic Code Book (64 bits)
CSSM_ALGMODE_CBC64	Cipher Block Chaining (64 bits)
CSSM_ALGMODE_OFB64	Output FeedBack (64 bits)
CSSM_ALGMODE_CFB64	Cipher FeedBack (64 bits)
CSSM_ALGMODE_CFB32	Cipher FeedBack (32 bits)
CSSM_ALGMODE_CFB16	Cipher FeedBack (16 bits)
CSSM_ALGMODE_CFB8	Cipher FeedBack (8 bits)
CSSM_ALGMODE_WRAP	SKIPJACK Wrap mechanism
CSSM_ALGMODE_PRIVATE_WRAP	SKIPJACK Private Wrap mechanism
CSSM_ALGMODE_RELAYX	SKIPJACK RELAYX mechanism
CSSM_ALGMODE_ECB128	Electronic Code Book (128 bits)
CSSM_ALGMODE_ECB96	Electronic Code Book (96 bits)
CSSM_ALGMODE_CBC128	Cipher Block Chaining (128 bits)
CSSM_ALGMODE_OAEP_HASH	Optimal Asymmetric Encryption Padding (OAEP) for RSA

NumberOfAttributes - Number of attributes associated with this service.

ContextAttributes - Pointer to data that describes the attributes. To retrieve the next attribute, advance the attribute pointer.

Privileged - When this flag is CSSM_TRUE, the context can perform cryptographic operations without being forced to follow the key recovery policy.

EncryptionProhibited - An integer indicating whether encryption is allowed. If encryption is allowed, this field is zero. Otherwise, the flags indicate which policy disallowed encryption.

WorkFactor - WorkFactor is the maximum number of bits that can be left out of Key Recovery Fields (KRFs) when they are generated. The recovered of the key must then search this number of bits to recover the key.

4.1.4 CSSM_CONTEXT_ATTRIBUTE

```
typedef struct cssm_context_attribute{
   uint32 AttributeType;
   uint32 AttributeLength;
   union {
      char *String;
```

```
        uint32 Uint32;
        CSSM_CRYPTO_DATA_PTR Crypto;
        CSSM_KEY_PTR Key;
        CSSM_DATA_PTR Data;
        CSSM_DATE_PTR Date;
        CSSM_RANGE_PTR Range;
        CSSM_VERSION_PTR Version;
        CSSM_KR_PROFILE_PTR KRProfile;
    } Attribute;
} CSSM_CONTEXT_ATTRIBUTE, *CSSM_CONTEXT_ATTRIBUTE_PTR;
```

Definitions:

AttributeType - An identifier describing the type of attribute. Valid attribute types are as follows in Table 4.

Table 4. Attribute Types

Value	Description	Data Type
CSSM_ATTRIBUTE_NONE	No attribute	None
CSSM_ATTRIBUTE_CUSTOM	Custom data	Opaque pointer
CSSM_ATTRIBUTE_DESCRIPTION	Description of attribute	String
CSSM_ATTRIBUTE_KEY	Key Data	CSSM_KEY
CSSM_ATTRIBUTE_INIT_VECTOR	Initialization vector	CSSM_DATA
CSSM_ATTRIBUTE_SALT	Salt	CSSM_DATA
CSSM_ATTRIBUTE_PADDING	Padding information	uint32
CSSM_ATTRIBUTE_RANDOM	Random data	CSSM_DATA
CSSM_ATTRIBUTE_SEED	Seed	CSSM_CRYPTO_DATA
CSSM_ATTRIBUTE_PASSPHRASE	Passphrase	CSSM_CRYPTO_DATA
CSSM_ATTRIBUTE_KEY_LENGTH	Key length specified in bits	uint32
CSSM_ATTRIBUTE_KEY_LENGTH_RANGE	Key length range specified in bits	CSSM_RANGE
CSSM_ATTRIBUTE_BLOCK_SIZE	Block size	uint32
CSSM_ATTRIBUTE_OUTPUT_SIZE	Output size	uint32
CSSM_ATTRIBUTE_ROUNDS	Number of runs or rounds	uint32
CSSM_ATTRIBUTE_IV_SIZE	Size of initialization vector	uint32
CSSM_ATTRIBUTE_ALG_PARAMS	Algorithm parameters	CSSM_DATA

Value	Description	Data Type
CSSM_ATTRIBUTE_LABEL	Label placed on an object when it is created	CSSM_DATA
CSSM_ATTRIBUTE_KEY_TYPE	Type of key to generate or derive	uint32
CSSM_ATTRIBUTE_MODE	Algorithm mode to use for encryption	uint32
CSSM_ATTRIBUTE_EFFECTIVE_BITS	Number of effective bits used in the RC2 cipher	uint32
CSSM_ATTRIBUTE_START_DATE	Starting date for an object's validity	CSSM_DATE
CSSM_ATTRIBUTE_END_DATE	Ending date for an object's validity	CSSM_DATE
CSSM_ATTRIBUTE_KEYUSAGE	Key usage	uint32
CSSM_ATTRIBUTE_KEYATTR	Key attributes	uint32
CSSM_ATTRIBUTE_VERSION	Object version	CSSM_VERSION
CSSM_ATTRIBUTE_ALG_ID	Algorithm ID	uint32
CSSM_ATTRIBUTE_ITERATION_COUNT	Number of iterations	uint32
CSSM_ATTRIBUTE_ROUNDS_RANGE	Minimum and maximum number of rounds	CSSM_RANGE
CSSM_ATTRIBUTE_KRPROFILE_LOCAL	Key Recovery Profile for the local user	CSSM_KR_PROFILE
CSSM_ATTRIBUTE_KRPROFILE_REMOTE	Key Recovery Profile for the remote user	CSSM_KR_PROFILE

The data referenced by a CSSM_ATTRIBUTE_CUSTOM attribute must be a single continuous memory block. This allows the KeyWorks to appropriately release all dynamically allocated memory resources.

AttributeLength - Length of the attribute data.

Attribute - Union representing the attribute data. The union member used is named after the type of data contained in the attribute. See Table 4 for the data types associated with each attribute type.

4.1.5 CSSM_CONTEXT_INFO

typedef CSSM_CONTEXT CSSM_CONTEXT_INFO

4.1.6 CSSM_CRYPTO_DATA

```
typedef struct cssm_crypto_data {
      CSSM_DATA_PTR Param;
      CSSM_CALLBACK Callback;
      uint32 CallbackID;
}CSSM_CRYPTO_DATA, *CSSM_CRYPTO_DATA_PTR
```

Definitions:

Param - A pointer to the parameter data and its size in bytes.

Callback - An optional callback routine for the service provider modules to obtain the parameter.

ID - A tag that identifies the callback.

4.1.7 CSSM_CSP_CAPABILITY

```
typedef CSSM_CONTEXT CSSM_CSP_CAPABILITY, *CSSM_CSP_CAPABILITY_PTR;
```

4.1.8 CSSM_CSP_FLAGS

```
typedef uint32 CSSM_CSP_FLAGS;
```

4.1.9 CSSM_CSP_HANDLE

The CSSM_CSP_HANDLE is used to identify the association between an application thread and an instance of a CSP module. It is assigned when an application causes KeyWorks to attach to a CSP. It is freed when an application causes KeyWorks to detach from a CSP. The application uses the CSSM_CSP_HANDLE with every CSP function call to identify the targeted CSP. The CSP uses the CSSM_CSP_HANDLE to identify the appropriate application's memory management routines when allocating memory on the application's behalf.

```
typedef uint32 CSSM_CSP_HANDLE/* Cryptographic Service Provider Handle */
```

4.1.10 CSSM_CSP_SESSION_TYPE

The CSSM_CSP_SESSION_TYPE is provided in Table 5.

Table 5. Session Types

Value		**Description**
CSSM_CSP_SESSION_EXCLUSIVE	0 x 0001	Single user CSP.
CSSM_CSP_SESSION_READWRITE	0 x 0002	Caller can read and write objects such as keys in the CSP.
CSSM_CSP_SESSION_SERIAL	0 x 0004	Multiuser, reentrant CSP that requires serial access.

4.1.11 CSSM_CSPSUBSERVICE

Three structures are used to contain all of the static information that describes a CSP module: cssm_moduleinfo, cssm_serviceinfo, and cssm_cspsubservice. This descriptive information is securely stored in the KeyWorks registry when the CSP module is installed with CSSM. A CSP module may implement multiple types of services and organize them as subservices.

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo and specifying the CSP module Globally Unique ID (GUID).

```
typedef struct cssm_cspsubservice {
   uint32 SubServiceId;
   CSSM_STRING Description;
   CSSM_CSP_FLAGS CspFlags;
   uint32 CspCustomFlags;
   uint32 AccessFlags;
   CSSM_CSPTYPE CspType;
   union {
      CSSM_SOFTWARE_CSPSUBSERVICE_INFO SoftwareCspSubService;
      CSSM_HARDWARE_CSPSUBSERVICE_INFO HardwareCspSubService;
   };
   CSSM_CSP_WRAPPEDPRODUCT_INFO WrappedProduct;
} CSSM_CSPSUBSERVICE, *CSSM_CSPSUBSERVICE_PTR;
```

Definitions:

SubServiceId - The subservice ID required for an attach call to connect a CSP to an individual subservice within a CSP.

Description - A NULL-terminated character string containing a text description of the subservice.

CspFlags - A bit-mask containing general flags defined by KeyWorks for CSPs. The mask may contain one or a combination of the following in Table 6.

Table 6. CSP Flags

CSSM_CSP_FLAGS Values	Description
CSSM_CSP_STORES_PRIVATE_KEYS	CSP can store private keys.
CSSM_CSP_STORES_PUBLIC_KEYS	CSP can store public keys.
CSSM_CSP_STORES_SESSION_KEYS	CSP can store session/secret keys.

CspCustomFlags - Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

AccessFlags - Flags that are required to be provided by the application during an attach call when specifying the subservice ID given in *SubServiceId*.

CspType - Identifier that determines the type of CSP information structure referenced by *CspInfo*. The following values and their corresponding CSP information structures are currently defined in Table 7.

Table 7. CSP Information Type Identifiers and Associated Structure Types

CSP Information Structure Identifier	Structure Type
CSSM_CSP_TYPE_SOFTWARE	CSSM_CSP_TYPE_SOFTWARE_INFO
CSSM_CSP_TYPE_PKCS11	CSSM_CSP_TYPE_PKCS11_INFO

SoftwareCspSubService/HardwareCspSubService - A CSP information structure of the type specified by *CspType*.

WrappedProduct - Pointer to a CSSM_CSP_WRAPPEDPRODUCTINFO structure describing a product that is wrapped by the CSP.

4.1.12 CSSM_CSPTYPE

```
typedef uint32 CSSM_CSPTYPE;
#define CSSM_CSP_SOFTWARE 1
#define CSSM_CSP_HARDWARE 2
```

4.1.13 CSSM_CSP_WRAPPEDPRODUCTINFO

```
typedef struct cssm_csp_wrappedproductinfo {
   CSSM_VERSION StandardVersion;
   CSSM_STRING StandardDescription;
   CSSM_VERSION ProductVersion;
   CSSM_STRING ProductDescription;
   CSSM_STRING ProductVendor;
   uint32 ProductFlags;
} CSSM_CSP_WRAPPEDPRODUCTINFO, *CSSM_CSP_WRAPPEDPRODUCTINFO_PTR;
```

Definitions:

StandardVersion - Version of the standard to which the wrapped product complies.

StandardDescription - A NULL-terminated character string containing a text description of the standard to which the wrapped product complies.

ProductVersion - Version of the product wrapped by the CSP.

ProductDescription - A NULL-terminated character string containing a text description of the product wrapped by the CSP.

ProductVendor - A NULL-terminated character string containing the name of the wrapped product's vendor.

ProductFlags - This version of KeyWorks has no flags defined. This field must be set to zero.

4.1.14 CSSM_DATA

The CSSM_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory. This memory must be allocated and freed using the memory management routines provided by the calling application via KeyWorks.

```
typedef struct cssm_data{
   uint32 Length;/* in bytes */
   uint8 *Data;
} CSSM_DATA, *CSSM_DATA_PTR
```

Definitions:

Length - Length of the data buffer in bytes.

Data - Points to the start of an arbitrary length data buffer.

4.1.15 CSSM_DATE

```
typedef struct cssm_date {
   uint8 Year[4];
   uint8 Month[2];
   uint8 Day[2];
} CSSM_DATE, *CSSM_DATE_PTR
```

Definitions:

Year – Four-digit ASCII representation of the year.

Month – Two-digit representation of the month.

Day – Two-digit representation of the day.

4.1.16 CSSM_HARDWARECSPSUBSERVICEINFO

```
typedef struct cssm_hardwarecspsubserviceinfo {
   uint32 NumberOfCapabilities;
   CSSM_CSP_CAPABILITY_PTR CapabilityList;
   void * Reserved;

   /* Reader/Slot Info */
   CSSM_STRING ReaderDescription;
   CSSM_STRING ReaderVendor;
   CSSM_STRING ReaderSerialNumber;
   CSSM_VERSION ReaderHardwareVersion;
   CSSM_VERSION ReaderFirmwareVersion;
   uint32 ReaderFlags;
   uint32 ReaderCustomFlags;

   CSSM_STRING TokenDescription;
   CSSM_STRING TokenVendor;
   CSSM_STRING TokenSerialNumber;
   CSSM_VERSION TokenHardwareVersion;
   CSSM_VERSION TokenFirmwareVersion;

   uint32 TokenFlags;
   uint32 TokenCustomFlags;
   uint32 TokenMaxSessionCount;
   uint32 TokenOpenedSessionCount;
```

```
   uint32 TokenMaxRWSessionCount;
   uint32 TokenOpenedRWSessionCount;
   uint32 TokenTotalPublicMem;
   uint32 TokenFreePublicMem;
   uint32 TokenTotalPrivateMem;
   uint32 TokenFreePrivateMem;
   uint32 TokenMaxPinLen;
   uint32 TokenMinPinLen;
   char TokenUTCTime[16];

   char *UserLabel;
   CSSM_DATA UserCACertificate;
} CSSM_HARDWARE_CSPSUBSERVICE_INFO,
*CSSM_HARDWARE_CSPSUBSERVICE_INFO_PTR;
```

Definitions:

NumberOfCapabilities - Number of capabilities in list.

CapabilityList - A context list that specifies the capabilities of the CSP.

Reserved - This field is reserved for future use and must always be set to NULL.

ReaderDescription - A NULL-terminated character string containing a description of the device reader.

ReaderVendor - A NULL-terminated string that contains the name of the reader vendor.

ReaderSerialNumber - A NULL-terminated string that contains the serial number of the reader.

ReaderHardwareVersion - Hardware version of the reader.
ReaderFirmwareVersion - Firmware version of the reader.

ReaderFlags - Bit-mask containing information about the reader. The flags specified in the mask are as follows in Table 8.

Table 8. PKCS#11 CSP Reader Flags

Reader Flag	**Description**
CSSM_CSP_RDR_TOKENPRESENT	Token is present in the reader.
CSSM_CSP_RDR_REMOVABLE	Reader supports removable tokens.
CSSM_CSP_RDR_HW	Reader is a hardware device.

ReaderCustomFlags - Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

The following fields may not be valid if the CSSM_CSP_RDR_TOKENPRESENT flag is not set in the *ReaderFlags* field. Unknown string and CSSM_DATA fields will be set to NULL, integer and date fields will be set to zero and, flag fields will have all flags set to false.

TokenDescription - A NULL-terminated character string that contains a text description of the token. This value may be NULL or equal to *ReaderDescription* if the token is not removable.

TokenVendor - A NULL-terminated string that contains the name of the token vendor. This value may be NULL or equal to *ReaderVendor* if the token is not removable.

TokenSerialNumber - A NULL-terminated string that contains the serial number of the token. This value may be NULL or equal to *ReaderSerialNumber* if the token is not removable.

TokenHardwareVersion - Hardware version of the token.

TokenFirmwareVersion - Firmware version of the token.

TokenFlags - Bit-mask containing information about the token. The flags specified in the mask are provided in Table 9.

Table 9. PKCS#11 CSP Token Flags

Token Flags	**Description**
CSSM_CSP_TOK_RNG	Token has random number generator.
CSSM_CSP_TOK_WRITE_PROTECTED	Token is write-protected.
CSSM_CSP_TOK_LOGIN_REQUIRED	User must login to access private objects.
CSSM_CSP_TOK_USER_PIN_INITIALIZED	User's PIN has been initialized.
CSSM_CSP_TOK_EXCLUSIVE_SESSION	An exclusive session currently exists.
CSSM_CSP_TOK_CLOCK_EXISTS	Token has built-in clock.
CSSM_CSP_TOK_ASYNC_SESSION	Token supports asynchronous operations.
CSSM_CSP_TOK_PROT_AUTHENTICATION	Token has protected authentication path.
CSSM_CSP_TOK_DUAL_CRYPTO_OPS	Token supports dual cryptographic operations.

TokenCustomFlags - Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

TokenMaxSessionCount - Maximum number of CSP handles referencing the token that may exist simultaneously.
TokenOpenedSessionCount - Number of CSP handles referencing the token that currently exists.

TokenTotalPublicMem - Amount of public storage space in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE if the CSP does not want to expose this information.

TokenFreePublicMem - Amount of public storage space available for use in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE (-1) if the CSP does not want to expose this information.

TokenTotalPrivateMem - Amount of private storage space in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE (-1) if the CSP does not want to expose this information.
TokenFreePrivateMem - Amount of private storage space available for use in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE if the CSP does not want to expose this information.

TokenMaxPinLen - Maximum length of passwords that can be used for authentication to the CSP.

TokenMinPinLen - Minimum length of passwords that can be used for authentication to the CSP.

TokenUTCTime - Character array containing the current Coordinated Universal Time (UTC) value in the CSP. The value is valid if the CSSM_CSP_TOK_CLOCK_EXISTS flag is true. The time is represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for month, day, hour, minute, and second; and 2 additional reserved '0' characters).

UserLabel - A NULL-terminated string containing the label of the token.

UserCACertificate - Certificate of the Certificate Authority (CA).

4.1.17 CSSM_HEADERVERSION

This data structure represents the version number of a key header structure. This version number is an integer that increments with each format revision of CSSM_KEYHEADER. The current revision number is represented by CSSM_KEYHEADER_VERSION, which equals 2 in this release of KeyWorks.

```
typedef uint32 CSSM_HEADERVERSION

#define CSSM_KEYHEADER_VERSION (2)
```

4.1.18 CSSM_KEY

This structure is used to represent keys in KeyWorks.

```
typedef struct cssm_key{
   CSSM_KEYHEADER KeyHeader;
   CSSM_DATA KeyData;
} CSSM_KEY, *CSSM_KEY_PTR;

typedef CSSM_KEY CSSM_WRAP_KEY, *CSSM_WRAP_KEY_PTR;
```

Definitions:

KeyHeader - Header describing the key, fixed length.

KeyData – Data representation of the key, variable length.

4.1.19 CSSM_KEYHEADER

The key header contains meta-data about a key. It contains information used by a CSP or application when using the associated key data. The service provider module is responsible for setting the appropriate values.

```
typedef struct cssm_keyheader {
   CSSM_HEADERVERSION HeaderVersion;
   CSSM_GUID CspId;
   uint32 BlobType;
   uint32 Format;
   uint32 AlgorithmId;
```

```
    uint32 KeyClass;
    uint32 KeySizeInBits;
    uint32 KeyAttr;
    uint32 KeyUsage;
    CSSM_DATE StartDate;
    CSSM_DATE EndDate;
    uint32 WrapAlgorithmId;
    uint32 WrapMode;
    uint32 Reserved;
} CSSM_KEYHEADER, *CSSM_KEYHEADER_PTR;
```

Definitions:

HeaderVersion - This is the version of the key header structure.

CspId - If known, the GUID of the CSP that generated the key. This value will not be known if a key is received from a third party, or extracted from a certificate.

BlobType - Describes the basic format of the key data. It can be any one of the following values in Table 10.

Table 10. Keyblob Type Identifiers

Identifier	Description
CSSM_KEYBLOB_RAW	The blob is a clear, raw key.
CSSM_KEYBLOB_RAW_BERDER	The blob is a clear key, DER-encoded.
CSSM_KEYBLOB_REFERENCE	The blob is a reference to a key.
CSSM_KEYBLOB_WRAPPED	The blob is a wrapped RAW key.
CSSM_KEYBLOB_WRAPPED_BERDER	The blob is a wrapped DER-encoded key.
CSSM_KEYBLOB_OTHER	The blob is a wrapped DER-encoded key.

Format - Describes the detailed format of the key data based on the value of the *BlobType* field. If the blob type has a non-reference basic type, then a CSSM_KEYBLOB_RAW_FORMAT identifier must be used, otherwise a CSSM_KEYBLOB_REF_FORMAT identifier is used. Any of the following values are valid as format identifiers in Table 11.

Table 11. Keyblob Format Identifiers

Keyblob Format Identifier	Description
CSSM_KEYBLOB_RAW_FORMAT_NONE	No further conversion needs to be done.
CSSM_KEYBLOB_RAW_FORMAT_PKCS1	RSA PKCS1 V1.5
CSSM_KEYBLOB_RAW_FORMAT_PKCS3	RSA PKCS3 V1.5
CSSM_KEYBLOB_RAW_FORMAT_MSCAPI	Microsoft CAPI V2.0
CSSM_KEYBLOB_RAW_FORMAT_PGP	PGP

Keyblob Format Identifier	Description
CSSM_KEYBLOB_RAW_FORMAT_FIPS186	U.S. Gov. FIPS 186 - DSS V
CSSM_KEYBLOB_RAW_FORMAT_BSAFE	RSA BSAFE V3.0
CSSM_KEYBLOB_RAW_FORMAT_PKCS11	RSA PKCS11 V2.0
CSSM_KEYBLOB_RAW_FORMAT_CDSA	Intel CDSA
CSSM_KEYBLOB_RAW_FORMAT_OTHER	Other, CSP defined.
CSSM_KEYBLOB_REF_FORMAT_INTEGER	Reference is a number or handle.
CSSM_KEYBLOB_REF_FORMAT_STRING	Reference is a string or name.
CSSM_KEYBLOB_REF_FORMAT_OTHER	Other, CSP defined.

AlgorithmId - The algorithm for which the key was generated. This value does not change when the key is wrapped. Any of the defined KeyWorks algorithm IDs may be used.

KeyClass - Class of key contained in the keyblob. Valid key classes are as follows in Table 12.

Table 12. Key Class Identifiers

Key Class Identifier	Description
CSSM_KEYCLASS_PUBLIC_KEY	Key is a public key.
CSSM_KEYCLASS_PRIVATE_KEY	Key is a private key.
CSSM_KEYCLASS_SESSION_KEY	Key is a session or symmetric key.
CSSM_KEYCLASS_SECRET_PART	Key is part of secret key.
CSSM_KEYCLASS_OTHER	Other

KeySizeInBits - This is the logical size of the key in bits. The logical size is the value referred to when describing the length of the key. For instance, an RSA key would be described by the size of its modulus and a DSA key would be represented by the size of its prime. Symmetric key sizes describe the actual number of bits in the key. For example, DES keys would be 64 bits and an RC4 key could range from 1 to 128 bits.

KeyAttr - Attributes of the key represented by the data. These attributes are used by CSPs to convey information about stored or referenced keys. The attributes are represented as a bit-mask (see Table 13).

Table 13. Key Attribute Flags

Attribute	Description
CSSM_KEYATTR_PERMANENT	Key is stored persistently in the CSP, i.e., PKCS#11 token object.
CSSM_KEYATTR_PRIVATE	Key is a private object and protected by either a user login, a password, or both.
CSSM_KEYATTR_MODIFIABLE	The key or its attributes can be modified.

Attribute	Description
CSSM_KEYATTR_SENSITIVE	Key is sensitive. It may only be extracted from the CSP in a wrapped state. It will always be false for raw keys.
CSSM_KEYATTR_ALWAYS_SENSITIVE	Key has always been sensitive. It will always be false for raw keys.
CSSM_KEYATTR_EXTRACTABLE	Key is extractable from the CSP. If this bit is not set, the key is either not stored in the CSP or cannot be extracted from the CSP under any circumstances. It will always be false for raw keys.
CSSM_KEYATTR_NEVER_EXTRACTABLE	Key has never been extractable. It will always be false for raw keys.

KeyUsage - A bit-mask representing the valid uses of the key. Any of the following values are valid in Table 14.

Table 14. Key Usage Flags

Usage Mask	Description
CSSM_KEYUSE_ANY	Key may be used for any purpose supported by the algorithm.
CSSM_KEYUSE_ENCRYPT	Key may be used for encryption.
CSSM_KEYUSE_DECRYPT	Key may be used for decryption.
CSSM_KEYUSE_SIGN	Key can be used to generate signatures. For symmetric keys this represents the ability to generate MACs.
CSSM_KEYUSE_VERIFY	Key can be used to verify signatures. For symmetric keys this represents the ability to verify MACs.
CSSM_KEYUSE_SIGN_RECOVER	Key can be used to perform signatures with message recovery. This form of a signature is generated using the CSSM_EncryptData API with the algorithm mode set to CSSM_ALGMODE_PRIVATE_KEY. This attribute is only valid for asymmetric algorithms.
CSSM_KEYUSE_VERIFY_RECOVER	Key can be used to verify signatures with message recovery. This form of a signature is verified using the CSSM_DecryptData API with the algorithm mode set to CSSM_ALGMODE_PRIVATE_KEY. This attribute is only valid for asymmetric algorithms.
CSSM_KEYUSE_WRAP	Key can be used to wrap another key.
CSSM_KEYUSE_UNWRAP	Key can be used to unwrap a key.
CSSM_KEYUSE_DERIVE	Key can be used as the source for deriving other keys.

StartDate - Date from which the corresponding key is valid. All fields of the CSSM_DATA structure will be set to zero if the date is unspecified or unknown. This date is not enforced by the CSP.

EndDate - Data that the key expires and can no longer be used. All fields of the CSSM_DATA structure will be set to zero is the date if unspecified or unknown. This date is not enforced by the CSP.

WrapAlgorithmId - If the key data contains a wrapped key, this field contains the algorithm used to create the wrapped blob. This field will be set to CSSM_ALGID_NONE if the key is not wrapped.

WrapMode - If the wrapping algorithm supports multiple wrapping modes, this field contains the mode used to wrap the key. This field is ignored if the *WrapAlgorithmId* is CSSM_ALGID_NONE.

Reserved - This field is reserved for future use. It should always be set to zero.

4.1.20 CSSM_KEY_SIZE

This structure holds the physical key size and the effective key size for a given key. The metric used is bits. The number of effective bits is the number of key bits that can be used in a cryptographic operation compared with the number of bits that may be present in the key. When the number of effective bits is less than the number of actual bits, this is known as "dumbing down."

```
typedef struct cssm_key_size {
   uint32 KeySizeInBits;/* Key size in bits */
   uint32 EffectiveKeySizeInBits; /* Effective key size in bits */
} CSSM_KEYSIZE, *CSSM_KEYSIZE_PTR
```

Definitions:

KeySizeInBits - The actual number of bits in a key.

EffectiveKeySizeInBits - The number of key bits that can be used for cryptographic operations.

4.1.21 CSSM_KEY_TYPE

```
typedef uint32 CSSM_KEY_TYPE, *CSSM_KEY_TYPE_PTR;
```

4.1.22 CSSM_NOTIFY_CALLBACK

This data structure defines a pointer to a function that applications can use to invoke an application-supplied function.

```
typedef CSSM_RETURN (CSSMAPI *CSSM_NOTIFY_CALLBACK)
   (CSSM_MODULE_HANDLE ModuleHandle,
      uint32 Application,
      uint32 Reason,
      void * Param)
```

Definitions:

ModuleHandle - Handle of the module to which the notification applies.

Application – Application-specific context indicator. This value is specified when a service provider module is attached.

Reason - One of the values is specified below in Table 15.

Table 15. Reasons

Reason	Value
CSSM_NOTIFY_SURRENDER	0
CSSM_NOTIFY_COMPLETE	1
CSSM_NOTIFY_DEVICE_REMOVED	2
CSSM_NOTIFY_DEVICE_INSERTED	3

Param - Used by the module that triggers the notification to pass relevant information about the notification to the application.

4.1.23 CSSM_PADDING

```
typedef enum cssm_padding {
   CSSM_PADDING_NONE          = 0,
   CSSM_PADDING_CUSTOM        = CSSM_PADDING_NONE+1,
   CSSM_PADDING_ZERO          = CSSM_PADDING_NONE+2,
   CSSM_PADDING_ONE           = CSSM_PADDING_NONE+3,
   CSSM_PADDING_ALTERNATE     = CSSM_PADDING_NONE+4,
   CSSM_PADDING_FF            = CSSM_PADDING_NONE+5,
   CSSM_PADDING_PKCS5         = CSSM_PADDING_NONE+6,
   CSSM_PADDING_PKCS7         = CSSM_PADDING_NONE+7,
   CSSM_PADDING_CipherStealing = CSSM_PADDING_NONE+8,
   CSSM_PADDING_RANDOM        = CSSM_PADDING_NONE+9
} CSSM_PADDING;
```

4.1.24 CSSM_QUERY_SIZE_DATA

```
typedef struct cssm_query_size_data {
   uint32 SizeInputBlock;
   uint32 SizeOutputBlock;
} CSSM_QUERY_SIZE_DATA, *CSSM_QUERY_SIZE_DATA_PTR
```

Definitions:

SizeInputBlock - The size of the input block in bytes.

SizeOutputBlock - The size of the output block in bytes.

4.1.25 CSSM_RANGE

```
typedef struct cssm_range {
uint32 Min;/* inclusive minimium value */
uint32 Max;/* inclusive maximium value */
} CSSM_RANGE, *CSSM_RANGE_PTR
```

Definitions:

Min - Minimum value in the range.

Max - Maximum value in the range.

4.1.26 CSSM_SOFTWARECSPSUBSERVICEINFO

```
typedef struct cssm_softwarecspsubserviceinfo {
   uint32 NumberOfCapabilities;
   CSSM_CSP_CAPABILITY_PTR CapabilityList;
   void* Reserved;
} CSSM_SOFTWARE_CSPSUBSERVICE_INFO,
*CSSM_SOFTWARE_CSPSUBSERVICE_INFO_PTR;
```

Definitions:

NumberOfCapabilities - Number of capabilities available from the CSP.

CapabilityList - Pointer to an array of CSSM_CSP_CAPABILITY structures that represent the capabilities available from the CSP.

Reserved – Reserved for future use.

4.2 Cryptographic Context Operations

4.2.1 CSSM_CSP_CreateAsymmetricContext

CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateAsymmetricContext
(CSSM_CSP_HANDLE CSPHandle,
uint32 AlgorithmID,
const CSSM_CRYPTO_DATA_PTR PassPhrase,
const CSSM_KEY_PTR Key,
uint32 Padding)

This function creates an asymmetric encryption cryptographic context and returns the cryptographic context handle. The handle can be used to call asymmetric encryption functions and cryptographic wrap/unwrap functions.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function. If a NULL handle is specified, KeyWorks returns an error.

AlgorithmID (input)
The algorithm identification number for the algorithm used for asymmetric encryption.

PassPhrase (input)
The passphrase is required to unlock the private key. The passphrase structure accepts an immediate value for the passphrase or the caller can specify a callback function the CSP can use to obtain the passphrase. The passphrase is needed only for signature operations, not verify operations. When the context is used for a wrap or unwrap operation, the passphrase can be used to generate a symmetric key for wrapping or unwrapping.

Key (input)
The key used for asymmetric encryption. The caller passes a pointer to a CSSM_KEY structure containing the key. When the context is used for a sign operation, the public key and passphrase are required to access the private key used for signing. When the context is used for a verify operation, the public key is used to verify the signature. When the context is used for a wrapkey operation, the public key can be used as the wrapping key. When the context is used for an unwrap operation, the public key and the passphrase can be used to access the private key used to perform the unwrapping.

Padding (input/optional)
The method for padding. Typically specified for ciphers that pad.

Return Value

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_EncryptData, CSSM_QuerySize, CSSM_EncryptDataInit, CSSM_EncryptDataUpdate, CSSM_EncryptDataFinal, CSSM_DecryptData, CSSM_DecryptDataInit, CSSM_DecryptDataUpdate, CSSM_DecryptDataFinal, CSSM_GetContext, CSSM_SetContext, CSSM_DeleteContext, CSSM_GetContextAttribute, CSSM_UpdateContextAttributes

4.2.2 CSSM_CSP_CreateDeriveKeyContext

CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateDeriveKeyContext
(CSSM_CSP_HANDLE CSPHandle,
uint32 AlgorithmID,
CSSM_KEY_TYPE DeriveKeyType,
uint32 DeriveKeyLength,
uint32 IterationCount,
const CSSM_DATA_PTR Salt,
const CSSM_CRYPTO_DATA_PTR Seed,
const CSSM_CRYPTO_DATA_PTR PassPhrase)

This function creates a cryptographic context to derive either a symmetric key or an asymmetric key, and returns a handle to the context. The cryptographic context handle can be used for calling the cryptographic derive key function.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function. If a NULL handle is specified, KeyWorks returns an error.

AlgorithmID (input)
The algorithm identification number for a derived key algorithm.

DeriveKeyType (input)
The type of key to derive.

DeriveKeyLength (input)
The length of key to derive.

IterationCount (input/optional)
The number of iterations to be performed during the derivation process. Used heavily by password-based derivation methods.

Salt (input/optional)
A Salt used to generate the key.

Seed (input/optional)
A seed used to generate a random number. The caller can both pass a seed and seed length in bytes or pass in a callback function. If NULL is passed, the CSP will use its default seed handling mechanism.

PassPhrase (input/optional)
The passphrase is required to unlock the private key. The passphrase structure accepts an immediate value for the passphrase or the caller can specify a callback function the CSP can use to obtain the passphrase. The passphrase is needed only for signature operations, not verify operations.

Return Value

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DeriveKey

4.2.3 CSSM_CSP_CreateDigestContext

CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateDigestContext
(CSSM_CSP_HANDLE CSPHandle,
uint32 AlgorithmID)

This function creates a digest cryptographic context, given a handle of a CSP and an algorithm identification number. The cryptographic context handle is returned. The cryptographic context handle can be used to call digest cryptographic functions.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function. If a NULL handle is specified, KeyWorks returns an error.

AlgorithmID (input)
The algorithm identification number for message digests.

Return Value

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DigestData, CSSM_DigestDataInit, CSSM_DigestDataUpdate, CSSM_DigestDataFinal, CSSM_GetContext, CSSM_SetContext, CSSM_DeleteContext, CSSM_GetContextAttribute, CSSM_UpdateContextAttributes

4.2.4 CSSM_CSP_CreateKeyGenContext

CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateKeyGenContext
(CSSM_CSP_HANDLE CSPHandle,
uint32 AlgorithmID,
const CSSM_CRYPTO_DATA_PTR PassPhrase,
uint32 KeySizeInBits,
const CSSM_CRYPTO_DATA_PTR Seed,
const CSSM_DATA_PTR Salt,
const CSSM_DATE_PTR StartDate,
const CSSM_DATE_PTR EndDate,
const CSSM_DATA_PTR Params)

This function creates a key generation cryptographic context and returns a handle to the context. The cryptographic context handle can be used to call key/keypair generation functions.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function. If a NULL handle is specified, KeyWorks returns an error.

AlgorithmID (input)
The algorithm identification number of the algorithm used for key generation.

PassPhrase (input)
The passphrase is required to unlock the private key. The passphrase structure accepts an immediate value for the passphrase or the caller can specify a callback function the CSP can use to obtain the passphrase. The passphrase is needed only for signature operations, not verify operations. Once the new key is created, the passphrase or nickname must be provided in all future references to access the private or symmetric key.

KeySizeInBits (input)
The logical size of the key (specified in bits). This refers to either the actual key size (for symmetric key generation) or the modulus size (for asymmetric key pair generation). This is the effective key size.

Seed (input/optional)
A seed used to generate the key. The caller can either pass a seed or seed length in bytes or pass in a callback function. If NULL is passed, the CSP will use its default seed handling mechanism.

Salt (input/optional)
A Salt used to generate the key.

StartDate (input/optional)
Date from which the corresponding key is valid. All fields of the CSSM_DATE structure will be set to zero if the date is unspecified or unknown. The CSP module does not enforce this date.

EndDate (input/optional)
Data that the key expires and can no longer be used. All fields of the CSSM_DATE structure will be set to zero if the date is unspecified or unknown. The CSP module does not enforce this date.

Params (input/optional)
A data buffer containing parameters required to generate a key pair for a specific algorithm.

Return Value

Returns a cryptographic context handle. If the handle is NULL, an error has occurred and KeyWorks was unable to create the context. Use CSSM_GetError to obtain the error code.

See Also

CSSM_GenerateKey, CSSM_GenerateKeyPair, CSSM_GetContext, CSSM_SetContext, CSSM_DeleteContext, CSSM_GetContextAttribute, CSSM_UpdateContextAttributes

4.2.5 CSSM_CSP_CreateMacContext

CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateMacContext
(CSSM_CSP_HANDLE CSPHandle,
uint32 AlgorithmID,
const CSSM_KEY_PTR Key)

This function creates a Message Authentication Code (MAC) cryptographic context and returns a handle to the context. The cryptographic context handle can be used to call MAC functions. Note that MAC contexts that use RC2 require an effective key size in bits attribute. To add this attribute, use CSSM_UpdateContextAttributes.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function. If a NULL handle is specified, KeyWorks returns error.

AlgorithmID (input)
The algorithm identification number for the MAC algorithm.

Key (input)
The key used to generate a MAC. Caller passes in a pointer to a CSSM_KEY structure containing the key.

Return Value

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_GenerateMac, CSSM_GenerateMacInit, CSSM_GenerateMacUpdate, CSSM_GenerateMacFinal, CSSM_VerifyMAC, CSSM_VerifyMACInit, CSSM_VerifyMACUpdate, CSSM_VerifyMACFinal, CSSM_GetContext, CSSM_SetContext, CSSM_DeleteContext, CSSM_GetContextAttribute, CSSM_UpdateContextAttributes

4.2.6 CSSM_CSP_CreatePassThroughContext

CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreatePassThroughContext
(CSSM_CSP_HANDLE CSPHandle,
const CSSM_KEY_PTR Key,
const CSSM_DATA_PTR ParamBufs,
uint32 ParamBufCount)

This function creates a custom cryptographic context and returns a handle to the context. The cryptographic context handle can be used to call the CSSM_CSP_PassThrough function for the CSP.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function. If a NULL handle is specified, KeyWorks returns error.

Key (input)
The key to be used for the context. The caller passes in a pointer to a CSSM_KEY structure containing the key.

ParamBufs (input)
Array of input buffers to the passthrough call.

ParamBufCount (input)
The number of input buffers pointed to by *ParamBufs*.

Return Value

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

Comments

A CSP can create its own set of custom functions. The context information can be passed through its own data structure. The CSSM_CSP_PassThrough function should be used along with the function ID to call the desired custom function.

See Also

CSSM_CSP_PassThrough, CSSM_GetContext, CSSM_SetContext, CSSM_DeleteContext, CSSM_GetContextAttribute, CSSM_UpdateContextAttributes

4.2.7 CSSM_CSP_CreateRandomGenContext

CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateRandomGenContext
(CSSM_CSP_HANDLE CSPHandle,
uint32 AlgorithmID,
const CSSM_CRYPTO_DATA_PTR Seed,
uint32 Length)

This function creates a random number generation cryptographic context, given a handle of a CSP, an algorithm identification number, a seed, and the length of the random number in bytes. The cryptographic context handle is returned and can be used for the random number generation function.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function. If a NULL handle is specified, KeyWorks returns an error.

AlgorithmID (input)
The algorithm identification number for random number generation.

Seed (input/optional)
A seed used to generate random number. The caller can either pass a seed or seed length in bytes or pass in a callback function. If NULL is passed, the CSP will use its default seed handling mechanism.

Length (input)
The length of the random number to be generated.

Return Value

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_GenerateRandom, CSSM_GetContext, CSSM_SetContext, CSSM_DeleteContext, CSSM_GetContextAttribute, CSSM_UpdateContextAttributes

4.2.8 CSSM_CSP_CreateSignatureContext

CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateSignatureContext
(CSSM_CSP_HANDLE CSPHandle,
uint32 AlgorithmID,
const CSSM_CRYPTO_DATA_PTR PassPhrase,
const CSSM_KEY_PTR Key)

This function creates a signature cryptographic context for sign and verify operations given a handle of a CSP, an algorithm identification number, a passphrase structure, and a key. The passphrase will be used to unlock the private key when this context is used to perform a signing operation. The cryptographic context handle is returned. The cryptographic context handle can be used to call sign and verify cryptographic functions.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function. If a NULL handle is specified, KeyWorks returns an error.

AlgorithmID (input)
The algorithm identification number for a signature/verification algorithm.

PassPhrase (input)
The passphrase is required to unlock the private key. The passphrase structure accepts an immediate value for the passphrase or the caller can specify a callback function the CSP can use to obtain the passphrase. The passphrase is needed only for signature operations, not verify operations.

Key (input)
The key used to sign. The caller passes in a pointer to a CSSM_KEY structure containing the key and the key length.

Return Value

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_SignData, CSSM_SignDataInit, CSSM_SignDataUpdate, CSSM_SignDataFinal, CSSM_VerifyData, CSSM_VerifyDataInit, CSSM_VerifyDataUpdate, CSSM_VerifyDataFinal, CSSM_GetContext, CSSM_SetContext, CSSM_DeleteContext, CSSM_GetContextAttribute, CSSM_UpdateContextAttributes

4.2.9 CSSM_CSP_CreateSymmetricContext

CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateSymmetricContext
(CSSM_CSP_HANDLE CSPHandle,
uint32 AlgorithmID,
uint32 Mode,
const CSSM_KEY_PTR Key,
const CSSM_DATA_PTR InitVector,
uint32 Padding,
uint32 Params)

This function creates a symmetric encryption cryptographic context and returns a handle to the context. The cryptographic context handle can be used to call symmetric encryption functions and the cryptographic wrap/unwrap functions.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function. If a NULL handle is specified, KeyWorks returns an error.

AlgorithmID (input)
The algorithm identification number for symmetric encryption.

Mode (input)
The mode of the specified algorithm ID.

Key (input)
The key used for symmetric encryption. The caller passes in a pointer to a CSSM_KEY structure containing the key. This key can be used directly for wrap and unwrap operations.

InitVector (input/optional)
The initial vector for symmetric encryption; typically specified for block ciphers.

Padding (input/optional)
The method for padding; typically specified for ciphers that pad.

Params (input/optional)
Specifies the number of rounds of encryption; used for ciphers with variable number of rounds, such as RC5. For ciphers such as RC2, this parameter specifies the effective key size in bits.

Return Value

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_EncryptData, CSSM_QuerySize, CSSM_EncryptDataInit, CSSM_EncryptDataUpdate, CSSM_EncryptDataFinal, CSSM_DecryptData, CSSM_DecryptDataInit, CSSM_DecryptDataUpdate, CSSM_DecryptDataFinal, CSSM_GetContext, CSSM_SetContext, CSSM_DeleteContext, CSSM_GetContextAttribute, CSSM_UpdateContextAttributes

4.2.10 CSSM_DeleteContext

CSSM_RETURN CSSMAPI CSSM_DeleteContext (CSSM_CC_HANDLE CCHandle)

This function frees the context structure allocated by any of the create context functions.

Parameters

CCHandle (input)
The handle associated with the context to be deleted.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CSP_CreateSymmetricContext, CSSM_CSP_CreateAsymmetricContext, CSSM_CSP_CreateKeyGenContext, CSSM_CSP_CreateDigestContext, CSSM_CSP_CreateSignatureContext, etc

4.2.11 CSSM_DeleteContextAttributes

CSSM_RETURN CSSMAPI CSSM_DeleteContextAttributes
(CSSM_CC_HANDLE CCHandle,
uint32 NumberAttributes,
const CSSM_CONTEXT_ATTRIBUTE_PTR ContextAttributes);

This function deletes internal data associated with given attribute type of the context handle.

Parameters

CCHandle (input)
The handle that describes the context from which some attributes will be deleted.

NumberAttributes (input)
The number of attributes to be deleted.

ContextAttributes (input)
An array of pointers to the attributes to be deleted from the context.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_GetContextAttribute, CSSM_UpdateContextAttributes

4.2.12 CSSM_FreeContext

CSSM_RETURN CSSMAPI CSSM_FreeContext (CSSM_CONTEXT_PTR Context)

This function frees the memory associated with the context structure.

Parameters

Context (input)
The pointer to the memory that describes the context structure.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_GetContext

4.2.13 CSSM_GetContext

CSSM_CONTEXT_PTR CSSMAPI CSSM_GetContext (CSSM_CC_HANDLE CCHandle)

This function retrieves the context information when provided with a context handle.

Parameters

CCHandle (input)
The handle to the context information.

Return Value

The pointer to the CSSM_CONTEXT structure that describes the context associated with the handle CCHandle. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code. Call CSSM_FreeContext to free the memory allocated by KeyWorks.

See Also

CSSM_SetContext, CSSM_FreeContext

4.2.14 CSSM_GetContextAttribute

CSSM_CONTEXT_ATTRIBUTE_PTR CSSMAPI CSSM_GetContextAttribute
(const CSSM_CONTEXT_PTR Context,
uint32 AttributeType)

This function retrieves the context attributes information for the given context handle and attribute type. Note that not all context attributes can be queried using this function. For example, key size cannot be queried. To determine the key size, query the key. The key size data is contained in the header of the key. The following attribute types can be retrieved using CSSM_GetContextAttribute:

- custom
- CSP handle
- passphrase
- effective bits
- key
- key length
- key type
- key attributes
- output size
- seed
- rounds
- salt
- start date
- end date
- remote KR profile
- local KR profile
- padding
- random
- mode
- algorithm parameters
- initialization vector

Parameters

Context (input)
A pointer to the context.

AttributeType (input)
The attribute type of the specified context given CCHandle.

Return Value

The pointer to the CSSM_ATTRIBUTE structure that describes the context attributes associated with the handle CCHandle and the attribute type. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code. Call CSSM_DeleteContextAttributes to free memory allocated by KeyWorks.

See Also

CSSM_DeleteContextAttributes, CSSMGetContext

4.2.15 CSSM_SetContext

CSSM_RETURN CSSMAPI CSSM_SetContext (CSSM_CC_HANDLE CCHandle, const CSSM_CONTEXT_PTR Context)

This function replaces the context information associated with an existing context handle with the new context information supplied in *Context*. Before replacing the context, this function queries the service provider module associated with the context to ensure that the services requested from it are available in the provider.

Parameters

CCHandle (input)
The handle to the context.

Context (input)
The context data describing the service to replace the current service associated with context handle CCHandle.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSMGetContext

4.2.16 CSSM_UpdateContextAttributes

CSSM_RETURN CSSMAPI CSSM_UpdateContextAttributes
(CSSM_CC_HANDLE CCHandle,
uint32 NumberAttributes,
const CSSM_CONTEXT_ATTRIBUTE_PTR ContextAttributes)

This function updates the security context. When an attribute is already present in the context, this update operation replaces the previously defined attribute with the current attribute.

Parameters

CCHandle (input)
The handle to the context.

NumberAttributes (input)
The number of CSSM_CONTEXT_ATTRIBUTE structures to allocate.

ContextAttributes (input)
Pointer to data that describes the attributes to be associated with this context.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_GetContextAttribute, CSSM_DeleteContextAttributes

4.3 Cryptographic Sessions and Logon

4.3.1 CSSM_CSP_ChangeLoginPassword

CSSM_RETURN CSSMAPI CSSM_CSP_ChangeLoginPassword
(CSSM_CSP_HANDLE CSPHandle,
const CSSM_CRYPTO_DATA_PTR OldPassword,
const CSSM_CRYPTO_DATA_PTR NewPassword)

Changes the login password of the current login session from the old password to the new password. The requesting user must have a login session in process.

Parameters

CSPHandle (input)
Handle of the CSP supporting the current login session.

OldPassword (input)
Current password used to log into the token.

NewPassword (input)
New password to be used for future logins by this user to this token.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_CSP_Login, CSSM_CSP_Logout

4.3.2 CSSM_CSP_Login

CSSM_RETURN CSSMAPI CSSM_CSP_Login (CSSM_CSP_HANDLE CSPHandle,
const CSSM_CRYPTO_DATA_PTR Password,
const CSSM_DATA_PTR pReserved)

Logs the user into the CSP, allowing for multiple login types and parallel operation notification.

Parameters

CSPHandle (input)
Handle of the CSP to log into.

Password (input)
Password used to log into the token.

pReserved (input)
This field is reserved for future use. The value NULL should always be given.

Return Value

CSSM_OK if login is successful, CSSM_FAIL is login fails. Use CSSM_GetError to determine the exact error.

See Also

CSSM_CSP_ChangeLoginPassword, CSSM_CSP_Logout

4.3.3 CSSM_CSP_Logout

CSSM_RETURN CSSMAPI CSSM_CSP_Logout (CSSM_CSP_HANDLE CSPHandle)

Terminates the login session associated with the specified CSP Handle.

Parameters

CSPHandle (input)
Handle for the target CSP.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_CSP_Login, CSSM_CSP_ChangeLoginPassword

4.4 Cryptographic Operations

4.4.1 CSSM_DecryptData

CSSM_RETURN CSSMAPI CSSM_DecryptData (const CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR CipherBufs,
uint32 CipherBufCount,
CSSM_DATA_PTR ClearBufs,
uint32 ClearBufCount,
uint32 *bytesDecrypted,
CSSM_DATA_PTR RemData)

This function decrypts the supplied encrypted data. The CSSM_QuerySize function can be used to estimate the output buffer size required. When working with U.S. exportable versions of the KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

CipherBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

CipherBufCount (input)
The number of *CipherBufs*.

ClearBufs (output)
A pointer to a vector of CSSM_DATA structures that contain the decrypted data resulting from the decryption operation.

ClearBufCount (input)
The number of *ClearBufs*.

BytesDecrypted (output)
The size of the decrypted data in bytes.

RemData (output)
A pointer to the CSSM_DATA structure for the last decrypted block.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; the application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place decryption can be done by supplying the same input and output buffer.

See Also

CSSM_QuerySize, CSSM_EncryptData, CSSM_DecryptDataInit, CSSM_DecryptDataUpdate, CSSM_DecryptDataFinal, CSSM_RequestCssmExemptionCSSM_DecryptDataFinal

4.4.2 CSSM_DecryptDataFinal

CSSM_RETURN CSSMAPI CSSM_DecryptDataFinal (CSSM_CC_HANDLE CCHandle,
CSSM_DATA_PTR RemData)

This function finalizes the staged decrypt function. When working with U.S. exportable versions of the KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RemData (output)
A pointer to the CSSM_DATA structure for the last decrypted block.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free . If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.
In-place decryption can be done by supplying the same input and output buffers.

See Also

CSSM_DecryptData, CSSM_DecryptDataInit, CSSM_DecryptDataUpdate, CSSM_RequestCssmExemption

4.4.3 CSSM_DecryptDataInit

CSSM_RETURN CSSMAPI CSSM_DecryptDataInit (CSSM_CC_HANDLE CCHandle)

This function initializes the staged decrypt function. When working with U.S. exportable versions of the IBM KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_DecryptData, CSSM_DecryptDataUpdate, CSSM_DecryptDataFinal, CSSM_RequestCssmExemption

4.4.4 CSSM_DecryptDataUpdate

CSSM_RETURN CSSMAPI CSSM_DecryptDataUpdate
(CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR CipherBufs,
uint32 CipherBufCount,
CSSM_DATA_PTR ClearBufs,
uint32 ClearBufCount,
uint32 *bytesDecrypted)

This function updates the staged decrypt function. The CSSM_QuerySize function can be used to estimate the output buffer size required for each update call. There may be algorithm-specific and token-specific rules restricting the lengths of data in CSSM_DecryptDataUpdate calls. When working with U.S. exportable versions of the KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

CipherBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

CipherBufCount (input)
The number of *CipherBufs*.

ClearBufs (output)
A pointer to a vector of CSSM_DATA structures that contain the decrypted data resulting from the decryption operation.

ClearBufCount (input)
The number of *ClearBufs*.

bytesDecrypted (output)
A pointer to uint32 for the size of the decrypted data in bytes.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.
In-place decryption can be done by supplying the same input and output buffers.

See Also

CSSM_DecryptData, CSSM_DecryptDataInit, CSSM_DecryptDataFinal, CSSM_QuerySize, CSSM_RequestCssmExemption

4.4.5 CSSM_DeriveKey

CSSM_RETURN CSSMAPI CSSM_DeriveKey (CSSM_CC_HANDLE CCHandle,
const CSSM_KEY_PTR BaseKey,
void *Param,
uint32 KeyUsage,
uint32 KeyAttr,
const CSSM_DATA_PTR KeyLabel,
CSSM_KEY_PTR DerivedKey)

This function derives a new asymmetric key using the context and information from the base key.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation.

BaseKey (input)
The base key used to derive the new key. The base key may be a public key, a private key, or an asymmetric key.

Param (input/output)
The use of this parameter varies depending on the derivation algorithms. Specific algorithms use *Params* to pass custom data to algorithms.

KeyUsage (input)
A bit-mask representing the valid uses of the key. See Table 14 for a list of valid values.

KeyAttr (input)
A bit-mask representing the attributes of the key represented by the data. These attributes are used by CSP service providers to convey information about stored or referenced keys.

KeyLabel (input/optional)
Pointer to a byte string that will be used as the label for the derived key.

DerivedKey (output)
A pointer to a CSSM_KEY structure that returns the derived key.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The KeyData field of the CSSM_KEY structure is not required to be allocated. In this case, the memory required to represent the key is allocated by the CSP. The application is required to free this memory. The CSP will only allocate memory if the *Data* field of *KeyData* is NULL and the *Length* field is zero.

See Also

CSSM_CSP_CreateDeriveKeyContext

4.4.6 CSSM_DigestData

CSSM_RETURN CSSMAPI CSSM_DigestData (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount,
CSSM_DATA_PTR Digest)

This function computes a message digest for the supplied data.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs*.

Digest (output)
A pointer to the CSSM_DATA structure for the message digest.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer this is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

See Also

CSSM_DigestDataInit, CSSM_DigestDataUpdate, CSSM_DigestDataFinal

4.4.7 CSSM_DigestDataClone

CSSM_CC_HANDLE CSSMAPI CSSM_DigestDataClone (CSSM_CC_HANDLE CCHandle)

This function clones a given staged message digest context with its cryptographic attributes and intermediate result.

Parameters

CCHandle (input)
The handle that describes the context of a staged message digest operation.

Return Value

The handle of cloned context. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

Comments

When a digest context is cloned, a new context is created with data associated with the parent context. Changes made to the parent context after calling this function will not be reflected in the cloned context. The cloned context could be used with the CSSM_DigestDataUpdate and CSSM_DigestDataFinal functions.

See Also

CSSM_DigestData, CSSM_DigestDataInit, CSSM_DigestDataUpdate, CSSM_DigestDataFinal

4.4.8 CSSM_DigestDataFinal

CSSM_RETURN CSSMAPI CSSM_DigestDataFinal (CSSM_CC_HANDLE CCHandle,
CSSM_DATA_PTR Digest)

This function finalizes the staged message digest function.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Digest (output)
A pointer to the CSSM_DATA structure for the message digest.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

See Also

CSSM_DigestData, CSSM_DigestDataInit, CSSM_DigestDataUpdate, CSSM_DigestDataClone

4.4.9 CSSM_DigestDataInit

CSSM_RETURN CSSMAPI CSSM_DigestDataInit (CSSM_CC_HANDLE CCHandle)

This function initializes the staged message digest operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_DigestData, CSSM_DigestDataUpdate, CSSM_DigestDataClone, CSSM_DigestDataFinal

4.4.10 CSSM_DigestDataUpdate

CSSM_RETURN CSSMAPI CSSM_DigestDataUpdate (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount)

This function updates the staged message digest operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs*.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_DigestData, CSSM_DigestDataInit, CSSM_DigestDataClone, CSSM_DigestDataFinal

4.4.11 CSSM_EncryptData

CSSM_RETURN CSSMAPI CSSM_EncryptData (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR ClearBufs,
uint32 ClearBufCount,
CSSM_DATA_PTR CipherBufs,
uint32 CipherBufCount,
uint32 *bytesEncrypted,
CSSM_DATA_PTR RemData)

This function encrypts the supplied data using information in the context. The CSSM_QuerySize function can be used to estimate the output buffer size required. When working with U.S. exportable versions of the KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters
CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ClearBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

ClearBufCount (input)
The number of *ClearBufs*.

CipherBufs (output)
A pointer to a vector of CSSM_DATA structures that contain the results of the operation on the data.

CipherBufCount (input)
The number of *CipherBufs*.

bytesEncrypted (output)
The size of the encrypted data in bytes.

RemData (output)
A pointer to the CSSM_DATA structure for the last encrypted block containing padded data.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.
In-place encryption can be done by supplying the same input and output buffers.

See Also

CSSM_QuerySize, CSSM_DecryptData, CSSM_EncryptDataInit, CSSM_EncryptDataUpdate, CSSM_EncryptDataFinal, CSSM_RequestCssmExemption

4.4.12 CSSM_EncryptDataFinal

CSSM_RETURN CSSMAPI CSSM_EncryptDataFinal (CSSM_CC_HANDLE CCHandle,
CSSM_DATA_PTR RemData)

This function finalizes the staged encrypt operation. When working with U.S. exportable versions of the KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RemData (output)
A pointer to the CSSM_DATA structure for the last encrypted block containing padded data.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.
In-place encryption can be done by supplying the same input and output buffers.

See Also

CSSM_EncryptData, CSSM_EncryptDataInit, CSSM_EncryptDataUpdate, CSSM_RequestCssmExemption

4.4.13 CSSM_EncryptDataInit

CSSM_RETURN CSSMAPI CSSM_EncryptDataInit (CSSM_CC_HANDLE CCHandle)

This function initializes the staged encrypt operation. There may be algorithm-specific and token-specific rules restricting the lengths of data in the following CSSM_EncryptDataUpdate calls that make use of these parameters. When working with U.S. exportable versions of the KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_EncryptData, CSSM_EncryptDataUpdate, CSSM_EncryptDataFinal, CSSM_RequestCssmExemption

4.4.14 CSSM_EncryptDataUpdate

CSSM_RETURN CSSMAPI CSSM_EncryptDataUpdate
(CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR ClearBufs,
uint32 ClearBufCount,
CSSM_DATA_PTR CipherBufs,
uint32 CipherBufCount,
uint32 *bytesEncrypted)

This function updates the staged encrypt operation. The CSSM_QuerySize function can be used to estimate the output buffer size required for each update call. There may be algorithm-specific and token-specific rules restricting the lengths of data in CSSM_EncryptUpdate calls. When working with U.S. exportable versions of the KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ClearBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

ClearBufCount (input)
The number of *ClearBufs*.

CipherBufs (output)
A pointer to a vector of CSSM_DATA structures that contain the encrypted data resulting from the encryption operation.

CipherBufCount (input)
The number of *CipherBufs*.

bytesEncrypted (output)
The size of the encrypted data in bytes.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.
In-place encryption can be done by supplying the same input and output buffers.

See Also

CSSM_EncryptData, CSSM_EncryptDataInit, CSSM_EncryptDataFinal, CSSM_QuerySize, CSSM_RequestCssmExemption

4.4.15 CSSM_GenerateAlgorithmParams

CSSM_RETURN CSSMAPI CSSM_GenerateAlgorithmParams
(CSSM_CC_HANDLE CCHandle,
uint32 ParamBits,
CSSM_DATA_PTR Param)

This function generates algorithm parameters for the specified context. These parameters include Diffie-Hellman key agreement parameters and DSA key generation parameters.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ParamBits (input)
Used to generate parameters for the algorithm (for example, Diffie-Hellman).

Param (output)
Pointer to CSSM_DATA structure used to obtain the key exchange parameter and the size of the key exchange parameter in bytes.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

4.4.16 CSSM_GenerateKey

CSSM_RETURN CSSMAPI CSSM_GenerateKey (CSSM_CC_HANDLE CCHandle,
uint32 KeyUsage,
uint32 KeyAttr,
const CSSM_DATA_PTR KeyLabel,
CSSM_KEY_PTR Key)

This function generates a symmetric key.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

KeyUsage (input)
A bit-mask representing the valid uses of the key. See Table 14 for a list of valid values.

KeyAttr (input)
A bit-mask representing the attributes of the key represented by the data. These attributes are used by CSP service providers to convey information about stored or referenced keys.

KeyLabel (input/optional)
Pointer to a byte string that will be used as a label/identifier for the derived key. If a key label is not used, this field should be set to NULL.

Key (output)
Pointer to CSSM_KEY structure containing the key.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The *KeyData* field of the CSSM_KEY structure is not required to be allocated. In this case, the memory required to represent the key is allocated by the CSP. The application is required to free this memory. The CSP will only allocate memory if the *Data* field of *KeyData* is NULL and the *Length* field is zero.

See Also

CSSM_GenerateRandom, CSSM_GenerateKeyPair

4.4.17 CSSM_GenerateKeyPair

CSSM_RETURN CSSMAPI CSSM_GenerateKeyPair (CSSM_CC_HANDLE CCHandle,
uint32 PublicKeyUsage,
uint32 PublicKeyAttr,
const CSSM_DATA_PTR PublicKeyLabel,
CSSM_KEY_PTR PublicKey,
uint32 PrivateKeyUsage,
uint32 PrivateKeyAttr,
const CSSM_DATA_PTR PrivateKeyLabel,
CSSM_KEY_PTR PrivateKey)

This function generates an asymmetric key pair.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

PublicKeyUsage (input/optional)
A bit-mask representing the valid uses of the public key. This field may be required by some CSP modules. Refer to the documentation provided with the CSP for more information. See Table 14 for a list of valid key usage values.

PublicKeyAttr (input/optional)
A bit-mask representing the attributes of the public key represented by the data. These attributes are used by CSP service providers to convey information about stored or referenced keys. This field may be required by some CSP modules. Refer to the documentation provided with the CSP for more information.

PublicKeyLabel (input/optional)
Pointer to a byte string that will be used as a label/identifier for the derived public key. If a key label is not used, this field should be set to NULL.

PublicKey (output)
Pointer to CSSM_KEY structure used to obtain the public key.

PrivateKeyUsage (input/optional)
A bit-mask representing the valid uses of the private key. This field may be required by some CSP modules. For more information, see the documentation provided with the CSP from the module vendor. See Table 14 for a list of valid key usage values.

PrivateKeyAttr (input/optional)
A bit-mask representing the attributes of the private key represented by the data. These attributes are used by CSP service providers to convey information about stored or referenced keys. This field may be required by some CSP modules. Refer to the documentation provided with the CSP for more information.

PrivateKeyLabel (input/optional)
Pointer to a byte string that will be used as a label/identifier for the derived private key. If a key label is not used, this field should be set to NULL.

PrivateKey (output)
Pointer to CSSM_KEY structure used to obtain the private key.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The *KeyData* field of the CSSM_KEY structures are not required to be allocated. In this case, the memory required to represent the key is allocated by the CSP. The application is required to free this memory. The CSP will only allocate memory if the *Data* field of *KeyData* is NULL and the *Length* field is zero.

See Also

CSSM_GenerateRandom

4.4.18 CSSM_GenerateMac

CSSM_RETURN CSSMAPI CSSM_GenerateMac (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount,
CSSM_DATA_PTR Mac)

This function generates a message authentication code for the supplied data.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs*.

Mac (output)
A pointer to the CSSM_DATA structure containing the message authentication code.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

See Also

CSSM_GenerateMacInit, CSSM_GenerateMacUpdate, CSSM_GenerateMacFinal

4.4.19 CSSM_GenerateMacFinal

CSSM_RETURN CSSMAPI CSSM_GenerateMacFinal (CSSM_CC_HANDLE CCHandle,
CSSM_DATA_PTR Mac)

This function finalizes the staged message authentication code operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Mac (output)
A pointer to the CSSM_DATA structure containing the message authentication code.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

See Also

CSSM_GenerateMac, CSSM_GenerateMacInit, CSSM_GenerateMacUpdate

4.4.20 CSSM_GenerateMacInit

CSSM_RETURN CSSMAPI CSSM_GenerateMacInit (CSSM_CC_HANDLE CCHandle)

This function initializes the staged message authentication code operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_GenerateMac, CSSM_GenerateMacUpdate, CSSM_GenerateMacFinal

4.4.21 CSSM_GenerateMacUpdate

CSSM_RETURN CSSMAPI CSSM_GenerateMacUpdate (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount)

This function updates the staged message authentication code operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs*.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_GenerateMac, CSSM_GenerateMacInit, CSSM_GenerateMacFinal

4.4.22 CSSM_GenerateRandom

CSSM_RETURN CSSMAPI CSSM_GenerateRandom (CSSM_CC_HANDLE CCHandle,
CSSM_DATA_PTR RandomNumber)

This function generates random data.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RandomNumber (output)
Pointer to CSSM_DATA structure used to obtain the random number and the size of the random number in bytes.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

4.4.23 CSSM_QueryKeySizeInBits

CSSM_RETURN CSSMAPI CSSM_QueryKeySizeInBits (CSSM_CSP_HANDLE CSPHandle,
CSSM_CC_HANDLE CCHandle,
CSSM_KEY_SIZE_PTR KeySize)

This function queries a CSP for the effective and real size of a key in bits.

Parameters

CSPHandle (input)
The handle that describes the CSP module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

KeySize (output)
Pointer to a CSSM_KEY_SIZE data structure returns the actual size and the effective size of the key in bits.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_GenerateRandom, CSSM_GenerateKeyPair

4.4.24 CSSM_QuerySize

CSSM_RETURN CSSMAPI CSSM_QuerySize (CSSM_CC_HANDLE CCHandle,
CSSM_BOOL Encrypt,
uint32 QuerySizeCount,
CSSM_QUERY_SIZE_DATA_PTR DataBlock)

This function queries for the size of the output data for Signature, Message Digest, and Message Authentication Code context types and queries for the algorithm block size, or the size of the output data for encryption and decryption context types. This function also can be used to query the output size requirements for the intermediate steps of a staged cryptographic operation (for example, CSSM_EncryptDataUpdate and CSSM_DecryptDataUpdate). There may be algorithm-specific and token-specific rules restricting the lengths of data in the following data update calls.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Encrypt (input)
When asymmetric and symmetric contexts are being used, *Encrypt* indicates whether an encryption (CSSM_TRUE) or a decryption (CSSM_FALSE) operation will be performed. For all other operations and context types, *Encrypt* should be set to CSSM_FALSE.

QuerySizeCount (input)
An integer that indicates the number of data blocks that are in *DataBlock*.

DataBlock (input/output)
A pointer to a CSSM_QUERY_SIZE_DATA structure that contains the size of the input and the output data blocks, in bytes.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_EncryptData, CSSM_EncryptDataUpdate, CSSM_DecryptData, CSSM_DecryptDataUpdate, CSSM_SignData, CSSM_VerifyData, CSSM_DigestData, CSSM_GenerateMac

4.4.25 CSSM_SignData

CSSM_RETURN CSSMAPI CSSM_SignData (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount,
CSSM_DATA_PTR Signature)

This function signs data using the private key associated with the public key specified in the context.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs* to be signed.

Signature (output)
A pointer to the CSSM_DATA structure containing the signature.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

See Also

CSSM_VerifyData, CSSM_SignDataInit, CSSM_SignDataUpdate, CSSM_SignDataFinal

4.4.26 CSSM_SignDataFinal

CSSM_RETURN CSSMAPI CSSM_SignDataFinal (CSSM_CC_HANDLE CCHandle,
CSSM_DATA_PTR Signature)

This function completes the final stage of the sign data operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Signature (output)
A pointer to the CSSM_DATA structure for the signature.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, which the application must later free. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

See Also

CSSM_SignData, CSSM_SignDataInit, CSSM_SignDataUpdate

4.4.27 CSSM_SignDataInit

CSSM_RETURN CSSMAPI CSSM_SignDataInit (CSSM_CC_HANDLE CCHandle)

This function initializes the staged sign data operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_SignData, CSSM_SignDataUpdate, CSSM_SignDataFinal

4.4.28 CSSM_SignDataUpdate

CSSM_RETURN CSSMAPI CSSM_SignDataUpdate (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount)

This function updates the data for the staged sign data operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs* to be signed.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_SignData, CSSM_SignDataInit, CSSM_SignDataFinal

4.4.29 CSSM_UnwrapKey

CSSM_RETURN CSSMAPI CSSM_UnwrapKey
(CSSM_CC_HANDLE CCHandle,
const CSSM_CRYPTO_DATA_PTR NewPassPhrase,
const CSSM_WRAP_KEY_PTR WrappedKey,
uint32 KeyAttr,
const CSSM_DATA_PTR KeyLabel,
CSSM_KEY_PTR UnwrappedKey)

This function unwraps the data using the context. When working with U.S. exportable versions of the KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation.

NewPassPhrase (input)
The passphrase or a callback function to be used to obtain the passphrase. If the unwrapped key is a private key and the persistent object mode is true, then the private key is unwrapped and securely stored by the CSP. The *NewPassPhrase* is used to secure the private key after it is unwrapped.
It is assumed that a known public key is associated with the private key.

WrappedKey (input)
A pointer to the wrapped key. The wrapped key may be a symmetric key or the private key of a public/private key pair. The unwrapping method is specified as meta-data within the wrapped key and is not specified outside of the wrapped key.

KeyAttr (input)
Attribute the unwrapped key will assume.

KeyLabel (input/optional)
Pointer to a byte string that will be used as the label for the unwrapped key.

UnwrappedKey (output)
A pointer to a CSSM_KEY structure that returns the unwrapped key.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

Comments

The KeyData field of the CSSM_KEY structure is not required to be allocated. In this case, the memory required to represent the key is allocated by the CSP. The application is required to free this memory. The CSP will only allocate memory if the *Data* field of *KeyData* is NULL and the *Length* field is zero.

See Also

CSSM_WrapKey, CSSM_RequestCssmExemption

4.4.30 CSSM_VerifyData

CSSM_BOOL CSSMAPI CSSM_VerifyData (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount,
const CSSM_DATA_PTR Signature)

This function verifies the input data against the provided signature.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs* to be verified.

Signature (input)
A pointer to a CSSM_DATA structure which contains the signature and the size of the signature.

Return Value

A CSSM_TRUE return value signifies the signature was successfully verified. When CSSM_FALSE is returned, either the signature was not successfully verified or an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_SignData, CSSM_VerifyDataInit, CSSM_VerifyDataUpdate, CSSM_VerifyDataFinal

4.4.31 CSSM_VerifyDataFinal

CSSM_BOOL CSSMAPI CSSM_VerifyDataFinal (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR Signature)

This function finalizes the staged verify data function.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Signature (input)
A pointer to a CSSM_DATA structure that contains the starting address for the signature to verify against and the length of the signature in bytes.

Return Value

A CSSM_TRUE return value signifies the signature successfully verified. When CSSM_FALSE is returned, either the signature was not successfully verified or an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_VerifyData, CSSM_VerifyDataInit, CSSM_VerifyDataUpdate

4.4.32 CSSM_VerifyDataInit

CSSM_RETURN CSSMAPI CSSM_VerifyDataInit (CSSM_CC_HANDLE CCHandle)

This function initializes the staged verify data operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_VerifyDataUpdate, CSSM_VerifyDataFinal, CSSM_VerifyData

4.4.33 CSSM_VerifyDataUpdate

CSSM_RETURN CSSMAPI CSSM_VerifyDataUpdate (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount)

This function updates the data to the staged verify data operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs* to be verified.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_VerifyData, CSSM_VerifyDataInit, CSSM_VerifyDataFinal

4.4.34 CSSM_VerifyMac

CSSM_RETURN CSSMAPI CSSM_VerifyMac (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount,
CSSM_DATA_PTR Mac)

This function verifies a message authentication code for the supplied data.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs*.

Mac (input)
A pointer to the CSSM_DATA structure containing the MAC to verify.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_VerifyMacInit, CSSM_VerifyMacUpdate, CSSM_VerifyMacFinal

4.4.35 CSSM_VerifyMacFinal

CSSM_RETURN CSSMAPI CSSM_VerifyMacFinal (CSSM_CC_HANDLE CCHandle,
CSSM_DATA_PTR Mac)

This function finalizes the staged message authentication code verification operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Mac (input)
A pointer to the CSSM_DATA structure containing the MAC to verify.

Return Value

CSSM_OK if the MAC verifies correctly, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_VerifyMac, CSSM_VerifyMacInit, CSSM_VerifyMacUpdate

4.4.36 CSSM_VerifyMacInit

CSSM_RETURN CSSMAPI CSSM_VerifyMacInit (CSSM_CC_HANDLE CCHandle)

This function initializes the staged message authentication code verification operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_VerifyMac, CSSM_VerifyMacUpdate, CSSM_VerifyMacFinal

4.4.37 CSSM_VerifyMacUpdate

CSSM_RETURN CSSMAPI CSSM_VerifyMacUpdate (CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR DataBufs,
uint32 DataBufCount)

This function updates the staged message authentication code verification operation.

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of *DataBufs*.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_VerifyMac, CSSM_VerifyMacInit, CSSM_VerifyMacFinal

4.4.38 CSSM_WrapKey

CSSM_RETURN CSSMAPI CSSM_WrapKey (CSSM_CC_HANDLE CCHandle,
const CSSM_CRYPTO_DATA_PTR PassPhrase,
const CSSM_KEY_PTR Key,
CSSM_KEY_PTR WrappedKey)

This function wraps the supplied key using the context. The key may be a symmetric key or the public key of a public/private key pair. If a symmetric key is specified, it is wrapped. If a public key is specified, the passphrase is used to unlock the corresponding private key, which is then wrapped. When working with U.S. exportable versions of the KeyWorks Toolkit, the caller may be required to possess specific exemptions or privileges in order to allow this call to complete successfully.

Parameters

CCHandle (input)
The handle to the context that describes this cryptographic operation.

PassPhrase (input)
The passphrase or a callback function to be used to obtain the passphrase that can be used by the CSP to unlock the private key before it is wrapped. This input is ignored when wrapping a symmetric, secret key.

Key (input)
A pointer to the target key to be wrapped. If a private key is to be wrapped, the target key is the public key associated with the private key. If a symmetric key is to be wrapped, the target key is that symmetric key.

WrappedKey (output)
A pointer to a CSSM_KEY structure that returns the wrapped key.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_UnwrapKey, CSSM_RequestCssmExemption

4.5 Extensibility Functions

The CSSM_CSP_PassThrough function allows CSP developers to extend the cryptographic functionality of the KeyWorks API. Because it is only exposed to KeyWorks as a function pointer, its name internal to the CSP can be assigned at the discretion of the CSP module developer. However, its parameter list and return value must match what is shown below.

4.5.1 CSSM_CSP_PassThrough

void * CSSMAPI CSSM_CSP_PassThrough (CSSM_CC_HANDLE CCHandle,
uint32 PassThroughId,
const void *InData)

Parameters

CCHandle (input)
The handle that describes the context of this cryptographic operation.

PassThroughId (input)
An identifier specifying the custom function to be performed.

InData (input)
A pointer to a module-specific structure containing the input data.

Return Value

A pointer to a module-specific structure containing the output data. If successful, this function returns a non-NULL value. A NULL value indicates that an error has occurred. Use CSSM_GetError to obtain a specific error code.

Chapter 5. Key Recovery Services API

5.1 Data Structures

5.1.1 CSSM_CERTGROUP

This data structure contains a set of certificates that are related based on cosignaturing. This certificate group is a syntactic representation of a trust model.

```
typedef struct cssm_certgroup{
        uint32 NumCerts; /* number of elements in CertList array */
        CSSM_DATA_PTR CertList; /* List of opaque certificates */
        void * reserved;
} CSSM_CERTGROUP, *CSSM_CERTGROUP_PTR;
```

5.1.2 CSSM_CONTEXT_ATTRIBUTE Extensions

The key recovery, context creation operations return key recovery context handles that are represented as cryptographic context handles. The CSSM_CONTEXT data structure has been extended to include the new types of attributes, as shown below:

```
typedef struct cssm_context_attribute {
   uint32 AttributeType;/* one of the defined CSSM_ATTRIBUTE_TYPEs */
   uint32 AttributeLength;/* length of attribute */
   union {
      uint8 *Description; uint32 *Length;
      void *Pointer;
      CSSM_CRYPTO_DATA_PTR SeedPassPhrase;
      CSSM_KEY_PTR Key;
      CSSM_DATA_PTR Data;
      CSSM_KR_PROFILE_PTR KRProfile;/*new attribute to hold KR profile*/
   } Attribute; /* data that describes attribute */
} CSSM_CONTEXT_ATTRIBUTE, *CSSM_CONTEXT_ATTRIBUTE_PTR;
```

Several new attribute types were defined to support the key recovery context attributes. The CSSM_ATTRIBUTE_TYPE enum is extended as follows:

```
CSSM_ATTRIBUTE_KRPROFILE_LOCAL = CSSM_ATTRIBUTE_LAST + 1,
CSSM_ATTRIBUTE_KRPROFILE_REMOTE= CSSM_ATTRIBUTE_LAST + 2
```

5.1.3 CSSM_KR_NAME

This data structure contains a typed name. The namespace type specifies what kind of name is contained in the third parameter.

```
typedef struct cssm_kr_name {
   uint8 Type; /* namespace type */
   uint8 Length; /* name string length */
   char *Name; /* name string */
} CSSM_KR_NAME, *CSSM_KR_NAME_PTR;
```

5.1.4 CSSM_KR_PROFILE

This data structure encapsulates the key recovery profile for a given user and a given key recovery mechanism.

```
typedef struct cssm_kr_profile {
   CSSM_KR_NAME UserName;
   CSSM_DATA_PTR UserCertificate;
   CSSM_CERTGROUP_PTR KRSCertChain;
   uint8 LE_KRANum;
   CSSM_CERTGROUP_PTR LE_KRACertChainList;
   uint8 ENT_KRANum;
   CSSM_CERTGROUP_PTR ENT_KRACertChainList;
   uint8 INDIV_KRANum;
   CSSM_CERTGROUP_PTR INDIV_KRACertChainList;
   CSSM_DATA_PTR INDIV_AuthenticationInfo;
   uint32KRSPFlags;
   CSSM_DATA_PTR KRSPExtensions;
} CSSM_KR_PROFILE, *CSSM_KR_PROFILE_PTR;
```

Definitions:

UserName - Name of user this profile profiles.

UserCertificate - PK certificate of user.

KRSCertChain - Reserved for future use.

LE_KRANum - Number of law enforcement cert chains in *LE_KRACertChainList*.

LE_KRACertChainList - List of certificate chains for law enforcement.

ENT_KRANum - Number of enterprise cert chain in *ENT_KRACertChainList.*

ENT_KRACertChainList - List of certificate chains for enterprise.

INDIV_KRANum - Number of individual cert chains in *INDIV_KRACertChainList.*

INDIV_KRACertChainList - List of certificate chains for individual key recovery.

INDIV_AuthenticationInfo - Authentication Information (AI) for user key recovery.

KRSPFlags - Flag values interpreted by Key Recovery Service Provider (KRSP).

KRSPExtensions - Reserved for extensions specific to a key recovery module.

5.1.5 CSSM_KRSP_HANDLE

```
typedef uint32 CSSM_KRSP_HANDLE /* Key Recovery Service Provider Handle */
```

5.1.6 CSSM_KRSPSUBSERVICE

Three structures are used to contain all of the static information that describes a KRSP module: cssm_moduleinfo, cssm_serviceinfo, and cssm_krspsubservice. This descriptive information is securely stored in the KeyWorks registry when the key recovery module is installed with KeyWorks. A KRSP module may implement multiple types of services and organize them as subservices.

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo and specifying the KRSP module Globally Unique ID (GUID).

```
typedef struct cssm_krspsubservice {
   uint32 SubServiceId;
   CSSM_STRING Description;/* Description of this sub service */
   CSSM_STRING Jurisdiction;
} CSSM_KRSPSUBSERVICE, *CSSM_KRSPSUBSERVICE_PTR;
```

5.1.7 CSSM_KR_WRAPPEDPRODUCTINFO

```
typedef struct cssm_kr_wrappedproductinfo {
   CSSM_VERSION StandardVersion;
   CSSM_STRING StandardDescription;
   CSSM_VERSION ProductVersion;
   CSSM_STRING ProductDescription;
   CSSM_STRING ProductVendor;
   uint32 ProductFlags;
} CSSM_KR_WRAPPEDPRODUCT_INFO, *CSSM_KR_WRAPPEDPRODUCT_INFO_PTR;
```

Description:

StandardVersion - Version of standard to which this product conforms.

StandardDescription - Description of standard to which this product conforms.

ProductVersion - Version of wrapped product/library.

ProductDescription - Description of wrapped product/library.

ProductVendor - Vendor of wrapped product/library.

ProductFlags - ProductFlags.

5.2 Key Recovery Module Management Operations

5.2.1 CSSM_KR_SetEnterpriseRecoveryPolicy

CSSM_RETURN CSSMAPI CSSM_KR_SetEnterpriseRecoveryPolicy
(char * RecoveryPolicyFileName,
const CSSM_CRYPTO_DATA_PTR OldPassPhrase,
const CSSSM_CRYPTO_DATA_PTR NewPassphrase)

This call establishes the identity of the file that contains the enterprise key recovery policy function. It allows the use of a passphrase for access control to the update of the enterprise policy module. The first time this function is invoked, the old passphrase should be "default" in the Param field of the CSSM_CRYPTO_DATA_PTR. A passphrase can be established at this time for subsequent access control to this function by entering it in the NewPassphrase parameter. If the passphrase is to be changed, both the old and new passphrases have to be supplied.

The policy function module is operating system platform-specific (for Win95 and NT, it may be a Dynamic Link Library (DLL); for UNIX-based platforms, it may be a separate executable that gets launched by the KeyWorks). It is expected that the policy function file will be protected using the available protection mechanisms of the operating system platform. The policy function is expected to conform to the following interface:

CSSM_BOOL EnterpriseRecoveryPolicy (CSSM_CONTEXT_PTR CryptoContext);

The CSSM_BOOL return value of this policy function will determine whether enterprise-based key recovery is mandated for the given cryptographic operation. CSSM_TRUE means that key recovery enablement is required for the given Context, and CSSM_FALSE means it is not.

Parameters

RecoveryPolicyFileName (input)
A pointer to a character string that specifies the filename of the module that contains the enterprise key recovery policy function. The filename may be a fully qualified pathname or a partial pathname.

OldPassPhrase (input)
The passphrase used to control access to this operation. This should be "default" when this function is invoked for the first time.

NewPassPhrase (input)
The value of the passphrase to be established for subsequent access to this function. It should be identical to the OldPassPhrase if the passphrase does not need to be updated.

Return Values

A CSSM return value. This function returns CSSM_OK if successful, and returns CSSM_FAIL if an error has occurred. Use CSSM_GetError to determine the error code.

5.3 Key Recovery Context Operations

Key recovery contexts are essentially cryptographic contexts. The following API functions deal with the creation of these special types of cryptographic contexts. Once these contexts are created, the regular KeyWorks context API functions may be used to manipulate these key recovery contexts.

5.3.1 CSSM_KR_CreateRecoveryEnablementContext

CSSM_CC_HANDLE CSSMAPI CSSM_KR_CreateRecoveryEnablementContext
(CSSM_KRSP_HANDLE KRSPHandle,
const CSSM_KR_PROFILE LocalProfile,
const CSSM_KR_PROFILE RemoteProfile)

This call creates a key recovery enablement context based on a KRSP handle (which determines the key recovery mechanism that is in use) and key recovery profiles for the local and remote parties involved in a cryptographic exchange. It is expected that the LocalProfile will contain sufficient information to perform law enforcement, enterprise, and individual key recovery enablement, whereas, the RemoteProfile will contain information to perform law enforcement and enterprise key recovery enablement only. However, any and all of the fields within the profiles may be set to NULL–in this case, default values for these fields are to be used when performing the recovery enablement operations.

Parameters

KRSPHandle (input)
The handle to the KRSP that will be used.

LocalProfile (input)
The key recovery profile for the local client.

RemoteProfile (input)
The key recovery profile for the remote client.

Return Values

A handle to the key recovery enablement context is returned. If the handle is NULL, that signifies that an error has occurred.

5.3.2 CSSM_KR_CreateRecoveryRegistrationContext

CSSM_CC_HANDLE CSSMAPI CSSM_KR_CreateRecoveryRegistrationContext
(CSSM_KRSP_HANDLE KRSPHandle)

This call creates a key recovery registration context based on a KRSP handle (which determines the key recovery mechanism that is in use). This context may be used for performing registration with Key Recovery Servers (KRSs) and/or Key Recovery Agents (KRAs).

Parameters

KRSPHandle (input)
The handle to the KRSP that will be used.

Return Values

A handle to the key recovery registration context is returned. If the handle is NULL, that signifies that an error has occurred.

5.3.3 CSSM_KR_CreateRecoveryRequestContext

CSSM_CC_HANDLE CSSMAPI CSSM_KR_CreateRecoveryRequestContext
(CSSM_KRSP_HANDLE KRSPHandle,
const CSSM_KR_PROFILE_PTR LocalProfile)

This call creates a key recovery request context based on a KRSP handle (which determines the key recovery mechanism that is in use). The profile for the local party and flag values to signify what kind of key recovery is desired. A handle to the key recovery request context is returned.

Parameters

KRSPHandle (input)
The handle to the KRSP that is to be used.

LocalProfile (input)
The key recovery profile for the local client. This parameter is relevant only when KRFlags is set to KR_INDIV.

Return Values

A handle to the key recovery context is returned. If the handle is NULL, that signifies that an error has occurred.

5.3.4 CSSM_KR_GetPolicyInfo

CSSM_RETURN CSSM_KR_GetPolicyInfo (CSSM_CC_HANDLE CCHandle,
uint32 EncryptionProhibited,
uint32 *WorkFactor)

This call returns the key recovery policy information for a given cryptographic context. The information returned constitutes the key recovery extension fields of a cryptographic context.

Parameters

CCHandle (input)
The handle to the cryptographic context that will be used.

EncryptionProhibited (input)
The usability field for law enforcement and enterprise key recovery. Possible values are:

- KR_LE - Signifies that law enforcement key recovery enablement needs to be done.
- KR_ENT - Signifies that enterprise key recovery enablement is required.

Workfactor (output)
The maximum permissible workfactor value that may be used for law enforcement key recovery.

Return Values

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

5.4 Key Recovery Registration Operations

5.4.1 CSSM_KR_RegistrationRequest

CSSM_RETURN CSSMAPI CSSM_KR_RegistrationRequest
(CSSM_CC_HANDLE
RecoveryRegistrationContext,
CSSM_DATA_PTR KRInData,
CSSM_CRYPTO_DATA_PTR UserCallback,
uint8 KRFlags,
unit32 *EstimatedTime
CSSM_HANDLE_PTR ReferenceHandle)

This function performs a key recovery registration operation. The *KRInData* parameter contains known input parameters for the recovery registration operation. A UserCallback function may be supplied to allow the registration operation to interact with the user interface, if necessary. When this operation is successful, a *ReferenceHandle* and an *EstimatedTime* parameter are returned; the *ReferenceHandle* will be used to invoke the CSSM_KR_RegistrationRetrieve function, after the *EstimatedTime* in seconds.

Parameters

RecoveryRegistrationContext (input)
The handle to the key recovery registration context.

KRInData(input)
Input data for key recovery registration.

UserCallback (input)
A callback function that may be used to collect further information from the user interface.

KRFlags (input)
Flag values for recovery registration. Defined values are:

- KR_INDIVIDUAL - signifies that the registration is for the IND scenario
- KR_ENT - signifies that the registration is for the ENT scenario
- KR_LE - signifies that the registration is for the LE scenario

EstimatedTime (output)
The estimated time after which the CSSM_KR_RegistrationRetrieve call should be invoked to obtain registration results.

ReferenceHandle (output)
The handle to use to invoke the CSSM_KR_RegistrationRetrieve function.

Return Values

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

5.4.2 CSSM_KR_RegistrationRetrieve

CSSM_RETURN CSSMAPI CSSM_KR_RegistrationRetrieve
(CSSM_KRSP_HANDLE hKRSP,
CSSM_HANDLE ReferenceHandle,
unit32 *EstimatedTime,
CSSM_KR_PROFILE_PTR KRProfile)

This function completes a key recovery registration operation. The results of a successful registration operation are returned through the *KRProfile* parameter, which may be used with the profile management API functions.

If the results are not available when this function is invoked, the *KRProfile* parameter is set to NULL, and the *EstimatedTime* parameter indicates when this operation should be repeated with the same *ReferenceHandle*.

Parameters

hKRSP (input)
The handle to the KRSP that will be used.

ReferenceHandle (input)
The handle to the key recovery registration request that will be completed.

EstimatedTime (output)
The estimated time after which this call should be repeated to obtain registration results. This is set to a non-zero value only when the KRProfile parameter is NULL.

KRProfile (input/output)
Key recovery profile that is filled in by the registration operation.

Return Values

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

5.5 Key Recovery Enablement Operations

5.5.1 CSSM_KR_GenerateRecoveryFields

CSSM_RETURN CSSMAPI CSSM_KR_GenerateRecoveryFields
(CSSM_CC_HANDLE hKRContext,
CSSM_CC_HANDLE CryptoContext,
CSSM_DATA_PTR KRSPOptions,
uint32 KRFlags,
CSSM_DATA_PTR KRFields)

This function generates the Key Recovery Fields (KRFs) for a cryptographic association given the key recovery context, the session specific key recovery attributes, and the handle to the cryptographic context containing the key that will be made recoverable. The session attributes and the flags are not interpreted at the KeyWorks layer. If this call returns successfully, and the caller possesses the CSSM_STRONG_CRYPTO_WITH_KR privilege, the EncryptionProhibited flags within the CryptoContext may be modified, allowing the CryptoContext handle to be used for the KeyWorks encrypt APIs. The generated KRFs are returned as an output parameter. The KRFlags parameter may be used to fine tune the contents of the KRFields produced by this operation.

Parameters

hKRContext (input)
The handle to the key recovery context for the cryptographic association.

CryptoContext (input)
The cryptographic context handle that points to the session key.

KRSPOptions (input)
The Key Recovery Service Provider (KRSP) specific options. These options are uninterpreted by the KeyWorks Framework, but passed on to the KRSP.

KRFlags (input)
Flag values for KRFs generation. Defined values are:

- KR_INDL - Signifies that only the individual KRFs should be generated.
- KR_ENT - Signifies that only the enterprise KRFs should be generated.
- KR_LE - Signifies that only the law enforcement KRFs should be generated.
- KR_ALL - Signifies that law enforcement, enterprise, and individual KRFs should be generated.
- KR_OPTIMIZE - Signifies that performance optimization options are to be adopted by a KRSP while implementing this operation.
- KR_DROP_WORKFACTOR - Signifies that the law enforcement portion of the KRFs should be generated without using the key size workfactor.

KRFields (output)
The KRFs in the form of an uninterpreted data blob.

Return Values

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_RequestCssmExemption

5.5.2 CSSM_KR_ProcessRecoveryFields

CSSM_RETURN CSSMAPI CSSM_KR_ProcessRecoveryFields
(CSSM_CC_HANDLE KeyRecoveryContext,
CSSM_CC_HANDLE CryptoContext,
CSSM_DATA_PTR KRSPOptions,
uint32 KRFlags,
CSSM_DATA_PTR KRFields)

This function processes a set of KRFs given the key recovery context and the cryptographic context for the decryption operation. If the call is successful, and the caller possesses the CSSM_STRONG_CRYPTO_WITH_KR privilege, the EncryptionProhibited flags within the CryptoContextmay be modified, allowing the CryptoContext handle to be used for the KeyWorks decrypt API calls.

Parameters

KeyRecoveryContext (input)
The handle to the key recovery context.

CryptoContext (input)
A handle to the cryptographic context for which the KRFs are to be processed.

KRSPOptions (input)
The KRSP specific options. These options are uninterpreted by the IBM KeyWorks Framework, but passed on to the KRSP.

KRFlags (input)
Flag values for KRFs processing. Defined values are:

- KR_ENT - Signifies that only the enterprise KRFs should be processed.
- KR_LE - Signifies that only the law enforcement KRFs should be processed.
- KR_ALL - Signifies that only the enterprise KRFs should be processed.
- KR_OPTIMIZE - Signifies that performance optimization options will be adopted by a KRSP while implementing this operation.

KRFields (input)
The KRFs to be processed.

Return Values

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

See Also

CSSM_RequestCssmExemption

5.6 Key Recovery Request Operations

5.6.1 CSSM_KR_GetRecoveredObject

CSSM_RETURN CSSMAPI CSSM_KR_GetRecoveredObject
(CSSM_KRSP_HANDLE KRSPHandle,
CSSM_HANDLE_PTR CacheHandle,
unit32 IndexInResults,
CSSM_CSP_HANDLE CSPHandle,
const CSSM_CRYPTO_DATA_PTR PassPhrase,
CSSM_KEY_PTR RecoveredKey,
unit32 Flags,
CSSM_DATA_PTR OtherInfo)

This function is used to step through the results of a recovery request operation in order to retrieve a single recovered key at a time along with its associated meta-information. The cache handle returned from a successful CSSM_KR_RecoveryRetrieve operation is used. When multiple keys are recovered by a single recovery request operation, the *IndexInResults* parameter indicates which item to retrieve through this function.

The *RecoveredKey* parameter serves as an input template for the key to be returned. If a private key is to be returned by this operation, the *PassPhrase* parameter is used to inject the private key into the CSP indicated by the *RecoveredKey* template; the corresponding public key is returned in the *RecoveredKey* parameter. Subsequently, the *PassPhrase* and the public key may be used to reference the private key when operations using the private key are required. The *OtherInfo* parameter may be used to return other meta-data associated with the recovered key.

Parameters

KRSPHandle (input)
The handle to the KRSP that is to be used.

CacheHandle (input)
Pointer to the handle returned from a successful CSSM_KR_RecoveryRequest operation.

IndexInResults (input)
The index into the results that are referenced by the *ResultsHandle* parameter.

CSPHandle (input/optional)
This parameter identifies the CSP that the recovered key should be injected into. It may be set to NULL if the key is to be returned in raw form to the caller.

PassPhrase (input)
This parameter is only relevant if the recovered key is a private key. It is used to protect the private key when it is inserted into the CSP specified by the *RecoveredKey* template.

RecoveredKey (output)
This parameter returns the recovered key.

Flags (input)
Flag values relevant for recovery of a key. Possible values are:

- CERT_RETRIEVE - If the recovered key is a private key, return the corresponding public key certificate in the *OtherInfo* parameter.

OtherInfo (output)
This parameter is used if there are additional information associated with the recovered key (such as the public key certificate when recovering a private key) that will be returned.

Return Values

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

5.6.2 CSSM_KR_RecoveryRequest

CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRequest
(CSSM_CC_HANDLE RecoveryRequestContext,
const CSSM_DATA_PTR KRInData,
const CSSM_CRYPTO_DATA_PTR UserCallback,
unit32 *EstimatedTime,
const CSSM_HANDLE_PTR ReferenceHandle)

This function performs a key recovery request operation. The *KRInData* contains known input parameters for the recovery request operation. A *UserCallback* function may be supplied to allow the recovery operation to interact with the user interface, if necessary. If the recovery request operation is successful, a *ReferenceHandle* and an *EstimatedTime* parameter are returned; the *ReferenceHandle* will be used to invoke the CSSM_KR_RecoveryRetrieve function, after the *EstimatedTime* in seconds.

Parameters

RecoveryRequestContext (input)
The handle to the key recovery request context.

KRInData (input)
Input data for key recovery requests. For encapsulation schemes, the KRFs are included in this parameter.

UserCallback (input)
A callback function that may be used to collect further information from the user interface.

EstimatedTime (output)
The estimated time after which the CSSM_KR_RecoveryRetrieve call should be invoked to obtain recovery results.

ReferenceHandle (output)
Handle returned when recovery request is successful. This handle may be used to invoke the CSSM_KR_RecoveryRetrieve function.

Return Values

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

5.6.3 CSSM_KR_RecoveryRequestAbort

CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRequestAbort
(CSSM_KR_HANDLE KRSPHandle,
CSSM_HANDLE CacheHandle)

Description

This function terminates a recovery request operation and releases any state information related to the recovery request.

Parameters

KRSPHandle (input)
The handle to the KRSP that is to be used.

CacheHandle (input)
The handle returned from a successful CSSM_KR_RecoveryRequest operation.

Return Values

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

5.6.4 CSSM_KR_RecoveryRetrieve

CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRetrieve
(CSSM_KRSP_HANDLE KRSPHandle,
CSSM_HANDLE_PTR ReferenceHandle,
unit32 *EstimatedTime,
CSSM_HANDLE_PTR CacheHandle,
unit32 *NumberOfRecoveredKeys)

This function completes a key recovery request operation. The *ReferenceHandle* parameter indicates which outstanding recovery request is to be completed. The results of a successful recovery operation are referenced by the *ResultsHandle* parameter, which may be used with the CSSM_KR_GetRecoveredObject function to retrieve the recovered keys.

If the results are not available at the time this function is invoked, the *CacheHandle* is NULL, and the *EstimatedTime* parameter indicates when this operation should be repeated with the same *ReferenceHandle*.

Parameters

KRSPHandle (input)
The handle to the KRSP that is to be used.

ReferenceHandle (input)
Handle that indicates which key recovery request operation is to be completed.

EstimatedTime (output)
The estimated time after which this call should be repeated to obtain recovery results. This is set to a non-zero value only when the *ResultsHandle* parameter is NULL.

CacheHandle (output)
Handle returned when recovery operation is successful. This handle may be used to get individual keys using the CSSM_KR_GetRecoveredObject function. This handle is NULL, if the *EstimatedTime* parameter is not zero.

NumberOfRecoveredKeys (output)
The number of recovered key objects that may be obtained using the *ResultsHandle*.

Return Values

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

5.6.5 CSSM_KR_QueryPolicyInfo

CSSM_RETURN CSSMAPI CSSM_KR_QueryPolicyInfo
(const uint32 AlgorithmID,
const uint32 Mode,
const uint32 Class,
CSSM_POLICY_INFO_PTR *PolicyInfoData)

This function queries the law enforcement CSSM policy in effect and returns relevant information for application use. No privilege is required to invoke this function.

The policy information reports the maximum key length that can be generated, per cipher algorithm type and mode, without a need to generate key recovery blocks. It also specifies whether it is the jurisdiction of manufacturing or the jurisdiction of use to enforce the given policy. For special situations where the jurisdiction of use prohibits generation of key recovery fields, that information will also be provided.

Applications can request policy information relative to a specific algorithm, by providing the CSSM algorithm identifier in first parameter to the call. If a CSSM_ALGID_NONE is provided in this field, the PolicyInfoData will contain information pertaining to the entire set of algorithms controlled for the law enforcement jurisdiction. The Mode parameter can be specified exactly, or set to CSSM_ALGMODE_NONE. In the latter case, all matching algorithm ids regardless of the actual mode are returned. The class parameter should be set to correctly to symmetric or asymmetric, otherwise the results will not be accurate.

If the API can not find a matching entry in the configured policies, the numberOfEntries field in PolicyInfoData is set to 0, and the return code to CSSM_OK. If the return code is set to CSSM_FAIL, there was an internal error that can be retrieved using CSSM_GetError API function.

Applications have the responsibility to free the memory associated with the policy information data when no longer needed.

Parameters

PolicyInfoData (input/output)
Pointer to a CSSM policy information data structure to receive the query results.

AlgorithmID (input)
CSSM defined algorithm identifier for which policy information is requested. This parameter must be CSSM_ALGID_NONE if global policy information is desired.

Mode (input)
The desired algorithm mode. This parameter can be set to CSSM_ALGMODE_NONE to get all applicable modes.

Class (input)
The class of the desired algorithm. The allowed values are CSSM_ALGCLASS_ASYMMETRIC and CSSM_ALGCLASS_SYMMETRIC.

Return Values

This function returns CSSM_OK if a privilege set was successfully retrieved. On error CSSM_FAIL is returned. Use CSSM_GetError to obtain the error code.

Chapter 6. Trust Policy Services API

The primary purpose of a Trust Policy (TP) module is to answer the question, "Is this certificate authorized for this action in this trust domain?" Applications are executed within some trust domain. For example, executing an installation program at the office takes place within the corporate information technology trust domain. Executing an installation program on a system at home takes place within the user's personal system trust domain. The TP that allows or blocks the installation action is different for the two domains. The corporate domain may require extensive credentials and accept only credentials signed by selected parties. The personal system domain may require only a credential that establishes the bearer as a known user on the local system.

The general KeyWorks trust model defines a set of basic trust objects that most (if not all) TPs use to model their trust domain and the policies over that domain. These basic trust objects include:

- Policies
- Certificates
- Defined sources of trust
- Certificate Revocation Lists
- Application-specific actions
- Evidence

Policies define the credentials required for authorization to perform an action on another object. For example, a system administrator policy controls creating new user accounts on a computer system. Certificates are the basic credentials representing a trust relationship among a set of two or more parties. When an organization issues certificates, it defines its issuing procedure in a Certification Practice Statement (CPS). The statement identifies existing policies with which it is consistent. The statement also can be the source of new policy definitions if the action and target object domains are not covered by an existing, published policy. An application domain can recognize multiple policies. A given policy can be recognized by multiple application domains.

Evaluation of trust depends on relationships among certificates. For example, certificate chains represent hierarchical trust, where a root authority is the source of trust. Entities attain a level of trust based on their relationship to the root authority. Certificate graphs represent an introducer model of trust, where the number and strength of endorsers (i.e., immediate links in the graph) increases the level of trust attained by an entity. In both models, the trust domain can define accepted sources of trust. These may be mandated by fiat or can be computed by some other means. In contrast to the sources of trust, Certificate Revocation Lists (CRLs) represent sources of distrust. TPs may consult these lists during the verification process.

Trust evaluation can be performed with respect to a specific action the bearer wishes to perform, with respect to a policy, or with respect to the application domain in general. In the latter case, the action is understood to be either one specific action, or all actions in the domain.

When verifying trust, a TP module processes a group of certificates. The result of verification is a list of evidence, which forms an audit trail of the process. The evidence may be a list of verified attribute values that were contained in the certificates, or the entire set of verified certificates, or some other information that serves as evidence of the verification. In the end, the trust and authorizations asserted are based on the authority implied by a set of assumed or otherwise specified public keys.

Many applications know a priori the TP modules it must use. The KeyWorks registry and query mechanism provides applications access to TP module descriptions. This information is provided by the TP module during installation and can assist the application in selecting the appropriate TP module for a given application domain.

6.1 Data Structures

6.1.1 CSSM_REVOKE_REASON

This data structure represents the reason a certificate is being revoked.

```
typedef enum cssm_revoke_reason {
   CSSM_REVOKE_CUSTOM = 0,
   CSSM_REVOKE_UNSPECIFIC = 1,
   CSSM_REVOKE_KEYCOMPROMISE = 2,
   CSSM_REVOKE_CACOMPROMISE = 3,
   CSSM_REVOKE_AFFILIATIONCHANGED = 4,
   CSSM_REVOKE_SUPERCEDED = 5,
   CSSM_REVOKE_CESSATIONOFOPERATION = 6,
   CSSM_REVOKE_CERTIFICATEHOLD = 7,
   CSSM_REVOKE_CERTIFICATEHOLDRELEASE = 8,
   CSSM_REVOKE_REMOVEFROMCRL = 9
} CSSM_REVOKE_REASON;
```

6.1.2 CSSM_TP_ACTION

This data structure represents a descriptive value defined by the TP module. A TP can define application-specific actions for the application domains over which the TP applies. Given a set of credentials, the TP module verifies authorizations to perform these actions.

```
typedef uint32 CSSM_TP_ACTION
```

6.1.3 CSSM_TP_HANDLE

This data structure represents the TP module handle. The handle value is a unique pairing between a TP module and an application that has attached that module. TP handles can be returned to an application as a result of the CSSM_ModuleAttach function.

```
typedef uint32 CSSM_TP_HANDLE/* Trust Policy Handle */
```

6.1.4 CSSM_TP_STOP_ON

This enumerated list defines the conditions controlling termination of the verification process by the TP module when a set of policies/conditions must be tested.

```
typedef enum cssm_tp_stop_on {
   CSSM_TP_STOP_ON_POLICY = 0,    /* use the pre-defined stopping criteria */
   CSSM_TP_STOP_ON_NONE = 1,      /* evaluate all condition whether T or F */
   CSSM_TP_STOP_ON_FIRST_PASS = 2, /* stop evaluation at first TRUE */
   CSSM_TP_STOP_ON_FIRST_FAIL = 3/* stop evaluation at first FALSE */
} CSSM_TP_STOP_ON;
```

6.1.5 CSSM_TPSUBSERVICE

Three structures are used to contain all of the static information that describes a TP module: cssm_moduleinfo, cssm_serviceinfo, and cssm_tpsubservice. This descriptive information is securely stored in the KeyWorks registry when the TP module is installed with KeyWorks. A TP module may implement multiple types of services and organize them as subservices. For example, a TP module supporting electronic transaction applications may organize its implementation into three subservices: one for micro-cash payments from an electronic wallet, a second for payments by credit card, and a third for payments by bank debit card. Most TP modules will implement exactly one subservice.

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo and specifying the trust policy module GUID.

```
typedef struct cssm_tpsubservice {
   uint32 SubServiceId;
   CSSM_STRING Description;
   CSSM_CERT_TYPE CertType;
   CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
   uint32 NumberOfPolicyIdentifiers;
   CSSM_FIELD_PTR PolicyIdentifiers;
   CSSM_TP_WRAPPEDPRODUCT_INFO WrappedProduct;
} CSSM_TPSUBSERVICE, *CSSM_TPSUBSERVICE_PTR;
```

Definitions:

SubServiceId - A unique, identifying number for the subservice described in this structure.

Description - A string containing a descriptive name or title for this subservice.

CertType - Type of certificate accepted by the TP module.

AuthenticationMechanism - An enumerated value defining the credential format accepted by the TP module. An authentication credential is required for some TP functions. Presented credentials must be of the required format.

NumberOfPolicyIdentifiers - The number of policies supported by this TP module.

PolicyIdentifiers - A list of the policies (represented by their identifiers) supported by this TP module. There must be *NumberOfPolicyIdentifiers* entries in this list.

WrappedProduct - Pointer to wrapped product information.

6.1.6 CSSM_TP_WRAPPEDPRODUCTINFO

```
typedef struct cssm_tp_wrappedproductinfo {
   CSSM_VERSION StandardVersion;
   CSSM_STRING StandardDescription;
   CSSM_STRING ProductVendor;
   uint32 ProductFlags;
} CSSM_TP_WRAPPEDPRODUCT_INFO, *CSSM_TP_WRAPPEDPRODUCT_INFO_PTR;
```

Description:

StandardVersion - Version of standard to which this product conforms.

StandardDescription - Description of standard to which this product conforms.

ProductVendor - Vendor of wrapped product/library.

ProductFlags - ProductFlags.

6.2 Trust Policy Operations

6.2.1 CSSM_TP_ApplyCrlToDb

CSSM_RETURN CSSMAPI CSSM_TP_ApplyCrlToDb (CSSM_TP_HANDLE TPHandle,
CSSM_CL_HANDLE CLHandle,
CSSM_CSP_HANDLE CSPHandle,
const CSSM_DB_LIST_PTR DBList,
const CSSM_DATA_PTR Crl)

This function updates persistent storage to reflect entries in the CRL. The TP module determines whether the memory-resident CRL is trusted, and if it should be applied to one or more of the persistent databases. Side effects of this function can include saving a persistent copy of the CRL in a data store or removing certificate records from a data store.

Parameters

TPHandle (input)
The handle that describes the TP module used to perform this function.

CLHandle (input/optional)
The handle that describes the Certificate Library (CL) module that can be used to manipulate the CRL as it is applied to the data store and to manipulate the certificates effected by the CRL, if required. If no CL module is specified, the TP module uses an assumed CL module, if required.

CSPHandle (input/optional)
The handle referencing a Cryptographic Service Provider (CSP) to be used to verify signatures on the CRL determining whether to trust the CRL and apply it to the data store. The TP module is responsible for creating the cryptographic context structures required to perform the verification operation. If no CSP is specified, the TP module uses an assumed CSP to perform these operations.

DBList (input/optional)
A list of handle pairs specifying a Data Storage Library (DL) module and a data store managed by that module. These data stores can contain certificates that might be effected by the CRL, they may contain CRLs, or both. If no DL and database (DB) handle pairs are specified, the TP module must use an assumed DL module and an assumed data store for this operation.

Crl (input)
A pointer to the CSSM_DATA structure containing a CRL to be applied to the data store.

Return Value

A CSSM_OK return value signifies that the revocations contained in the CRL have been appropriately applied to the specified database. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CrlGetFirstItem, CSSM_CL_CrlGetNextItem, CSSM_DL_CertRevoke

6.2.2 CSSM_TP_CertRevoke

CSSM_DATA_PTR CSSMAPI CSSM_TP_CertRevoke
(CSSM_TP_HANDLE TPHandle,
CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
const CSSM_DB_LIST_PTR DBList,
const CSSM_DATA_PTR OldCrl,
const CSSM_CERTGROUP_PTR CertGroupToBeRevoked,
const CSSM_CERTGROUP_PTR RevokerCertGroup,
CSSM_REVOKE_REASON Reason)

This function updates a CRL. The TP module determines whether the revoking certificate can revoke the target certificates. If authorized, a CRL record is added to the CRL and returned to the caller.

Parameters

TPHandle (input)
The handle that describes the TP module used to perform this function.

CLHandle (input/optional)
The handle that describes the CL module that can be used to manipulate the certificates targeted for revocation and the revoker's certificates. If no CL module is specified, the TP module uses an assumed CL module, if required.

CCHandle (input)
The handle that describes the context for a cryptographic operation. The cryptographic context specifies the handle of the CSP that must be used to perform the operation.

DBList (input/optional)
A list of handle pairs specifying a DL module and a data store managed by that module. These data stores can be used to store or retrieve objects (such as certificate and CRLs) related to the subject certificate and revoker's certificate. If no DL and DB handle pairs are specified, the TP module can uses an assumed DL module and an assumed data store, if required.

OldCrl (input/optional)
A pointer to the CSSM_DATA structure containing an existing CRL. If this input is NULL, a new list is created.

CertGroupToBeRevoked (input)
A pointer to the CSSM_CERTGROUP structure containing one or more related certificates to be revoked.

RevokerCertGroup (input)
A pointer to the CSSM_CERTGROUP structure containing the certificate used to revoke the target certificates.

Reason (input)
The reason for revoking the target certificates.

Return Value

A pointer to the CSSM_DATA structure containing the updated CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

6.2.3 CSSM_TP_CertSign

CSSM_DATA_PTR CSSMAPI CSSM_TP_CertSign
(CSSM_TP_HANDLE TPHandle,
CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
const CSSM_DB_LIST_PTR DBList,
const CSSM_DATA_PTR CertToBeSigned,
const CSSM_CERTGROUP_PTR SignerCertGroup,
const CSSM_FIELD_PTR SignScope,
uint32 ScopeSize)

This function signs a certificate and enforces a specific signing policy, such as X.509, or another standard that the TP module supports.

Parameters

TPHandle (input)
The handle that describes the TP module used to perform this function.

CLHandle (input/optional)
The handle that describes the CL module that can be used to manipulate the certificate to be signed. If no CL module is specified, the TP module uses an assumed CL module, if required.

CCHandle (input)
The handle that describes the context for a cryptographic operation. The cryptographic context specifies the handle of the CSP that must be used to perform the operation.

DBList (input/optional)
A list of handle pairs specifying a DL module and a data store managed by that module. These data stores can be used to store, retrieve objects (such as certificate and CRLs) related to the signer's certificate, or a data store for storing a resulting signed CRL. If no DL and DB handle pairs are specified, the TP module can use an assumed DL module and an assumed data store, if required.

CertToBeSigned (input)
A pointer to the CSSM_DATA structure containing a certificate to be signed.

SignerCertGroup (input)
A pointer to the CSSM_CERTGROUP structure containing one or more related certificates used to sign the certificate.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the tags of the certificate fields to be included in the signing process. If the signing scope is null, the TP Module must assume a default scope (portions of the certificate to be hashed) when performing the signing process.

ScopeSize (input)
The number of entries in the sign scope list. If the signing scope is not specified, the input parameter value for scope size must be zero.

Return Values

A pointer to a CSSM_DATA structure containing the signed certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

6.2.4 CSSM_TP_CrlSign

CSSM_DATA_PTR CSSMAPI CSSM_TP_CrlSign
(CSSM_TP_HANDLE TPHandle,
CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
const CSSM_DB_LIST_PTR DBList,
const CSSM_DATA_PTR CrlToBeSigned,
const CSSM_CERTGROUP_PTR SignerCertGroup,
const CSSM_FIELD_PTR SignScope,
uint32 ScopeSize)

This function signs a CRL. The TP module determines whether the signer's certificate is trusted to sign the CRL. If trust is satisfied, then the TP module has the option to carry out the service or to return a permission status without performing the service. This allows the library to support external as well as internal CRL service models. In either model, once a CRL is signed, revocation records can no longer be added to that CRL. To do so, would break the integrity of the signature resulting in a non-verifiable, rejected CRL.

Parameters

TPHandle (input)
The handle that describes the TP module used to perform this function.

CLHandle (input/optional)
The handle that describes the CL module that can be used to manipulate the certificates to be signed. If no CL module is specified, the TP module uses an assumed CL module, if required.

CCHandle (input)
The handle that describes the context for a cryptographic operation. The cryptographic context specifies the handle of the CSP and other cryptographic parameters that must be used to perform the operation.

DBList (input/optional)
A list of handle pairs specifying a DL module and a data store managed by that module. These data stores can be used to store, retrieve objects (such as certificate and CRLs) related to the signer's certificate, or a data store for storing a resulting signed CRL. If no DL and DB handle pairs are specified, the TP module can use an assumed DL module and an assumed data store, if required.

CrlToBeSigned (input)
A pointer to the CSSM_DATA structure containing a CRL to be signed.

SignerCertGroup (input)
A pointer to the CSSM_CERTGROUP structure containing one or more related certificates used to sign the CRL.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the tags of the CRL fields to be included in the signing process. If the signing scope is null, the TP Module must assume a default scope (portions of the CRL to be hashed) when performing the signing process.

ScopeSize (input)
The number of entries in the sign scope list. If the signing scope is not specified, the input parameter value for scope size must be zero.

Return Value

A pointer to the CSSM_DATA structure containing the signed CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

6.2.5 CSSM_TP_CrlVerify

CSSM_BOOL CSSMAPI CSSM_TP_CrlVerify
(CSSM_TP_HANDLE TPHandle,
CSSM_CL_HANDLE CLHandle,
CSSM_CSP_HANDLE CSPHandle,
const CSSM_DB_LIST_PTR DBList,
const CSSM_DATA_PTR CrlToBeVerified,
const CSSM_CERTGROUP_PTR SignerCertGroup,
const CSSM_FIELD_PTR VerifyScope,
uint32 ScopeSize)

This function determines whether the CRL is trusted. The conditions for trust are part of the TP module. It can include conditions such as validity of the signer's certificate, verification of the signature on the CRL, the identity of the signer, the identity of the sender of the CRL, the date the CRL was issued, the effective dates on the CRL, etc.

Parameters

TPHandle (input)
The handle that describes the TP module used to perform this function.

CLHandle (input/optional)
The handle that describes the CL module that can be used to manipulate the certificates to be verified. If no CL module is specified, the TP module uses an assumed CL module, if required.

CSPHandle (input/optional)
The handle referencing a CSP to be used to verify signatures on the signer's certificate and on the CRL. The TP module is responsible for creating the cryptographic context structure required to perform the verification operation. If no CSP is specified, the TP module uses an assumed CSP to perform the operations.

DBList (input/optional)
A list of handle pairs specifying a DL module and a data store managed by that module. These data stores can be used to store or retrieve objects (such as certificate and CRLs) related to the signer's certificate. If no DL and DB handle pairs are specified, the TP module can use an assumed DL module and an assumed data store, if required.

CrlToBeVerified (input)
A pointer to the CSSM_DATA structure containing a signed CRL to be verified.

SignerCertGroup (input)
A pointer to the CSSM_CERTGROUP structure containing one or more related certificates used to sign the CRL.

VerifyScope (input/optional)
A pointer to the CSSM_FIELD array indicating the CRL fields to be included in the CRL signature verification process. A null input verifies the signature assuming the module’s default sets of fields were used in the signaturing process (this can include all fields in the CRL).

ScopeSize (input)
The number of entries in the verify scope list. If the verification scope is not specified, the input parameter value for scope size must be zero.

Return Value

A CSSM_TRUE return value signifies that the CRL can be trusted. When CSSM_FALSE is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

6.3 Group Functions

6.3.1 CSSM_TP_CertGroupConstruct

CSSM_CERTGROUP_PTR CSSMAPI CSSM_TP_CertGroupConstruct
(CSSM_TP_HANDLE TPHandle,
CSSM_CL_HANDLE CLHandle,
CSSM_CSP_HANDLE CSPHandle,
CSSM_CERTGROUP_PTR CertGroupFrag,
CSSM_DB_LIST_PTR DBList)

This function constructs an ordered certificate group using the certificates in *CertGroupFrag* as a starting point. There is no implied ordering for the certificates in *CertGroupFrag* except that the certificate in position 0 of the certificate group is assumed to the starting point for constructing the remaining certificate group. An ordering relationship may be defined and recorded in the certificates themselves or assumed by the TP model.

The certificate group is augmented by adding semantically related certificates obtained by searching the certificate data stores specified in *DBList*. For example, if the certificate model is a hierarchical model of certificate chains, the leaf certificate in the chain is a CertGroup fragment and the complete certificate chain, including the root certificate, is the anticipated result of the construction operation.

Parameters

TPHandle (input)
The handle to the TP module to perform this operation.

CLHandle (input/optional)
The handle to the CL module that can be used to manipulate and parse values in stored in the certgroup certificates. If no CL module is specified, the TP module uses an assumed CL module.

CSPHandle(input)
The handle to the CSP that can be used for verification of certificate chain while constructing the certificate group.

CertGroupFrag (input)
A list of certificates that form a possibly incomplete set of certificates. This set is used as the base set for constructing a complete certificate group.

DBList (input)
A list of handle pairs specifying a DL module and a data store managed by that module. These data stores should contain certificates (and possibly other security objects). The data stores should be searched to complete construction of a semantically related certificate group.

Return Value

A list of certificates that form a complete certificate group based on the original subset of certificates and the certificate data stores. A NULL list indicates an error.

See Also

CSSM_TP_CertGroupPrune, CSSM_TP_CertGroupVerify

6.3.2 CSSM_TP_CertGroupPrune

CSSM_CERTGROUP_PTRCSSMAPI CSSM_TP_CertGroupPrune
(CSSM_TP_HANDLE TPHandle,
CSSM_CL_HANDLE CLHandle
CSSM_CERTGROUP_PTR OrderedCertGroup,
CSSM_DB_LIST_PTR DBList)

This function removes certificates from a certificate group. The prune operation can remove those certificates that have been signed by any local certificate authority, as it is possible that these certificates will not be meaningful on other systems.

This operation also can remove additional certificates that can be added to the certificate group again using the CSSM_CertGroupConstruct function. The pruned certificate group should be suitable for transmission to external hosts, which can in turn reconstruct and verify the certificate group.

The *DBList* parameter specifies a set of data stores containing certificates that should be pruned from the group.

Parameters

TPHandle (input)
The handle to the TP module to perform this operation.

CLHandle (input/optional)
The handle to the CL module that can be used to manipulate and parse the certgroup certificates and the certificates in the specified data stores. If no CL module is specified, the TP module uses an assumed CL module.

OrderedCertGroup (input)
The initial, complete set of certificates from which certificates will be selectively removed.

DBList (input)
A list of handle pairs specifying a DL module and a data store managed by that module. These data stores should contain certificates (and possibly other security objects also). The data stores are searched for certificates semantically related to those in the certificate group to determine whether they should be removed from the certificate group.

Return Value

Returns a certificate group containing those certificates which are verifiable credentials outside of the local system. If the list is NULL, an error has occurred.

See Also

CSSM_TP_CertGroupConstruct, CSSM_TP_CertGroupVerify

6.3.3 CSSM_TP_CertGroupVerify

CSSM_BOOL CSSMAPI CSSM_TP_CertGroupVerify
(CSSM_TP_HANDLE TPHandle,
CSSM_CL_HANDLE CLHandle,
CSSM_DB_LIST_PTR DBList,
CSSM_CSP_HANDLE CSPHandle,
const CSSM_FIELD_PTRPolicyIdentifiers,
uint32 NumberofPolicyIdentifiers,
CSSM_TP_STOP_ON VerificationAbortOn,
const CSSM_CERTGROUP_PTR CertToBeVerified,
const CSSM_DATA_PTR AnchorCerts,
uint32 NumberofAnchorCerts,
const CSSM_FIELD_PTR VerifyScope,
uint32 ScopeSize,
CSSM_TP_ACTION Action,
const CSSM_DATA_PTR Data,
CSSM_DATA_PTR *Evidence,
uint32 *EvidenceSize)

This function verifies the signatures on each certificate in the group. Each certificate in the group has an associated signing certificate that was used to sign the subject certificate. Determination of the associated signing certificate is implied by the certificate model. For example, when verifying an X.509 certificate chain, the signing certificate for a certificate C is known to be the certificate of the issuers of certificate C. In a multisignature, web of trust model, the signing certificates can be any certificates in the CertGroup or unknown certificates.

Signature verification is performed on the *VerifyScope* fields for all certificates in the *CertGroup*. Additional validation tests can be performed on the certificates in the group depending on the certificate model supported by the TP. For example, certificate expiration dates can be checked and appropriate CRLs can be searched as part of the verification process.

Parameters

TPHandle (input)
The handle to the TP module to perform this operation.

CLHandle (input/optional)
The handle to the CL module that can be used to manipulate and parse the certgroup certificates and the certificates in the specified data stores. If no CL module is specified, the TP module uses an assumed CL module.

DBList (input/optional)
A list of handle pairs specifying a DL module and a data store managed by that module. These data stores should contain zero or more trusted certificates. If no data stores are specified, the TP module can assume a default data store, if required.

CSPHandle (input)
The handle of a CSP that can be used for verification of the certificate chain.

PolicyIdentifiers (input/optional)
The policy identifier is an object identifier (OID)/value pair. The CSSM_OID structure contains the name of the policy and the value is an optional caller-specified input value for the TP module to use when applying the policy.

NumberofPolicyIdentifiers (input)
The number of policy identifiers provided in the *PolicyIdentifiers* parameter.

VerificationAbortOn (input/optional)
When a TP module verifies multiple conditions or multiple policies, the TP module can allow the caller to specify when to abort the verification process. If supported by the TP module, this selection can effect the evidence returned by the TP module to the caller. The default stopping condition is to stop evaluation according to the policy defined in the TP Module. The specifiable stopping conditions and their meaning are defined in Table 16.

Table 16. Specifiable Stopping Conditions

CSSM_TP_STOP_ON	**Definitions**
CSSM_STOP_ON_POLICY	Stop verification whenever the policy dictates it.
CSSM_STOP_ON_NONE	Stop verification only after all conditions have been tested (ignoring the pass-fail status of each condition).
CSSM_STOP_ON_FIRST_PASS	Stop verification on the first condition that passes.
CSSM_STOP_ON_FIRST_FAIL	Stop verification on the first condition that fails.

The TP module may ignore the caller's specified stopping condition and revert to the default of stopping according to the policy embedded in the module.

CertToBeVerified (input)
A pointer to the CSSM_CERTGROUP structure containing a certificate containing at least one signature for verification. An unsigned certificate template cannot be verified.

AnchorCerts (input/optional)
A pointer to the CSSM_DATA structure containing one or more certificates to be used in order to validate the subject certificate. These certificates can be root certificates, cross-certified certificates, and certificates belonging to locally designated sources of trust.

NumberofAnchorCerts (input)
The number of anchor certificates provided in the *AnchorCerts* parameter.

VerifyScope (input/optional)
A pointer to the CSSM_FIELD array containing the OID indicators specifying the certificate fields to be used in the verification process. If VerifyScope is not specified, the TP Module must assume a default scope (portions of each certificate) when performing the verification process.

ScopeSize (input)
The number of entries in the verify scope list. If the verification scope is not specified, the input scope size must be zero.

Action (input/optional)
An application-specific and application-defined action to be performed under the authority of the input certificate. If no action is specified, the TP module defines a default action and performs verification assuming that action is being requested. Note that it is possible that a TP module verifies certificates for only one action.

Data (input/optional)
A pointer to the CSSM_DATA structure containing the application-specific data or a reference to the application-specific data upon which the requested action should be performed. If no data is

specified, the TP module defines one or more default data objects upon which the action or default action would be performed.

Evidence (output/optional)
A pointer to a list of CSSM_DATA objects containing an audit trail of evidence constructed by the TP module during the verification process. Typically, this is a list of certificates and CRLs that were used to establish the validity of the *CertToBeVerified*, but other objects may be appropriate for other types of TPs.

EvidenceSize (output)
The number of entries in the *Evidence* list. The returned value is zero if no evidence is produced. *Evidence* may be produced even when verification fails. This evidence can describe why and how the operation failed to verify the subject certificate.

Return Value

CSSM_TRUE if the certificate group verified. CSSM_FALSE if the certificate did not verify or an error condition occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_TP_CertGroupConstruct, CSSM_TP_CertGroupPrune

6.4 Extensibility Functions

6.4.1 CSSM_TP_PassThrough

void * CSSMAPI CSSM_TP_PassThrough (CSSM_TP_HANDLE TPHandle,
CSSM_CL_HANDLE CLHandle,
CSSM_DL_HANDLE DLHandle,
CSSM_DB_HANDLE DBHandle,
CSSM_CC_HANDLE CCHandle,
uint32 PassThroughId,
const void *InputParams)

This function allows applications to call TP module-specific operations that have been exported. Such operations may include queries or services specific to the domain represented by the TP module.

Parameters

TPHandle (input)
The handle that describes the TP module used to perform this function.

CLHandle (input/optional)
The handle that describes the CL module that can be used to manipulate the subject certificate and anchor certificates. If no CL module is specified, the TP module uses an assumed CL module, if required.

DLHandle (input/optional)
The handle that describes the DL module that can be used to store or retrieve objects (such as certificate and CRLs) related to the subject certificate and anchor certificates. If no DL module is specified, the TP module uses an assumed DL module, if required.

DBHandle (input/optional)
The handle that describes the data store that can be accessed to store or retrieve objects (such as certificate and CRLs) related to the subject certificate and anchor certificates. If no data store is specified, the TP module uses an assumed data store, if required.

CCHandle (input/optional)
The handle that describes the context for a cryptographic operation. The cryptographic context specifies the handle of the CSP that must be used to perform the operation. If no cryptographic context is specified, the TP module uses an assumed context, if required.

PassThroughId (input)
An identifier assigned by the TP module to indicate the exported function to perform.

InputParams (input)
A pointer to a module, implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested TP module.

Return Value

A pointer to a module, implementation-specific structure containing the output from the passthrough function. The output data must be interpreted by the calling application based on externally available information. If the pointer is NULL, an error has occurred.

Chapter 7. Certificate Library Services API

The primary purpose of a Certificate Library (CL) module is to perform syntactic operations on a specific certificate format and its associated Certificate Revocation List (CRL) format. This encapsulation allows applications and TP modules to focus on the usage of certificates rather than the mechanics of format manipulation.

The syntactic operations on certificates include field management operations and cryptographic operations. Field management operations allow an application to input fields into a certificate and retrieve fields from a certificate without knowledge of the certificate's content organization or encoding format. Cryptographic operations on certificates encode the proper fields of a certificate in the proper order prior to executing certificate signing and verification.

The syntactic operations on CRLs mirror the operations on their corresponding certificate format. CRL field management operations allow the insertion and retrieval of CRL fields, including addition and removal of certificates from the revocation list. The CL module manages the translation from the certificate to be revoked to its representation in the CRL. The CL module also properly encodes the necessary fields of a CRL prior to signing and verification.

Each CL module may implement some or all of these functions on certificates and CRLs. The available functions are registered with KeyWorks when the module is attached. Each CL module should be accompanied with documentation specifying supported functions, nonsupported functions, and module-specific passthrough functions. It is the responsibility of the application developer to obtain and use this information when developing applications using a selected CL module.

A CL module's functionality may be partitioned, as appropriate, between the local client and a remote server. For example, a CL module may redirect the *CSSM_CL_CertSign* function to a Certificate Authority (CA) server application, but perform the *CSSM_CL_CertGetKeyInfo* function as a local operation.

CL modules manipulate memory-based objects only. The persistence of certificates, CRLs, and other security-related objects is an independent property of these objects. It is the responsibility of the application and/or the TP module to use data storage modules to make objects persistent (if appropriate).

7.1 Data Structures

This section describes the data structures that may be passed to or returned from a CL function. They will be used by applications to prepare data to be passed as input parameters into KeyWorks API function calls that will be passed without modification to the appropriate CL module. The CL module is then responsible for interpreting them and returning the appropriate data structure to the calling application via KeyWorks. These data structures are defined in the header file, cssmtype.h, which is distributed with KeyWorks.

7.1.1 CSSM_CA_SERVICES

This bit-mask defines the additional certificate-creation-related services that an issuing CA can offer. Such services include (but are not limited to) archiving the certificate and keypair, publishing the certificate to one or more certificate directory services, and sending automatic, out-of-band notifications of the need to renew a certificate. A CA may offer any subset of these services.

```
typedef uint32 CSSM_CA_SERVICES;

#define CSSM_CA_KEY_ARCHIVE       0x0001
#define CSSM_CA_CERT_PUBLISH      0x0002
#define CSSM_CA_CERT_NOTIFY_RENEW 0x0004
#define CSSM_CA_CRL_DISTRIBUTE    0x0008
```

7.1.2 CSSM_CERT_ENCODING

This variable specifies the certificate encoding format supported by a CL.

```
typedef enum cssm_cert_encoding {
   CSSM_CERT_ENCODING_UNKNOWN =0x00,
   CSSM_CERT_ENCODING_CUSTOM=0x01,
   CSSM_CERT_ENCODING_BER =0x02,
   CSSM_CERT_ENCODING_DER =0x03,
   CSSM_CERT_ENCODING_NDR =0x04,
} CSSM_CERT_ENCODING, *CSSM_CERT_ENCODING_PTR;
```

7.1.3 CSSM_CERTGROUP

This structure contains a set of certificates. It is assumed that the certificates are related based on cosignaturing. The certificate group is a syntactic representation of a trust model. All certificates in the group must be of the same type. Typically, the certificates are related in some manner, but this is not required.

```
typedef struct cssm_certgroup {
   uint32 NumCerts;
   CSSM_DATA_PTR CertList;
   void *reserved;
} CSSM_CERTGROUP, *CSSM_CERTGROUP_PTR;
```

Definitions:

NumCerts - Number of certificates in the group.

CertList - List of certificates.

reserved - Reserved for future use.

7.1.4 CSSM_CERT_TYPE

This variable specifies the type of certificate format supported by a CL and the types of certificates understood for import and export. They are expected to define such well-known certificate formats as X.509 Version 3 and Simple Distributed Security Infrastructure (SDSI) as well as custom certificate formats. The list of enumerated values can be extended for new types by defining a label with an associated value greater than CSSM_CL_CUSTOM_CERT_TYPE.

```
typedef enum cssm_cert_type {
   CSSM_CERT_UNKNOWN =0x00,
   CSSM_CERT_X_509v1 =0x01,
   CSSM_CERT_X_509v2 =0x02,
   CSSM_CERT_X_509v3 =0x03,
```

```
   CSSM_CERT_Fortezza = 0x07,
   CSSM_CERT_PGP =0x04,
   CSSM_CERT_SPKI = 0x05,
   CSSM_CERT_SDSIv1 = 0x06,
   CSSM_CERT_Intel =0x08,
   CSSM_CERT_ATTRIBUTE_BER = 0x09, /* ber encoded X.509 attribute cert */
   CSSM_CERT_X509_CRL = 0x10,
   CSSM_CERT_LAST = 0x7FFF
} CSSM_CERT_TYPE, *CSSM_CERT_TYPE_PTR;

/* Applications wishing to define their own custom certificate
 * type should create a random uint32 whose value is greater than
 * the CSSM_CL_CUSTOM_CERT_TYPE */
   CSSM_CL_CUSTOM_CERT_TYPE0x08000
```

7.1.5 CSSM_CL_CA_CERT_CLASSINFO

```
typedef struct cssm_cl_ca_cert_classinfo {
   CSSM_STRING CertClassName;
   CSSM_DATA CACert;
} CSSM_CL_CA_CERT_CLASSINFO, *CSSM_CL_CA_CERT_CLASSINFO_PTR;
```

Description:

CertClassName - Name of a certificate class issued by this certificate authority.

CACert - CA certificate for this cert class.

7.1.6 CSSM_CL_CA_PRODUCTINFO

This structure holds product information about a backend CA that is accessible to the CL module. The CL module vendor is not required to provide this information, but may choose to do so. For example, a CL module that implements upstream protocols to a particular type of commercial CA can record information about that CA service in this structure.

```
typedef struct cssm_cl_ca_productinfo {
   CSSM_VERSION StandardVersion;
   CSSM_STRING StandardDescription;
   CSSM_VERSION ProductVersion;
   CSSM_STRING ProductDescription;
   CSSM_STRING ProductVendor;
   CSSM_CERT_TYPE CertType;
   CSSM_CA_SERVICES AdditionalServiceFlags;
   uint32 NumberOfCertClasses;
   CSSM_CL_CA_CERT_CLASSINFO CertClassNames;
} CSSM_CL_CA_PRODUCTINFO, *CSSM_CL_CA_PRODUCTINFO_PTR;
```

Definitions:

StandardVersion - If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription - If this product conforms to an industry standard, this is a description of that standard.

ProductVersion - Version number information for the actual product version used in this version of the CL module.

ProductDescription - A string describing the product.

ProductVendor - The name of the product vendor.

CertType - An enumerated value specifying the certificate and CRL type that the CA manages.

AdditionalServiceFlags - A bit-mask indicating the additional services a caller can request from a CA (as side effects and in conjunction with other service requests).

NumberOfCertClasses - The number of classes or levels of certificates managed by this CA.

CertClassNames - Names of the certificate classes issued by this CA.

7.1.7 CSSM_CL_ENCODER_PRODUCTINFO

This structure holds product information about embedded products that a CL module uses to provide its services. The CL module vendor is not required to provide this information, but may choose to do so. For example, a CL module that manipulates X.509 certificates may embed a third-party tool that parses, encodes, and decodes those certificates. The CL module vendor can describe such embedded products using this structure.

```
typedef struct cssm_cl_encoder_productinfo {
   CSSM_VERSION StandardVersion;
   CSSM_STRING StandardDescription;
   CSSM_VERSION ProductVersion;
   CSSM_STRING ProductDescription;
   CSSM_STRING ProductVendor;
   CSSM_CERT_TYPE CertType;
   uint32 ProductFlags;
} CSSM_CL_ENCODER_PRODUCTINFO, *CSSM_CL_ENCODER_PRODUCTINFO_PTR;
```

Definitions:

StandardVersion - If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription - If this product conforms to an industry standard, this is a description of that standard.

ProductVersion - Version number information for the actual product version used in this version of the CL module.

ProductDescription - A string describing the product.

ProductVendor - The name of the product vendor.

CertType - An enumerated value specifying the certificate and CRL type that the CA manages.

ProductFlags - A bit-mask indicating any selectable features of the embedded product that the CL module selected for use.

7.1.8 CSSM_CL_HANDLE

The CSSM_CL_HANDLE is used to identify the association between an application thread and an instance of a CL module. It is assigned when an application causes KeyWorks to attach to a CL. It is freed when an application causes KeyWorks to detach from a CL. The application uses the CSSM_CL_HANDLE with every CL function call to identify the targeted CL. The CL module uses the CSSM_CL_HANDLE to identify the appropriate application's memory management routines when allocating memory on the application's behalf.

typedef uint32 CSSM_CL_HANDLE

7.1.9 CSSM_CLSUBSERVICE

Three structures are used to contain all of the static information that describes a CL module: cssm_moduleinfo, cssm_serviceinfo, and cssm_clsubservice. This descriptive information is securely stored in the KeyWorks registry when the CL module is installed with KeyWorks. A CL module may implement multiple types of services and organize them as subservices. For example, a CL module supporting X.509 encoded certificates may organize its implementation into three subservices: one for X.509 Version 1, a second for X.509 Version 2, and a third for X.509 Version 3. Most CL modules will implement exactly one subservice.

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo and specifying the CL module Globally Unique ID (GUID).

```
typedef struct cssm_clsubservice {
  uint32 SubServiceId;
  CSSM_STRING Description;
  CSSM_CERT_TYPE CertType;
  CSSM_CERT_ENCODING CertEncoding;
  CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
  uint32 NumberOfTemplateFields;
  CSSM_OID_PTR CertTemplates;
  uint32 NumberOfTranslationTypes;
  CSSM_CERT_TYPE_PTR CertTranslationTypes;
  CSSM_CL_WRAPPEDPRODUCT_INFO WrappedProduct;
} CSSM_CLSUBSERVICE, *CSSM_CLSUBSERVICE_PTR;
```

Definitions:

SubServiceId - A unique, identifying number for the subservice described in this structure.

Description - A string containing a description name or title for this subservice.

CertType - An identifier for the type of certificate. This parameter is also used to determine the certificate data format.

CertEncoding - An identifier for the certificate encoding format.

AuthenticationMechanism - An enumerated value defining the credential format accepted by the CL module. Authentication credential may be required when requesting certificate creation or other CL functions. Presented credentials must be of the required format.

NumberOfTemplateFields - The number of certificate fields. This number also indicates the length of the *CertTemplate* array.
CertTemplates - A pointer to an array of tag/value pairs which identify the field values of a certificate.

NumberOfTranslationTypes - The number of certificate types that this CL module can import and export. This number also indicates the length of the *CertTranslationTypes* array.

CertTranslationTypes - A pointer to an array of certificate types. This array indicates the certificate types that can be imported into and exported from this CL module's native certificate type.

WrappedProduct - A data structure describing the embedded products and CA service used by the CL module.

7.1.10 CSSM_CL_WRAPPEDPRODUCTINFO

This structure lists the set of embedded products and the CA service used by the CL module to implement its services. The CL module is not required to provide any of this information, but may choose to do so.

```
typedef struct cssm_cl_wrappedproductinfo {
   CSSM_CL_ENCODER_PRODUCTINFO_PTR EmbeddedEncoderProducts;
   uint32 NumberOfEncoderProducts;
   CSSM_CL_CA_PRODUCTINFO_PTR AccessibleCAProducts;
   uint32 NumberOfCAProducts;
} CSSM_CL_WRAPPEDPRODUCTINFO, *CSSM_CL_WRAPPEDPRODUCTINFO_PTR;
```

Definitions:

EmbeddedEncoderProducts - An array of structures that describe each embedded encoder product used in this CL module implementation.

NumberOfEncoderProducts - A count of the number of distinct embedded certificate encoder products used in the CL module implementation.

AccessibleCAProducts - An array of structures that describe each type of CA accessible through this CL module implementation.

NumberOfCAProducts - A count of the number of distinct CA products described in the array *AccessibleCAProducts*.

7.1.11 CSSM_FIELD

This structure contains the object identifier (OID)/value pair for any item that can be identified by an OID. A CL module uses this structure to hold an OID/value pair for a field in a certificate or CRL.

```
typedef struct cssm_field {
   CSSM_OID FieldOid;
   CSSM_DATA FieldValue;
}CSSM_FIELD, *CSSM_FIELD_PTR
```

Definitions:

FieldOid - The OID that identifies the certificate or CRL data type or data structure.

FieldValue - A CSSM_DATA type which contains the value of the specified OID in a contiguous block of memory.

7.1.12 CSSM_OID

The OID is used to hold an identifier for the data types and data structures that comprise the fields of a certificate or CRL. The underlying representation and meaning of the identifier is defined by the CL module. For example, a CL module can choose to represent its identifiers in any of the following forms:

- A character string in a character set native to the platform
- A DER-encoded X.509 OID that must be parsed
- An S-expression that must be evaluated
- An enumerated value that is defined in header files supplied by the CL module

typedef CSSM_DATA CSSM_OID, *CSSM_OID_PTR

7.2 Certificate Operations

7.2.1 CSSM_CL_CertAbortQuery

CSSM_RETURN CSSMAPI CSSM_CL_CertAbortQuery (CSSM_CL_HANDLE CLHandle,
CSSM_HANDLE ResultsHandle)

This function terminates the get operation initiated by CSSM_CL_CertGetFirstFieldValue or CSSM_CL_GetNextFieldValue, and allows the CL to release all intermediate state information associated with the query. This function should be called even if all values retrieved by the call to CSSM_CL_CertGetFirstFieldValue are obtained by repeated calls to CSSM_CL_CertGetNextFieldValue.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a get field value request.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertGetFirstFieldValue, CSSM_CL_CertGetNextFieldValue

7.2.2 CSSM_CL_CertCreateTemplate

CSSM_DATA_PTR CSSMAPI CSSM_CL_CertCreateTemplate
(CSSM_CL_HANDLE CLHandle,
const CSSM_FIELD_PTR CertTemplate,
uint32 NumberOfFields)

This function allocates and initializes memory for a certificate based on the input OID/value pairs specified in the *CertTemplate*. The initialization process includes encoding all certificate field values according to the format required by the certificate representation. The function returns the initialized template containing encoded values. The memory is allocated using the calling application's memory management routines.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

CertTemplate (input)
A pointer to an array of OID/value pairs that identify the field values to initialize a new certificate.

NumberOfFields (input)
The number of certificate field values specified in the CertTemplate.

Return Value

A pointer to the CSSM_DATA structure containing the unsigned certificate template. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertRequest, CSSM_CL_CertGetFirstFieldValue

7.2.3 CSSM_CL_CertDescribeFormat

CSSM_OID_PTR CSSMAPI CSSM_CL_CertDescribeFormat (CSSM_CL_HANDLE CLHandle,
uint32 *NumberOfFields)

This function returns a list of the object identifiers used to describe the certificate format supported by the specified CL.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

NumberOfFields (output)
The length of the returned array of OIDs.

Return Value

A pointer to the array of CSSM_OIDs that represent the supported certificate format. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertGetAllFields, CSSM_CL_CertGetFirstFieldValue, CSSM_CL_CertGetNextFieldValue, CSSM_CL_CertAbortQuery, CSSM_CL_CertGetKeyInfo

7.2.4 CSSM_CL_CertExport

CSSM_DATA_PTR CSSMAPI CSSM_CL_CertExport (CSSM_CL_HANDLE CLHandle,
CSSM_CERT_TYPE TargetCertType,
const CSSM_DATA_PTR NativeCert)

This function exports a certificate from the native format of the specified CL into the specified target certificate format. The set of *TargetCertTypes* supported for export varies with the CL module. See the documentation provided by the module vendor for a list of target certificate formats.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

TargetCertType (input)
A unique value which identifies the target type of the certificate being exported.

NativeCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be exported.

Return Value

A pointer to the CSSM_DATA structure containing the target-type certificate exported from the native certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertImport

7.2.5 CSSM_CL_CertGetAllFields

CSSM_FIELD_PTR CSSMAPI CSSM_CL_CertGetAllFields (CSSM_CL_HANDLE CLHandle,
const CSSM_DATA_PTR Cert,
uint32 *NumberOfFields)

This function returns a list of the values stored in the input certificate.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate whose fields will be returned.

NumberOfFields (output)
The length of the returned array of fields.

Return Value

A pointer to an array of CSSM_FIELD structures that contain the values of all of the fields of the input certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertGetFirstFieldValue, CSSM_CL_CertDescribeFormat, CSSM_CL_CertView

7.2.6 CSSM_CL_CertGetFirstFieldValue

CSSM_DATA_PTR CSSMAPI CSSM_CL_CertGetFirstFieldValue (CSSM_CL_HANDLE CLHandle,
const CSSM_DATA_PTR Cert,
const CSSM_OID_PTR CertField,
CSSM_HANDLE_PTR ResultsHandle,
uint32 *NumberOfMatchedFields)

This function returns the value of the designated certificate field. If more than one field matches the *CertField* OID, the first matching field will be returned. The number of matching fields is an output parameter, as is the ResultsHandle to be used to retrieve the remaining matching fields.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate.

CertField (input)
A pointer to an OID that identifies the field value to be extracted from the *Cert*.

ResultsHandle (output)
A pointer to the CSSM_HANDLE that should be used to obtain any additional matching fields.

NumberOfMatchedFields (output)
The number of fields that match the *CertField* OID.

Return Value

A pointer to the CSSM_DATA structure containing the value of the requested field. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertGetNextFieldValue, CSSM_CL_CertAbortQuery, CSSM_CL_CertGetAllFields

7.2.7 CSSM_CL_CertGetKeyInfo

CSSM_KEY_PTR CSSMAPI CSSM_CL_CertGetKeyInfo (CSSM_CL_HANDLE CLHandle,
const CSSM_DATA_PTR Cert)

This function returns the public key and integral information about the key from the specified certificate. The key structure returned is a compound object. It can be used in any function requiring a key, such as creating a cryptographic context.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate from which to extract the public key information.

Return Value

A pointer to the CSSM_KEY structure containing the public key and possibly other key information. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertGetFirstFieldValue

7.2.8 CSSM_CL_CertGetNextFieldValue

CSSM_DATA_PTR CSSMAPI CSSM_CL_CertGetNextFieldValue (CSSM_CL_HANDLE CLHandle, CSSM_HANDLE ResultsHandle)

This function returns the value of a certificate field, when that field occurs multiple times in a certificate. Certificates with repeated fields (such as multiple signatures) have multiple field values corresponding to a single OID. A call to the function CSSM_CL_CertGetFirstFieldValue initiates the process and returns a results handle identifying the certificate from which values are being obtained and the OID corresponding to those values. The CSSM_CL_CertGetNextFieldValue function can be called repeatedly to obtain these values one at a time.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a certificate query.

Return Value

A pointer to the CSSM_DATA structure containing the value of the requested field. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertGetFirstFieldValue, CSSM_CL_CertAbortQuery

7.2.9 CSSM_CL_CertImport

CSSM_DATA_PTR CSSMAPI CSSM_CL_CertImport (CSSM_CL_HANDLE CLHandle,
CSSM_CERT_TYPE ForeignCertType,
const CSSM_DATA_PTR ForeignCert)

This function imports a certificate from the specified foreign format into the native format of the specified CL. The set of ForeignCertTypes supported for import varies with the CL module. See the documentation provided by the module vendor for a list of supported foreign certificate formats.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

ForeignCertType (input)
A unique value that identifies the type of the certificate being imported.

ForeignCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be imported into the CL modules native certificate type.

Return Value

A pointer to the CSSM_DATA structure containing the native-type certificate imported from the foreign certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertExport

7.2.10 CSSM_CL_CertSign

CSSM_DATA_PTR CSSMAPI CSSM_CL_CertSign (CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR CertToBeSigned,
const CSSM_DATA_PTR SignerCert,
const CSSM_FIELD_PTR SignScope,
uint32 ScopeSize)

This function creates a signed certificate by signing the fields of the input certificate as indicated by the *SignScope* array.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

CertToBeSigned (input)
The DER-encoded certificate to be signed.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be used to sign the subject certificate.

SignScope (input)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be signed. A null input signs all the fields in the certificate.

ScopeSize (input)
The number of entries in the sign scope list.

Return Value

A pointer to the CSSM_DATA structure containing the signed certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertVerify

7.2.11 CSSM_CL_CertVerify

CSSM_BOOL CSSMAPI CSSM_CL_CertVerify (CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR CertToBeVerified,
const CSSM_DATA_PTR SignerCert,
const CSSM_FIELD_PTR VerifyScope,
uint32 ScopeSize)

This function verifies that the signed certificate has not been altered since it was signed by the designated signer. Only one signature is verified by this function. If the certificate to be verified includes multiple signatures, this function must be applied once for each signature to be verified. This function verifies a digital signature over the certificate fields specified by VerifyScope. If the verification scope fields are not specified, the function performs verification using a preselected set of fields in the certificate.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

CertToBeVerified (input)
A pointer to the CSSM_DATA structure containing a certificate containing at least one signature for verification. An unsigned certificate template cannot be verified.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate used to sign the subject certificate.

VerifyScope (input/optional)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be used in verifying the signature (i.e., the fields that were used to calculate the signature). If the verify scope is null, the CL module assumes that its default set of certificate fields were used to calculate the signature and those same fields are used in the verification process.

ScopeSize (input)
The number of entries in the verify scope list. If the verification scope is not specified, the input value for scope size must be zero.

Return Value

CSSM_TRUE if the certificate signature verified. CSSM_FALSE if the certificate signature did not verify or an error condition occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CertSign

7.3 Certificate Revocation List Operations

7.3.1 CSSM_CL_CrlAbortQuery

CSSM_RETURN CSSMAPI CSSM_CL_CrlAbortQuery (CSSM_CL_HANDLE CLHandle,
CSSM_HANDLE ResultsHandle)

This function terminates the query initiated by CSSM_CL_CrlGetFirstFieldValue or CSSM_CL_CrlGetNextFieldValue and allows the CL to release all intermediate state information associated with the get operation.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a CRL query.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CrlGetFirstFieldValue, CSSM_CL_CrlGetNextFieldValue

7.3.2 CSSM_CL_CrlAddCert

CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlAddCert (CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR Cert,
const CSSM_DATA_PTR RevokerCert,
const CSSM_FIELD_PTR CrlEntryFields,
uint32 NumberOfFields,
const CSSM_DATA_PTR OldCrl)

This function revokes the input certificate by adding a record representing the certificate to the CRL. The values for the new entry in the CRL are specified by the list of OID/value input pairs. The reason for revocation is a typical value specified in the list. The revoker's certificate is used to sign the new CRL entry. The operation is valid only if the CRL has not been closed by the process of signing the CRL (i.e., execution of the function CSSM_CL_CrlSign). Once the CRL has been signed, entries can not be added or removed.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be revoked.

RevokerCert (input)
A pointer to the CSSM_DATA structure containing the revoker's certificate.

CrlEntryFields (input)
An array of OID/value pairs specifying the initial values for descriptive data fields of the new CRL entry.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlEntryFields input parameter.

OldCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to which the newly revoked certificate will be added.

Return Value

A pointer to the CSSM_DATA structure containing the updated CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CrlRemoveCert

7.3.3 CSSM_CL_CrlCreateTemplate

CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlCreateTemplate
(CSSM_CL_HANDLE CLHandle,
const CSSM_FIELD_PTR CrlTemplate,
uint32 NumberOfFields)

This function creates an unsigned, memory-resident CRL. Fields in the CRL are initialized with the descriptive data specified by the OID/value input pairs. The specified OID/value pairs can initialize all or a subset of the general attribute fields in the new CRL. Subsequent values may be set using the CSSM_CL_CrlSetFieldValues operation. The new CRL contains no revocation records.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

CrlTemplate (input)
An array of OID/value pairs specifying the initial values for descriptive data fields of the new CRL.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlTemplate input parameter.

Return Value

A pointer to the CSSM_DATA structure containing the new CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

7.3.4 CSSM_CL_CrlDescribeFormat

CSSM_OID_PTR CSSMAPI CSSM_CL_CrlDescribeFormat (CSSM_CL_HANDLE CLHandle,
uint32 *NumberOfFields)

This function returns a list of the object identifiers used to describe the CRL format supported by the specified CL.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

NumberOfFields (output)
The length of the returned array of OIDs.

Return Value

A pointer to the array of CSSM_OIDs which represent the supported CRL format. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

7.3.5 CSSM_CL_CrlGetFirstFieldValue

CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlGetFirstFieldValue (CSSM_CL_HANDLE CLHandle,
const CSSM_DATA_PTR Crl,
const CSSM_OID_PTR CrlField,
CSSM_HANDLE_PTR ResultsHandle,
uint32 *NumberOfMatchedFields)

This function returns the value of the designated CRL field. If more than one field matches the *CrlField* OID, the first matching field will be returned. The number of matching fields, *NumberOfMatchedFields*, is an output parameter, as is the *ResultsHandle* to be used to retrieve the remaining matching fields.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

Crl (input)
A pointer to the CSSM_DATA structure which contains the CRL from which the first revocation record is to be retrieved.

CrlField (input)
An OID that identifies the field value to be extracted from the *Crl*.

ResultsHandle (output)
A pointer to the CSSM_HANDLE that should be used to obtain any additional matching fields.

NumberOfMatchedFields (output)
The number of fields that match the *CrlField* OID.

Return Value

Returns a pointer to a CSSM_DATA structure containing the first field that matched the *CrlField*. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CrlGetNextFieldValue, CSSM_CL_CrlAbortQuery

7.3.6 CSSM_CL_CrlGetNextFieldValue

CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlGetNextFieldValue (CSSM_CL_HANDLE CLHandle, CSSM_HANDLE ResultsHandle)

This function returns the value of a CRL field, when that field occurs multiple times in a CRL. CRL with repeated fields have multiple field values corresponding to a single OID. A call to the function CSSM_CL_CrlGetFirstFieldValue initiates the process and returns a results handle identifying the CRL from which values are being obtained and the OID corresponding to those values. The CSSM_CL_CrlGetNextFieldValue function can be called repeatedly to obtain these values one at a time.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a CRL query.

Return Value

Returns a pointer to a CSSM_DATA structure containing the next field in the CRL that matched the *CrlField* specified in the CL_CrlGetFirstFieldValue function. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CrlGetFirstFieldValue, CSSM_CL_CrlAbortQuery

7.3.7 CSSM_CL_CrlRemoveCert

CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlRemoveCert (CSSM_CL_HANDLE CLHandle,
const CSSM_DATA_PTR Cert,
const CSSM_DATA_PTR OldCrl)

This function reinstates a certificate by removing it from the specified CRL. The operation is valid only if the CRL has not been closed by the process of signing the CRL using the function CSSM_CL_CrlSign. Once the CRL has been signed, entries can not be added or removed.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be reinstated.

OldCrl (input)
A pointer to the CSSM_DATA structure containing the CRL from which the certificate is to be removed.

Return Value

A pointer to the CSSM_DATA structure containing the updated CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CrlAddCert

7.3.8 CSSM_CL_CrlSetFields

CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlSetFields (CSSM_CL_HANDLE CLHandle,
const CSSM_FIELD_PTR CrlTemplate,
uint32 NumberOfFields,
const CSSM_DATA_PTR OldCrl)

This function will set the fields of the input CRL to the new values, specified by the input OID/value pairs. If there is more than one possible instance of an OID (e.g., as in an extension or CRL record), then a NEW field with the specified value is added to the CRL.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

CrlTemplate (input)
Any array of field OID/value pairs containing the values to initialize the CRL attribute fields.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlTemplate input parameter.

OldCrl (input)
The CRL to be updated with the new attribute values. The CRL must be unsigned and available for update.

Return Value

A pointer to the modified, unsigned CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

7.3.9 CSSM_CL_CrlSign

CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlSign (CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR UnsignedCrl,
const CSSM_DATA_PTR SignerCert,
const CSSM_FIELD_PTR SignScope,
uint32 ScopeSize)

This function signs the fields of the CRL indicated in the *SignScope* parameter, in accordance with the specified cryptographic context. Once the CRL has been signed it may not be modified. This means that entries cannot be added or removed from the CRL through application of the CSSM_CL_CrlAddCert or CSSM_CL_CrlRemoveCert operations. A signed CRL can be verified, applied to a data store, and searched for values.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

UnsignedCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to be signed.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be used to sign the CRL.

SignScope (*input/optional*)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be signed. If the signing scope is NULL, the CL module includes a default set of CRL fields in the signing process.

ScopeSize (*input*)
The number of entries in the sign scope list. If the signing scope is not specified, the input scope size must be zero.

Return Value

A pointer to the CSSM_DATA structure containing the signed CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CrlVerify

7.3.10 CSSM_CL_CrlVerify

CSSM_BOOL CSSMAPI CSSM_CL_CrlVerify (CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
const CSSM_DATA_PTR CrlToBeVerified,
const CSSM_DATA_PTR SignerCert,
const CSSM_FIELD_PTR VerifyScope,
uint32 ScopeSize)

This function verifies that the signed CRL has not been altered since it was signed by the designated signer. It does this by verifying the digital signature over the fields specified by the VerifyScope parameter.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

CrlToBeVerified (input)
A pointer to the CSSM_DATA structure containing the CRL to be verified.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate used to sign the CRL.

VerifyScope (*input/optional*)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be verified. If the verification scope is NULL, the CL module assumes that a default set of fields were used in the signing process, and those same fields are used in the verification process.

ScopeSize (*input*)
The number of entries in the verify scope list. If the verification scope is not specified, the input value for scope size must be zero.

Return Value

A CSSM_TRUE return value signifies that the CRL verifies successfully. When CSSM_FALSE is returned, either the CRL verified unsuccessfully or an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_CL_CrlSign

7.3.11 CSSM_CL_IsCertInCrl

CSSM_BOOL CSSMAPI CSSM_CL_IsCertInCrl (CSSM_CL_HANDLE CLHandle,
const CSSM_DATA_PTR Cert,
const CSSM_DATA_PTR Crl)

This function searches the CRL for a record corresponding to the certificate. The CRL itself may be signed or unsigned. Each entry within the CRL is signed by the revoker's certificate, hence an unsigned list can be validly searched for individually signed CRL entries.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be located.

Crl (input)
A pointer to the CSSM_DATA structure containing the CRL to be searched.

Return Value

A CSSM_TRUE return value signifies that the certificate is in the CRL. When CSSM_FALSE is returned, either the certificate is not in the CRL or an error has occurred. Use CSSM_GetError to obtain the error code.

7.4 Extensibility Functions

7.4.1 CSSM_CL_PassThrough

void * CSSMAPI CSSM_CL_PassThrough (CSSM_CL_HANDLE CLHandle,
CSSM_CC_HANDLE CCHandle,
uint32 PassThroughId,
const void*InputParams)

This function allows applications to call CL module-specific operations. Such operations may include queries or services that are specific to the domain represented by the CL module.

Parameters

CLHandle (input)
The handle that describes the CL module used to perform this function.

CCHandle (input/optional)
The handle that describes the context of the cryptographic operation. If the module-specific operation does not perform any cryptographic operations a cryptographic context is not required.

PassThroughId (input)
An identifier assigned by the CL module to indicate the exported function to perform.

InputParams (input)
A pointer to a module, implementation-specific structures containing parameters to be interpreted in a function-specific manner by the requested CL module. This parameter can be used as a pointer to an array of CSSM_DATA structures.

Return Value

A pointer to a module, implementation-specific structure containing the output from the passthrough function. The output data must be interpreted by the calling application based on externally available information. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

Chapter 8. Data Storage Library Services API

The primary purpose of a Data Storage Library (DL) module is to provide persistent storage of security-related objects including certificates, Certificate Revocation Lists (CRLs), keys, and policy objects. A DL module is responsible for the creation and accessibility of one or more data stores. A single DL module can be tightly tied to a Certificate Library (CL) and/or Trust Policy (TP) module, or can be independent of all other module types. A single data store can contain a single object type in one format, a single object type in multiple formats, or multiple object types. The persistent repository can be local or remote; for example, a DL can provide client access to a remote directory/storage service.

KeyWorks stores and manages meta-information about a DL in the KeyWorks registry. This information describes the storage and retrieval capabilities of a DL. Applications can query the KeyWorks registry to obtain information about the available DLs and attach to a DL that provides the needed services. Some DL services can acquire and store meta-information about each of the data stores it manages. When this information is available it is stored in the IBM KeyWorks registry. Not all DL service providers can supply this information.

The DL APIs define a data storage model that can be implemented using a custom storage device, a traditional local or remote file system service, a Database Management System (DBMS) package, or a complete (local or remote) information management system. The abstract data model defined by the DL APIs partitions all values stored in a data record into two categories: one or more mutable attributes and one opaque data object. The attribute values can be directly manipulated by the application and the DL module. Values stored within the opaque data object must be accessed using parsing functions. For example, a DL that stores certificates cannot interpret the format of those certificates. A set of parsing functions such as those defined in a CL module can be used to parse the opaque certificate object. The DL module defines a default set of parsing functions. An application can define a KeyWorks module to be used for parsing or can define its own set of parsing functions to be used during a data storage session.

8.1 Data Storage Data Structures

8.1.1 CSSM_DB_ACCESS_TYPE

This structure indicates a user's desired level of access to a data store.

```
typedef struct cssm_db_access_type {
    CSSM_BOOL ReadAccess;
    CSSM_BOOL WriteAccess;
    CSSM_BOOL PrivilegedMode; /* versus user mode */
    CSSM_BOOL Asynchronous; /* versus synchronous */
} CSSM_DB_ACCESS_TYPE, *CSSM_DB_ACCESS_TYPE_PTR;
```

Definitions:

ReadAccess - A Boolean indicating that the user requests read access.

WriteAccess - A Boolean indicating that the user requests write access.

PrivilegedMode - A Boolean indicating that the user requests privileged operations.

Asynchronous - A Boolean indicating that the user requests asynchronous access.

8.1.2 CSSM_DB_ATTRIBUTE_DATA

This data structure holds an attribute value that can be stored in an attribute field of a persistent record. The structure contains a value for the data item and a reference to the meta-information (typing information and schema information) associated with the attribute.

```
typedef struct cssm_db_attribute_data {
   CSSM_DB_ATTRIBUTE_INFO Info;
   CSSM_DATA Value;
} CSSM_DB_ATTRIBUTE_DATA, *CSSM_DB_ATTRIBUTE_DATA_PTR;
```

Definitions:

Info - A reference to the meta-information/schema describing this attribute in relationship to the data store at large.

Value - The data-present value assigned to the attribute.

8.1.3 CSSM_DB_ATTRIBUTE_INFO

This data structure describes an attribute of a persistent record. The description is part of the schema information describing the structure of records in a data store. The description includes the format of the attribute name and the attribute name itself. The attribute name implies the underlying data type of a value that may be assigned to that attribute.

```
typedef struct cssm_db_attribute_info {
   CSSM_DB_ATTRIBUTE_NAME_FORMAT AttributeNameFormat;
   union {
      char * AttributeName; /* eg. "record label" */
      CSSM_OID AttributeID; /* eg. CSSMOID_RECORDLABEL */
      uint32 AttributeNumber;
   };
} CSSM_DB_ATTRIBUTE_INFO, *CSSM_DB_ATTRIBUTE_INFO_PTR;
```

Definitions:

AttributeNameFormat - Indicates which of the three formats was selected to represent the attribute name.

AttributeName - A character string representation of the attribute name.

AttributeID - A DER-encoded object identifier (OID) representation of the attribute name.

AttributeNumber - A numeric representation of the attribute name.

8.1.4 CSSM_DB_ATTRIBUTE_NAME_FORMAT

This enumerated list defines three formats used to represent an attribute name. The name can be represented by a character string in the native string encoding of the platform, by a number, or the name can be represented by an opaque OID structure that is interpreted by the DL module.

```
typedef enum cssm_db_attribute_name_format {
   CSSM_DB_ATTRIBUTE_NAME_AS_STRING = 0,
   CSSM_DB_ATTRIBUTE_NAME_AS_OID = 1,
```

```
   CSSM_DB_ATTRIBUTE_NAME_AS_NUMBER = 2
} CSSM_DB_ATTRIBUTE_NAME_FORMAT, *CSSM_DB_ATTRIBUTE_NAME_FORMAT_PTR;
```

8.1.5 CSSM_DB_CERTRECORD_SEMANTICS

These bit-masks define a list of usage semantics for how certificates may be used. It is anticipated that additional sets of bit-masks will be defined listing the usage semantics of how other record types can be used, such as CRL record semantics, key record semantics, policy record semantics, etc.

```
CSSM_DB_CERT_USE_ROOT0x00000001          /* a self-signed root cert */
CSSM_DB_CERT_USE_TRUSTED 0x00000002      /* re-issued locally */
CSSM_DB_CERT_USE_SYSTEM0x00000004            /* contains IBM KeyWorks system cert */
CSSM_DB_CERT_USE_OWNER 0x00000008            /* private key, owned by the system's user */
CSSM_DB_CERT_USE_REVOKED 0x00000010  /* revoked cert, used w\CRL APIs */
CSSM_DB_CERT_SIGNING 0x00000011          /* use cert for signing only */
CSSM_DB_CERT_PRIVACY 0x00000012          /* use cert for encryption only */
```

8.1.6 CSSM_DB_CONJUNCTIVE

These are the conjunctive operations that can be used when specifying a selection criterion.

```
typedef enum cssm_db_conjunctive{
   CSSM_DB_NONE = 0,
   CSSM_DB_AND = 1,
   CSSM_DB_OR = 2
} CSSM_DB_CONJUNCTIVE, *CSSM_DB_CONJUNCTIVE_PTR;
```

8.1.7 CSSM_DB_HANDLE

A unique identifier for an open data store.

```
typedef uint32 CSSM_DB_HANDLE/* Data Store Handle */
```

8.1.8 CSSM_DB_INDEXED_DATA_LOCATION

This enumerated list defines where within a record the indexed data values reside. Indexes can be constructed on attributes or on fields within the opaque object in the record. CSSM_DB_INDEX_ON_UNKNOWN indicates that the logical location of the index value between these two categories is unknown.

```
typedef enum cssm_db_indexed_data_location {
   CSSM_DB_INDEX_ON_UNKNOWN = 0,
   CSSM_DB_INDEX_ON_ATTRIBUTE = 1,
   CSSM_DB_INDEX_ON_RECORD = 2
} CSSM_DB_INDEXED_DATA_LOCATION;
```

8.1.9 CSSM_DB_INDEX_INFO

This structure contains the meta-information or schema description of an index defined on an attribute. The description includes the type of index (e.g., unique key or nonunique key), the logical location of the

indexed attribute in the KeyWorks record (e.g., an attribute, a field within the opaque object in the record, or unknown), and the meta-information on the attribute itself.

```
typedef struct cssm_db_index_info {
   CSSM_DB_INDEX_TYPE IndexType;
   CSSM_DB_INDEXED_DATA_LOCATION IndexedDataLocation;
   CSSM_DB_ATTRIBUTE_INFO Info;
} CSSM_DB_INDEX_INFO, *CSSM_DB_INDEX_INFO_PTR;
```

Definitions:

IndexType - A CSSM_DB_INDEX_TYPE.

IndexedDataLocation - A CSSM_DB_INDEXED_DATA_LOCATION.

Info - The meta-information description of the attribute being indexed.

8.1.10 CSSM_DB_INDEX_TYPE

This enumerated list defines two types of indexes: indexes with unique values (i.e., primary database keys) and indexes with nonunique values. These values are used when creating a new data store and defining the schema for that data store.

```
typedef enum cssm_db_index_type {
   CSSM_DB_INDEX_UNIQUE = 0,
   CSSM_DB_INDEX_NONUNIQUE = 1
} CSSM_DB_INDEX_TYPE;
```

8.1.11 CSSM_DBINFO

This structure contains the meta-information about an entire data store. The description includes the types of records stored in the data store, the attribute schema for each record type, the index schema for all indexes over records in the data store, the type of authentication mechanism used to gain access to the data store, and other miscellaneous information used by the DL module to manage the data store in a secure manner.

```
typedef struct cssm_dbInfo {
   uint32 NumberOfRecordTypes;
   CSSM_DB_PARSING_MODULE_INFO_PTR DefaultParsingModules;
   CSSM_DB_RECORD_ATTRIBUTE_INFO_PTR RecordAttributeNames;
   CSSM_DB_RECORD_INDEX_INFO_PTR RecordIndexes;

   /* access restrictions for opening this data store */
   CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;

   /* transparent integrity checking options for this data store */
   CSSM_BOOL RecordSigningImplemented;
   CSSM_DATA SigningCertificate;
   CSSM_GUID SigningCsp;

   /* additional information */
   CSSM_BOOL IsLocal;
```

```
   char *AccessPath; /* URL, dir path, etc */
   void *Reserved;
} CSSM_DBINFO, *CSSM_DBINFO_PTR;
```

Definitions:

NumberOfRecordTypes - The number of distinct record types stored in this data store.

DefaultParsingModules - A pointer to a list of (record-type, Globally Unique ID (GUID)) pairs which define the default parsing module for each record type.

RecordAttributeNames - The meta-information (schema) about the attributes associated with each record type that can be stored in this data store.

RecordIndexes - The meta-information (schema) about the indexes that are defined over each of the record types that can be stored in this data store.

AuthenticationMechanism - Defines the authentication mechanism required when accessing this data store.

RecordSigningImplemented - A flag indicating whether or not the DL module provides record integrity service based on digital signaturing of the data store records.

SigningCertificate - The certificate used to sign data store records when the transparent record integrity option is in effect.

SigningCsp - The GUID for the Cryptographic Service Provider (CSP) to be used to sign data store records when the transparent record integrity option is in effect.

IsLocal - Indicates whether the physical data store is local.

AccessPath - A character string describing the access path to the data store, such as a uniform resource locator (URL), a file system pathname, a remote directory service name, etc.

Reserved - Reserved for future use.

8.1.12 CSSM_DB_OPERATOR

These are the logical operators that can be used when specifying a selection predicate.

```
typedef enum cssm_db_operator {
   CSSM_DB_EQUAL = 0,
   CSSM_DB_NOT_EQUAL = 1,
   CSSM_DB_APPROX_EQUAL = 2,
   CSSM_DB_LESS_THAN = 3,
   CSSM_DB_GREATER_THAN = 4,
   CSSM_DB_EQUALS_INITIAL_SUBSTRING = 5,
   CSSM_DB_EQUALS_ANY_SUBSTRING = 6,
   CSSM_DB_EQUALS_FINAL_SUBSTRING = 7,
   CSSM_DB_EXISTS = 8
} CSSM_DB_OPERATOR, *CSSM_DB_OPERATOR_PTR;
```

8.1.13 CSSM_DB_PARSING_MODULE_INFO

This structure aggregates the GUID of a default parsing module with the record type that it parses. A parsing module can parse multiple record types. The same GUID would be repeated with each record type parsed by the module.

```
typedef struct cssm_db_parsing_module_info {
   CSSM_DB_RECORDTYPERecordType;
   CSSM_GUIDModule;
} CSSM_DB_PARSING_MODULE_INFO, *CSSM_DB_PARSING_MODULE_INFO_PTR;
```

Definitions:

RecordType - The type of record parsed by the module specified by GUID.

Module - A GUID identifying the default parsing module for the specified record type.

8.1.14 CSSM_DB_RECORD_ATTRIBUTE_DATA

This structure aggregates the actual data values for all of the attributes in a single record.

```
typedef struct cssm_db_record_attribute_data {
   CSSM_DB_RECORDTYPE DataRecordType;
   uint32 SemanticInformation;
   uint32 NumberOfAttributes;
   CSSM_DB_ATTRIBUTE_DATA_PTR AttributeData;
} CSSM_DB_RECORD_ATTRIBUTE_DATA, *CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR;
```

Definitions:

DataRecordType - A CSSM_DB_RECORDTYPE.

SemanticInformation - A bit-mask of type CSSM_XXXRECORD_SEMANTICS defining how the record can be used. Currently, these bit-masks are defined only for certificate records (CSSM_CERTRECORD_SEMANTICS). For all other record types, a bit-mask of zero must be used or a set of semantically meaningful masks must be defined.

NumberOfAttributes - The number of attributes in the record of the specified type.

AttributeData - A list of attribute name/value pairs.

8.1.15 CSSM_DB_RECORD_ATTRIBUTE_INFO

This structure contains the meta-information or schema information about all of the attributes in a particular record type. The description specifies the record type, the number of attributes in the record type, and a type information for each attribute.

```
typedef struct cssm_db_record_attribute_info {
   CSSM_DB_RECORDTYPE DataRecordType;
   uint32 NumberOfAttributes;
   CSSM_DB_ATTRIBUTE_INFO_PTR AttributeInfo;
} CSSM_DB_RECORD_ATTRIBUTE_INFO, *CSSM_DB_RECORD_ATTRIBUTE_INFO_PTR;
```

Definitions:

DataRecordType - A CSSM_DB_RECORDTYPE.

NumberOfAttributes - The number of attributes in a record of the specified type.

AttributeInfo - A list of pointers to the type (schema) information for each of the attributes.

8.1.16 CSSM_DB_RECORD_INDEX_INFO

This structure contains the meta-information or schema description of the set of indexes defined on a single record type. The description includes the type of the record, the number of indexes, and the meta-information describing each index.

```
typedef struct cssm_db_record_index_info {
   CSSM_DB_RECORDTYPE DataRecordType;
   uint32 NumberOfIndexes;
   CSSM_DB_INDEX_INFO_PTR IndexInfo;
} CSSM_DB_RECORD_INDEX_INFO, *CSSM_DB_RECORD_INDEX_INFO_PTR;
```

Definitions:

DataRecordType - A CSSM_DB_RECORDTYPE.

NumberOfIndexes - The number of indexes defined on the records of the given type.

IndexInfo - An array of pointer to the meta-description of each index defined over the specified record type.

8.1.17 CSSM_DB_RECORD_PARSING_FNTABLE

This structure defines the three prototypes for functions that can parse the opaque data object stored in a record. It is used in the CSSM_DbSetRecordParsingFunctions function to override the default parsing module for a given record type. The DL module developer designates the default parsing module for each record type stored in the data store.

```
typedef struct cssm_db_record_parsing_fntable {
   CSSM_DATA_PTR (CSSMAPI *RecordGetFirstFieldValue)
      (CSSM_HANDLE Handle,
      CSSM_DB_RECORDTYPE RecordType,
      const CSSM_DATA_PTR Data,
      const CSSM_OID_PTR DataField,
      CSSM_HANDLE_PTR ResultsHandle,
      uint32 *NumberOfMatchedFields);
   CSSM_DATA_PTR (CSSMAPI *RecordGetNextFieldValue)
      (CSSM_HANDLE Handle,
      CSSM_HANDLE ResultsHandle);
   CSSM_RETURN (CSSMAPI *RecordAbortQuery)
      (CSSM_HANDLE Handle,
      CSSM_HANDLE ResultsHandle);
} CSSM_DB_RECORD_PARSING_FNTABLE, *CSSM_DB_RECORD_PARSING_FNTABLE_PTR;
```

Definitions:

**RecordGetFirstFieldValue* - A function to retrieve the value of a field in the opaque object. The field is specified by attribute name. The results handle holds the state information required to retrieve subsequent values having the same attribute name.

**RecordGetNextFieldValue* - A function to retrieve subsequent values having the same attribute name from a record parsed by the first function in this table.

**RecordAbortQuery* - Stop subsequent retrieval of values having the same attribute name from within the opaque object.

8.1.18 CSSM_DB_RECORDTYPE

This enumerated list defines the categories of persistent security-related objects that can be managed by a DL module. These categories are in one-to-one correspondence with types of records that can be managed by a DL module.

```
typedef enum cssm_db_recordtype {
   CSSM_DL_DB_RECORD_GENERIC = 0,
   CSSM_DL_DB_RECORD_CERT = 1,
   CSSM_DL_DB_RECORD_CRL = 2,
   CSSM_DL_DB_RECORD_PUBLIC_KEY = 3,
   CSSM_DL_DB_RECORD_PRIVATE_KEY = 4,
   CSSM_DL_DB_RECORD_SYMMETRIC_KEY = 5,
   CSSM_DL_DB_RECORD_POLICY = 6
} CSSM_DB_RECORDTYPE;
```

8.1.19 CSSM_DB_UNIQUE_RECORD

This structure contains an index descriptor and a module-defined value. The index descriptor may be used by the module to enhance the performance when locating the record. The module-defined value must uniquely identify the record. For a DBMS, this may be the record data. For a Public-Key Cryptographic Standard (PKCS#11) DL, this may be an object handle. Alternately, the DL may have a module-specific scheme for identifying data that has been inserted or retrieved.

```
typedef struct cssm_db_unique_record {
   CSSM_DB_INDEX_INFO RecordLocator;
   CSSM_DATA RecordIdentifier;
} CSSM_DB_UNIQUE_RECORD, *CSSM_DB_UNIQUE_RECORD_PTR;
```

Definitions:

RecordLocator -The information describing how to locate the record efficiently.

RecordIdentifier - The actual data record values.

8.1.20 CSSM_DL_DB_HANDLE

This data structure holds a pair of handles, one for a DL, and another for a data store that is opened and being managed by the DL.

```
typedef struct cssm_dl_db_handle {
```

```
   CSSM_DL_HANDLE DLHandle;
   CSSM_DB_HANDLE DBHandle;
} CSSM_DL_DB_HANDLE, *CSSM_DL_DB_HANDLE_PTR;
```

Definitions:

DLHandle - Handle of an attached module that provides DL services.

DBHandle - Handle of an open data store that is currently under the management of the DL module specified by the DLHandle.

8.1.21 CSSM_DL_DB_LIST

This data structure defines a list of handle pairs of (DL handle, data store handle).

```
typedef struct cssm_dl_db_list {
   uint32 NumHandles;
   CSSM_DL_DB_HANDLE_PTR DLDBHandle;
} CSSM_DB_LIST, *CSSM_DB_LIST_PTR;
```

Definitions:

NumHandles - Number of (DL handle, data store handle) pairs in the list.

DLDBHandle - List of (DL handle, data store handle) pairs.

8.1.22 CSSM_DL_CUSTOM_ATTRIBUTES

This structure can be used by DL module developers to define a set of attributes for a custom data store format.

```
typedef void *CSSM_DL_CUSTOM_ATTRIBUTES;
```

8.1.23 CSSM_DL_FFS_ATTRIBUTES

This structure can be used by DL module developers to define a set of attributes for a flat file system data store format.

```
typedef void *CSSM_DL_FFS_ATTRIBUTES;
```

8.1.24 CSSM_DL_HANDLE

A unique identifier for an attached module that provides DL services.

```
typedef uint32 CSSM_DL_HANDLE/* Data Storage Library Handle */
```

8.1.25 CSSM_DL_LDAP_ATTRIBUTES

This structure can be used by DL module developers to define a set of attributes for a Lightweight Directory Access Protocol (LDAP) data store format.

```
typedef void *CSSM_DL_LDAP_ATTRIBUTES;
```

8.1.26 CSSM_DL_ODBC_ATTRIBUTES

This structure can be used by DL module developers to define a set of attributes for an Open Database Connectivity (ODBC) data store format.

```
typedef void *CSSM_DL_ODBC_ATTRIBUTES;
```

8.1.27 CSSM_DL_PKCS11_ATTRIBUTES

Each type of DL module can define its own set of type-specific attributes. This structure contains the attributes that are specific to a PKCS#11 compliant data storage device.

```
typedef struct cssm_dl_pkcs11_attributes {
   uint32 DeviceAccessFlags;
} *CSSM_DL_PKCS11_ATTRIBUTES;
```

Definitions:

DeviceAccessFlags - Specifies the PKCS#11 specific access modes applicable for accessing persistent objects in a PKCS#11 data store.

8.1.28 CSSM_DLSUBSERVICE

Three structures are used to contain all of the static information that describes a DL module: cssm_moduleinfo, cssm_serviceinfo, and cssm_dlsubservice. This descriptive information is securely stored in the KeyWorks registry when the DL module is installed with KeyWorks. A DL module may implement multiple types of services and organize them as subservices. For example, a DL module supporting two types of remote directory services may organize its implementation into two subservices: one for an X.509 certificate directory and a second for custom enterprise policy data store. Most DL modules will implement exactly one subservice.

Not all DL modules can maintain a summary of managed data stores. In this case, the DL module reports its number of data stores as CSSM_DB_DATASTORES_UNKNOWN. Data stores can (and probably do) exist, but the DL module cannot provide a list of them.

```
#define     CSSM_DB_DATASTORES_UNKNOWN        (-1)
```

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo and specifying the DL module GUID.

```
typedef struct cssm_dlsubservice {
   uint32 SubServiceId;
   CSSM_STRING Description;
   CSSM_DLTYPE Type;
   union {
      CSSM_DL_CUSTOM_ATTRIBUTES CustomAttributes;
      CSSM_DL_LDAP_ATTRIBUTES LdapAttributes;
      CSSM_DL_ODBC_ATTRIBUTES OdbcAttributes;
      CSSM_DL_PKCS11_ATTRIBUTES Pkcs11Attributes;
      CSSM_DL_FFS_ATTRIBUTES FfsAttributes;
   } Attributes;

   CSSM_DL_WRAPPEDPRODUCT_INFO WrappedProduct;
```

```
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;

    /* meta-information about the query support provided by the module */
    uint32 NumberOfRelOperatorTypes;
    CSSM_DB_OPERATOR_PTR RelOperatorTypes;
    uint32 NumberOfConjOperatorTypes;
    CSSM_DB_CONJUNCTIVE_PTR ConjOperatorTypes;
    CSSM_BOOL QueryLimitsSupported;

    /* meta-information about the encapsulated data stores (if known) */
    uint32 NumberOfDataStores;
    CSSM_NAME_LIST_PTR DataStoreNames;
    CSSM_DBINFO_PTR DataStoreInfo;

    /* additional information */
    void *Reserved;
} CSSM_DLSUBSERVICE, *CSSM_DLSUBSERVICE_PTR;
```

Definitions:

SubServiceId - A unique, identifying number for the subservice described in this structure.

Description - A string containing a descriptive name or title for this subservice.

Type - An identifier for the type of underlying data store the DL module uses to provide persistent storage.

Attributes - A structure containing attributes that define additional parameter values specific to the DL module type.

WrappedProduct - Pointer to a CSSM_DL_WRAPPEDPRODUCT_INFO structure describing a product that is wrapped by the DL module.

AuthenticationMechanism - Defines the authentication mechanism required when using this DL module. This authentication mechanism is distinct from the authentication mechanism (specified in a cssm_dbInfo structure) required to access a specific data store.

NumberOfRelOperatorTypes - The number of distinct relational operators the DL module accepts in selection queries for retrieving records from its managed data stores.

RelOperatorTypes - The list of specific relational operators that can be used to formulate selection predicates for queries on a data store. The list contains NumberOfRelOperatorTypes operators.

NumberOfConjOperatorTypes - The number of distinct conjunctive operator the DL module accepts in selection queries for retrieving records from its managed data stores.

ConjOperatorTypes - A list of specific conjunctive operators that can be used to formulate selection predicates for queries on a data store. The list contains NumberOfConjOperatorTypes operators.

QueryLimitsSupported - A Boolean indicating whether query limits are effective when the DL module executes a query.

NumberOfDataStores - The number of data stores managed by the DL module. This information may not be known by the DL module, in which case this value will equal CSSM_DB_DATASTORES_UNKNOWN.

DataStoreNames - A list of names of the data stores managed by the DL module. This information may not be known by the DL module and hence may not be available. The list contains NumberOfDataStores entries.

DataStoreInfo - A list of pointers to the meta-information (schema) for each data store managed by the DL module. This information may not be known in advance by the DL module and hence may not be available through this structure. The list contains NumberOfDataStores entries.

Reserved - Reserved for future use.

8.1.29 CSSM_DLTYPE

This enumerated list defines the types of underlying DBMSs that can be used by the DL module to provide services. It is the option of the DL module to disclose this information.

```
typedef enum cssm_dltype {
   CSSM_DL_UNKNOWN = 0,
   CSSM_DL_CUSTOM = 1,
   CSSM_DL_LDAP = 2,
   CSSM_DL_ODBC = 3,
   CSSM_DL_PKCS11 = 4,
   CSSM_DL_FFS = 5,/* flat file systemor fast file system */
   CSSM_DL_MEMORY = 6,
   CSSM_DL_REMOTEDIR = 7
} CSSM_DLTYPE, *CSSM_DLTYPE_PTR;
```

8.1.30 CSSM_DL_WRAPPEDPRODUCTINFO

This structure lists the set of data store services used by the DL module to implement its services. The DL module vendor is not required to provide this information, but may choose to do so. For example, a DL module that uses a commercial DBMS can record information about that product in this structure. Another example is a DL module that supports certificate storage through an X.500 certificate directory server. The DL module can describe the X.500 directory service in this structure.

```
typedef struct cssm_dl_wrappedproductinfo {
   CSSM_VERSION StandardVersion;
   CSSM_STRING StandardDescription;
   CSSM_VERSION ProductVersion;
   CSSM_STRING ProductDescription;
   CSSM_STRING ProductVendor;
   uint32 ProductFlags;
} CSSM_DL_WRAPPEDPRODUCT_INFO, *CSSM_DL_WRAPPEDPRODUCT_INFO_PTR;
```

Definitions:

StandardVersion - If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription - If this product conforms to an industry standard, this is a description of that standard.

ProductVersion - Version number information for the actual product version used in this version of the DL module.

ProductDescription - A string describing the product.

ProductVendor - The name of the product vendor.

ProductFlags - A bit-mask enumerating selectable features of the database service that the DL module uses in its implementation.

8.1.31 CSSM_NAME_LIST

```
typedef struct cssm_name_list {
   uint32 NumStrings;
   char** String;
} CSSM_NAME_LIST, *CSSM_NAME_LIST_PTR;
```

8.1.32 CSSM_QUERY

This structure holds a complete specification of a query to select records from a data store.

```
typedef struct cssm_query {
   CSSM_DB_RECORDTYPE RecordType;
   CSSM_DB_CONJUNCTIVE Conjunctive;
   uint32 NumSelectionPredicates;
   CSSM_SELECTION_PREDICATE_PTR SelectionPredicate;
   CSSM_QUERY_LIMITS QueryLimits;
   CSSM_QUERY_FLAGS QueryFlags;
} CSSM_QUERY, *CSSM_QUERY_PTR;
```

Definitions:

RecordType - Specifies the type of record to be retrieved from the data store.

Conjunctive - The conjunctive operator to be used in constructing the selection predicate for the query.

NumSelectionPredicates - The number of selection predicates to be connected by the specified conjunctive operator to form the query.

SelectionPredicate - The list of selection predicates to be combined by the conjunctive operator to form the data store query.

QueryLimits - Defines the time and space limits for processing the selection query. The constant values CSSM_QUERY_TIMELIMIT_NONE and CSM_QUERY_SIZELIMIT_NONE should be used to specify no limit on the resources used in processing the query.

QueryFlags - An integer that indicates the return format of the key data. This integer is represented by CSSM_QUERY_RETURN_DATA. When CSSM_QUERY_RETURN_DATA is 1, the key record is returned in Common Data Security Architecture (CDSA) format. When CSSM_QUERY_RETURN_DATA is 0, the information is returned in raw format (a format native to the individual CSP, such as BSAFE or PKCS#11).

8.1.33 CSSM_QUERY_LIMITS

This structure defines the time and space limits a caller can set to control early termination of the execution of a data store query. The constant values CSSM_QUERY_TIMELIMIT_NONE and CSM_QUERY_SIZELIMIT_NONE should be used to specify no limit on the resources used in processing the query. These limits are advisory. Not all DL modules recognize and act upon the query limits set by a caller.

```
#define    CSSM_QUERY_TIMELIMIT_NONE          0
#define    CSSM_QUERY_SIZELIMIT_NONE          0

typedef struct cssm_query_limits {
   uint32 TimeLimit;
   uint32 SizeLimit;
} CSSM_QUERY_LIMITS, *CSSM_QUERY_LIMITS_PTR;
```

Definitions:

TimeLimit - Defines the maximum number of seconds of resource time that should be expended performing a query operation. The constant value CSSM_QUERY_TIMELIMIT_NONE means no time limit is specified.

SizeLimit - Defines the maximum number of records that should be retrieved in response to a single query. The constant value CSSM_QUERY_SIZELIMIT_NONE means no space limit is specified.

8.1.34 CSSM_SELECTION_PREDICATE

This structure defines the selection predicate to be used for database queries.

```
typedef struct cssm_selection_predicate {
   CSSM_DB_OPERATOR DbOperator;
   CSSM_DB_ATTRIBUTE_DATA Attribute;
} CSSM_SELECTION_PREDICATE, *CSSM_SELECTION_PREDICATE_PTR;
```

Definitions:

DbOperator - The relational operator to be used when comparing a value to the values stored in the specified attribute in the data store.

Attribute - The meta-information about the attribute to be searched and the attribute value to be used for comparison with values in the data store.

8.2 Data Storage Functions

8.2.1 CSSM_DL_Authenticate

CSSM_RETURN CSSMAPI CSSM_DL_Authenticate
(CSSM_DL_DB_HANDLE DLDBHandle,
const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
const CSSM_USER_AUTHENTICATION_PTR
UserAuthentication)

This function allows the caller to provide authentication credentials to the DL module at a time other than data store creation, deletion, open, import, and export. *AccessRequest* defines the type of access to be associated with the caller. If the authentication credentials apply to access and use of a DL module in general, then the data store handle specified in the *DLDBHandle* must be NULL. When the authorization credentials are applied to a specific data store, the handle for that data store must be specified in the *DLDBHandle* pair.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module used to perform this function and the data store to which access is being requested. If the form of authentication being requested is authentication to the DL module in general, then the data store handle must be NULL.

AccessRequest (input)
An indicator of the requested access mode for the data store or DL module in general.

UserAuthentication (input)
The caller's credential as required for obtaining authorized access to the data store or to the DL module in general.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

8.2.2 CSSM_DL_DbClose

CSSM_RETURN CSSMAPI CSSM_DL_DbClose (CSSM_DL_DB_HANDLE DLDBHandle)

This function closes an open data store.

Parameters

DLDBHandle (input)
A handle structure containing the DL handle for the attached DL module and the DB handle for an open data store managed by the DL. This specifies the open data store to be closed.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DbOpen

8.2.3 CSSM_DL_DbCreate

CSSM_DB_HANDLE CSSMAPI CSSM_DL_DbCreate
(CSSM_DL_HANDLE DLHandle,
const char *DbName,
const CSSM_DBINFO_PTR DBInfo,
const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
const CSSM_USER_AUTHENTICATION_PTR
UserAuthentication,
const void *OpenParameters)

This function creates and opens a new data store. The name of the new data store is specified by the input parameter *DbName*. The record schema for the data store is specified in the *DBInfo* structure. The newly created data store is opened under the specified access mode. If user authentication credentials are required, they must be provided. Also, additional open parameters may be required and are supplied in the *OpenParameters*.

Parameters

DLHandle (input)
The handle that describes the DL module used to perform this function.

DbName (input)
The general, external name for the new data store.

DBInfo (input)
A pointer to a structure describing the format/schema of each record type that will be stored in the new data store.

AccessRequest (input)
An indicator of the requested access mode for the data store, such as read-only or read/write.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

Return Value

The handle the newly created and open data store. When NULL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DbOpen, CSSM_DL_DbClose, CSSM_DL_DbDelete

8.2.4 CSSM_DL_DbDelete

CSSM_RETURN CSSMAPI CSSM_DL_DbDelete
(CSSM_DL_HANDLE DLHandle,
const char *DbName,
const CSSM_USER_AUTHENTICATION_PTR
UserAuthentication)

This function deletes all records from the specified data store and removes all state information associated with that data store.

Parameters

DLHandle (input)
The handle that describes the DL module to be used to perform this function.

DbName (input)
A pointer to the string containing the logical name of the data store.

UserAuthentication (input/optional)
The caller's credentials as required for obtaining access (and consequently deletion capability) to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DbCreate, CSSM_DL_DbOpen, CSSM_DL_DbClose

8.2.5 CSSM_DL_DbExport

CSSM_RETURN CSSMAPI CSSM_DL_DbExport
(CSSM_DL_HANDLE DLHandle,
const char *DbDestinationName,
const char *DbSourceName,
CSSM_BOOL InfoOnly,
const CSSM_USER_AUTHENTICATION_PTR
UserAuthentication)

This function exports a copy of the data store records from the source data store to a data container that can be used as the input data source for the CSSM_DL_DbImport function. The DL module may require additional user authentication to determine whether the user is authorized to create a copy of an existing data store.

Parameters

DLHandle (input)
The handle that describes the DL module to be used to perform this function.

DbSourceName (input)
The name of the data store from which the records are to be exported.

DbDestinationName (*input)*
The name of the destination data container that will contain a copy of the source data store's records.

InfoOnly (input)
A Boolean value indicating what to export. If CSSM_TRUE, export only the DBInfo, which describes the data store. If CSSM_FALSE, export both the DBInfo and all of the records in the specified data store.

UserAuthentication (input/optional)
The caller's credentials as required for authorization to copy a data store. If the DL module requires no additional credentials to perform this operation, then user authentication can be NULL.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DbImport

8.2.6 CSSM_DL_DbGetRecordParsingFunctions

CSSM_DB_RECORD_PARSING_FNTABLE_PTR CSSMAPI
CSSM_DL_DbGetRecordParsingFunction
(CSSM_DL_HANDLE DLHandle,
const char* DbName,
CSSM_DB_RECORDTYPE RecordType)

This function gets the records parsing function table, which operates on records of the specified type from the specified data store. Three record parsing functions can be returned in the table. The functions can be implemented to parse multiple record types. However, in order to parse multiple record types, multiple calls to CSSM_DL_DbGetRecordParsingFunctions must be made, once for each record type whose parsing functions are required by the caller. The DL module uses these functions to parse the opaque data object stored in a data store record. If no parsing function table has been set for a given record type, then a NULL value is returned.

Parameters

DLHandle (input)
The handle that describes the DL module to be used to perform this function.

DbName (input)
The name of the data store with which the parsing functions is associated.

RecordType (input)
The record type whose parsing functions are requested by the caller.

Return Value

A pointer to a function table for the parsing function appropriate to the specified record type. When NULL is returned, either no function table has been set for the specified record type or an error has occurred. Use CSSM_GetError to obtain the error code and determine the reason for the NULL result.

See Also

CSSM_DL_SetRecordParsingFunctions

8.2.7 CSSM_DL_DbImport

CSSM_RETURN CSSMAPI CSSM_DL_DbImport
(CSSM_DL_HANDLE DLHandle,
const char *DbDestinationName,
const char *DbSourceName,
const CSSM_DBINFO_PTR DBInfo,
CSSM_BOOL InfoOnly,
const CSSM_USER_AUTHENTICATION_PTR
UserAuthentication)

This function creates a new data store, or adds to an existing data store, by importing records from the specified data source. It is assumed that the data source contains records exported from a data store using the function CSSM_DL_DbExport.

The *DbDestinationName* parameter specifies the name of a new or existing data store. If a new data store is being created, the *DBInfo* structure provides the meta-information (schema) for the new data store. This structure describes the record attributes and the index schema for the new data store. If the data store already exists, then the existing meta-information (schema) is used. (Dynamic schema evolution is not supported.)

Typically, user authentication is required to create a new data store or to write to an existing data store. An authentication credential is presented to the DL module in the form required by the module. (See the documentation provided with the DL module for information on the required form.) The resulting data store is not opened as a result of this operation.

Parameters

DLHandle (input)
The handle that describes the DL module to be used to perform this function.

DbDestinationName (input)
The name of the destination data store into which the records will be inserted.

DbSourceName (*input)*
The name of the data source from which to obtain the records that are added to the data store.

DBInfo (*input/optional)*
A data structure containing a detailed description of the meta-information (schema) for the new data store. If a new data store is being created, then the caller must specify the meta-information (schema), or the data source must include the meta-information required for proper import of the records. If meta-information is supplied by the caller and specified in the data source, then the meta-information provided by the caller overrides the meta-information recorded in the data source.

If the data store exists and records are being added, then this pointer must be NULL. The existing meta-information will be used and the schema cannot be evolved.

InfoOnly (input)
A Boolean value indicating what to import. If CSSM_TRUE, import only the DBInfo, which describes the data store. If CSSM_FALSE, import both the DBInfo and all of the records exported from a data store.

UserAuthentication (input/optional)
The caller's credential as required for authorization to create a data store. If the DL module requires no additional credentials to create a new data store, then user authentication can be NULL.

Return Value

A CSSM_OK return value signifies that the function completed successfully and the new data store was created. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DbExport

8.2.8 CSSM_DL_DbOpen

CSSM_DB_HANDLE CSSMAPI CSSM_DL_DbOpen
(CSSM_DL_HANDLE DLHandle,
const char *DbName,
const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
const CSSM_USER_AUTHENTICATION_PTR
UserAuthentication,
const void *OpenParameters)

This function opens the data store with the specified logical name under the specified access mode. If user authentication credentials are required, they must be provided. Also, additional open parameters may be required to open a given data store and are supplied in the OpenParameters.

Parameters

DLHandle (input)
The handle that describes the DL module to be used to perform this function.

DbName (input)
A pointer to the string containing the logical name of the data store.

AccessRequest (input)
An indicator of the requested access mode for the data store, such as read-only or read/write.

UserAuthentication (input/optional)
The caller's credentials as required for obtaining access to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

Return Value

The handle to the opened data store. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DbClose

8.2.9 CSSM_DL_DbSetRecordParsingFunctions

CSSM_RETURN CSSMAPI CSSM_DL_DbSetRecordParsingFunctions
(CSSM_DL_HANDLE DLHandle,
const char* DbName,
CSSM_DB_RECORDTYPE RecordType,
const CSSM_DB_RECORD_PARSING_FNTABLE_PTR
FunctionTable)

This function sets the record parsing function table, overriding the default parsing module, for records of the specified type in the specified data store. Three record parsing functions can be specified in the table. The functions can be implemented to parse multiple record types. In this case, multiple calls to CSSM_DL_DbSetRecordParsingFunctions must be made, once for each record type that should be parsed using these functions. The DL module uses these functions to parse the opaque data object stored in a data store record. If no parsing function table has been set for a given record type, then the default parsing module is invoked for that record type.

Parameters

DLHandle (input)
The handle that describes the DL module to be used to perform this function.

DbName (input)
The name of the data store with which to associate the parsing functions.

RecordType (input)
One of the record types parsed by the functions specified in the function table.

FunctionTable (input)
The function table referencing the three parsing functions to be used with the data store specified by DbName.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_GetRecordParsingFunctions

8.2.10 CSSM_DL_GetDbNameFromHandle

char * CSSMAPI CSSM_DL_GetDbNameFromHandle (CSSM_DL_DB_HANDLE DLDBHandle)

This function retrieves the data source name corresponding to an opened database handle. A DL module is responsible for allocating the memory required for the list.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module used to perform this function and the data store to which access is being requested. If the form of authentication being requested is authentication to the DL module in general, the data store handle must be NULL.

Return Value

Returns a string that contains a data store name. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

8.3 Data Record Operations

8.3.1 CSSM_DL_AbortQuery

CSSM_RETURN CSSMAPI CSSM_DL_DataAbortQuery (CSSM_DL_DB_HANDLE DLDBHandle,
CSSM_HANDLE ResultsHandle)

This function terminates the query initiated by CSSM_DL_DataGetFirst or CSSM_DL_DataGetNext, and allows a DL to release all intermediate state information associated with the query.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store from which records were selected by the initiating query.

ResultsHandle (input)
The selection handle returned from the initial query function.

Return Value

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DataGetFirst, CSSM_DL_DataGetNext

8.3.2 CSSM_DL_DataDelete

CSSM_RETURN CSSMAPI CSSM_DL_DataDelete
(CSSM_DL__DB_HANDLE DLDBHandle,
CSSM_DB_RECORDTYPE RecordType,
const CSSM_DB_UNIQUE_RECORD_PTR
UniqueRecordIdentifier)

This function removes from the specified data store, the data record specified by *UniqueRecordIdentifier*.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store from which to delete the specified data record.

RecordType (input/optional)
An indicator of the type of record to be deleted from the data store. The *UniqueRecordIdentifier* may be unique only among records of the same type. If the data store contains only one record type, or the unique identifiers managed are globally unique, then the record type need not be specified.

UniqueRecordIdentifier (input)
A pointer to a CSSM_DB_UNIQUE_RECORD identifier containing unique identification of the data record to be deleted from the data store. The identifier may be unique only among records of a given type. Once the associated record has been deleted, this unique record identifier can not be used in future references.

Return Value

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DataInsert

8.3.3 CSSM_DL_DataGetFirst

CSSM_DB_UNIQUE_RECORD_PTR CSSMAPI CSSM_DL_DataGetFirst
(CSSM_DL_DB_HANDLE DLDBHandle,
const CSSM_QUERY_PTR Query,
CSSM_HANDLE_PTRResultsHandle,
CSSM_BOOL*EndOfDataStore,
CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR
Attributes,
CSSM_DATA_PTRData)

This function retrieves the first data record in the data store that matches the selection criteria. The selection criteria (including selection predicate and comparison values) is specified in the *Query* structure. The DL module can use internally managed indexing structures to enhance the performance of the retrieval operation. This function returns the first record that satisfies the query in the list of *Attributes* and the opaque *Data* object. This function also returns a flag indicating whether additional records also satisfied the query and a results handle to be used when retrieving subsequent records. Finally, this function returns a unique record identifier associated with the retrieved record. This structure can be used in future references to the retrieved data record.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store to search for records satisfying the query.

Query (input/optional)
The query structure specifying the selection predicates used to query the data store. The structure contains meta-information about the search fields and the relational and conjunctive operators forming the selection predicate. The comparison values to be used in the search are specified in the *Attributes* and *Data* parameter. If no query is specified, the DL module can return the first record in the data store (i.e., perform sequential retrieval) or return an error.

ResultsHandle (output)
This handle should be used to retrieve subsequent records that satisfied this query.

EndOfDataStore (output)
A flag indicating whether a record satisfying this query was available to be retrieved in the current operation. If CSSM_TRUE, then a record was available and was retrieved unless an error condition occurred. If CSSM_FALSE, then all records satisfying the query have been previously retrieved and no record has been returned by this operation.

Attributes (output)
A list of attributes values (and corresponding meta-information) from the retrieved record.

Data (output)
The opaque object stored in the retrieved record.

Return Value

If successful and *EndOfDataStore* is CSSM_FALSE, this function returns a pointer to a CSSM_UNIQUE_RECORD structure containing a unique record locator and the record. If the pointer is NULL and *EndOfDataStore* is CSSM_TRUE, then a normal termination condition has occurred. If the pointer is NULL and *EndOfDataStore* is CSSM_FALSE, then an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DataGetNext, CSSM_DL_DataAbortQuery

8.3.4 CSSM_DL_DataGetNext

CSSM_DB_UNIQUE_RECORD_PTR CSSMAPI CSSM_DL_DataGetNext
(CSSM_DL_DB_HANDLE DLDBHandle,
CSSM_HANDLE ResultsHandle,
CSSM_BOOL *EndOfDataStore,
CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
CSSM_DATA_PTR Data)

This function returns the next data record referenced by the *ResultsHandle*. The *ResultsHandle* references a set of records selected by an invocation of the CSSM_DL_DataGetFirst function. The record values are returned in the *Attributes* and *Data* parameters. A flag indicates whether additional records satisfying the original query remain to be retrieved. The function also returns a unique record identifier for the return record.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store from which records were selected by the initiating query.

ResultsHandle (output)
The handle identifying a set of records retrieved by a query executed by the CSSM_DL_DataGetFirst function.

EndOfDataStore (output)
A flag indicating whether a record satisfying this query was available to be retrieved in the current operation. If CSSM_TRUE, then a record was available and was retrieved unless an error condition occurred. If CSSM_FALSE, then all records satisfying the query have been previously retrieved and no record has been returned by this operation.

Attributes (output)
A list of attributes values (and corresponding meta-information) from the retrieved record.

Data (output)
The opaque object stored in the retrieved record.

Return Value

If successful and *EndOfDataStore* is CSSM_FALSE, this function returns a pointer to a CSSM_UNIQUE_RECORD structure containing a unique record locator and the record. If the pointer is NULL and *EndOfDataStore* is CSSM_TRUE, then a normal termination condition has occurred. If the pointer is NULL and *EndOfDataStore* is CSSM_FALSE, then an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DataGetFirst, CSSM_DL_DataAbortQuery

8.3.5 CSSM_DL_DataInsert

CSSM_DB_UNIQUE_RECORD_PTR CSSMAPI CSSM_DL_DataInsert
(CSSM_DL_DB_HANDLE DLDBHandle,
CSSM_DB_RECORDTYPE RecordType,
const CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
const CSSM_DATA_PTR Data)

This function creates a new persistent data record of the specified type by inserting it into the specified data store. The values contained in the new data record are specified by the *Attributes* and the *Data* fields. The attribute value list contains zero or more attribute values. The DL modules can assume default values for unspecified attribute values or can return an error condition when required attributes values are not specified by the caller. The *Data* is an opaque object to be stored in the new data record.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store in which to insert the new data record.

RecordType (input)
Indicates the type of data record being added to the data store.

Attributes (input/optional)
A list of structures containing the attribute values to be stored in that attribute and the meta-information (schema) describing those attributes. The list contains at most one entry per attribute in the specified record type. The DL module can assume default values for those attributes that are not assigned values by the caller or may return an error. If the specified record type does not contain any attributes, this parameter must be NULL.

Data (input/optional)
A pointer to the CSSM_DATA structure that contains the opaque data object to be stored in the new data record. If the specified record type does not contain an opaque data object, this parameter must be NULL.

Return Value

A pointer to a CSSM_DB_UNIQUE_RECORD_POINTER containing a unique identifier associated with the new record. This unique identifier structure can be used in future references to this record. When NULL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DataDelete

8.3.6 CSSM_DL_FreeUniqueRecord

CSSM_RETURN CSSMAPI CSSM_DL_FreeUniqueRecord
(CSSM_DL_DB_HANDLE DLDBHandle,
CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord)

Frees the pointer to a CSSM_DB_UNIQUE_RECORD data structure. The record itself and the data it contains is unchanged. To delete the record, call CSSM_DL_DataDelete before invoking CSSM_DL_FreeUniqueRecord.

Parameters

DLDBHandle (input)
The handle pair that describes the DL module to be used to perform this function and the open data store in which to insert the new data record.

UniqueRecord (input)
Pointer to a unique record.

Return Value

A CSSM_OK return value signifies that the unique record pointer was freed. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

See Also

CSSM_DL_DataDelete, CSSM_DL_DataInsert, CSSM_DL_DataGetFirst, CSSM_DL_DataGetNext

8.4 Extensibility Functions

8.4.1 CSSM_DL_PassThrough

void * CSSMAPI CSSM_DL_PassThrough (CSSM_DL_DB_HANDLE DLDBHandle,
uint32 PassThroughId,
const void *InputParams)

This function allows applications to call data storage library module-specific operations that have been exported. Such operations may include queries or services that are specific to the domain represented by a DL module.

Parameters

DLDBHandle (input)
The handle pair that describes the data storage library module to be used to perform this function and the open data store upon which the function is to be performed.

PassThroughId (input)
An identifier assigned by a DL module to indicate the exported function to be performed.

InputParams (input)
A pointer to a module, implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested DL module. This parameter can be used as a pointer to an array of CSSM_DATA_PTRs.

Return Value

A pointer to a module, implementation-specific structure containing the output from the passthrough function. The output data must be interpreted by the calling application based on externally-available information. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

Chapter 9. IBM KeyWorks Error Handling

This section describes the error handling features in IBM KeyWorks that provide a consistent mechanism across all layers of KeyWorks for returning errors to the caller. All KeyWorks API functions return one of the following:

- CSSM_RETURN - An enumerated type consisting of CSSM_OK and CSSM_FAIL. If it is CSSM_FAIL, an error code indicating the reason for failure can be obtained by calling CSSM_GetError.
- CSSM_BOOL - KeyWorks functions returning this data type return either CSSM_TRUE or CSSM_FALSE. If the function returns CSSM_FALSE, an error code may be available (but not always) by calling CSSM_GetError.
- A pointer to a data structure, a handle, a file size, or whatever is logical for the function to return. An error code may be available (but not always) by calling CSSM_GetError.

The information returned from CSSM_GetError includes both the error number and a Globally Unique ID (GUID) that associates the error with the module that set it. The GUID of each module can be obtained by calling CSSM_ListModules. CSSM_CompareGuids can then be called to determine from which module an error came.

Each module must have a mechanism for reporting their errors to the calling application. In general, there are two types of errors a module can return:

- Errors defined by KeyWorks that are common to a particular type of service provider module
- Errors reserved for use by individual service provider modules

Since some errors are predefined by KeyWorks, those errors have a set of predefined numeric values that are reserved by KeyWorks and cannot be redefined by modules. For errors that are particular to a module, a different set of predefined values has been reserved for their use. Table 17 lists the range of error numbers defined for KeyWorks and service provider modules. Appendix A lists the errors defined by KeyWorks.

Table 17. IBM KeyWorks Framework and Module Error Numbers

Error Number Range	KeyWorks Component
1000 – 1999	CSP errors defined by IBM KeyWorks
2000 - 2999	CSP errors reserved for individual CSP modules
3000 – 3999	CL errors defined by IBM KeyWorks
4000 – 4999	CL errors reserved for individual CL modules
5000 – 5999	DL errors defined by IBM KeyWorks
6000 – 6999	DL errors reserved for individual DL modules
7000 – 7999	TP errors defined by IBM KeyWorks
8000 – 8999	TP errors reserved for individual TP modules
9000 – 9499	KR errors defined by KeyWorks
9500 – 9999	KR errors reserved for individual KR modules
10000 – 19999	KeyWorks errors

The calling application must determine how to handle the error returned by CSSM_GetError. Detailed descriptions of the error values will be available in the corresponding specification, the cssmerr.h header file, and the documentation for specific modules. If a routine does not know how to handle the error, it may choose to pass the error to its caller.

Error values returned by a function should not be overwritten, if at all possible. For example, if a CSP call returns an error indicating that it could not encrypt the data, the caller should not overwrite it with an error simply indicating that the CSP failed, as it destroys valuable error handling and debugging information.

9.1 Data Structures

9.1.1 CSSM_BOOL

This data type is used to indicate a true or false condition.

```
typedef uint32 CSSM_BOOL;

#define CSSM_TRUE 1
#define CSSM_FALSE 0
```

Definitions:

CSSM_TRUE - Indicates a true result or a true value.

CSSM_FALSE - Indicates a false result or a false value.

9.1.2 CSSM_ERROR

```
typedef struct cssm_error {
   uint32 error;
   CSSM_GUID guid;
```

} CSSM_ERROR, *CSSM_ERROR_PTR

9.1.3 CSSM_RETURN

```
typedef enum cssm_return {
        CSSM_OK = 0,
        CSSM_FAIL = -1
} CSSM_RETURN
```

9.2 Error Handling Functions

9.2.1 CSSM_ClearError

void CSSMAPI CSSM_ClearError (void)

This function sets the current error value to CSSM_OK. This can be called if the current error value has been handled and therefore is no longer a valid error.

Parameters

None

See Also

CSSM_SetError, CSSM_GetError

9.2.2 CSSM_CompareGuids

CSSM_BOOL CSSMAPI CSSM_CompareGuids (CSSM_GUID guid1,
CSSM_GUID guid2)

This function determines if two GUIDs are equal.

Parameters

guid1 (input)
A GUID.

guid1 (input)
A GUID.

Return Value

CSSM_TRUE if the two GUIDs are equal, CSSM_FALSE if they are not equal.

Notes

GUIDs are returned in the error information of CSSM_GetError. Once you know which type of error is returned (i.e., CSP, CL, TP, DL), you can call CSSM_ListModules to get a list of all the modules that are registered and their GUIDs in order to determine which module set the error. This can be useful for debugging purposes if there is more than one type of module for each type installed on the system.

See Also

CSSM_GetError, CSSM_ListModules

9.2.3 CSSM_DestroyError

CSSM_RETURN CSSMAPI CSSM_DestroyError (void)

This function destroys the error information for a thread/process and frees any necessary memory. It should be called by the function performing clean up before a thread/process exits.

Parameters

None

Return Value

CSSM_OK if the error information was successfully destroyed. If CSSM_FAIL is returned, no error information will be available.

Notes

CSSM_DestroyError does not need to be called if you have loaded the KeyWorks DLL.

See Also

CSSM_InitError

9.2.4 CSSM_GetError

CSSM_ERROR_PTR CSSMAPI CSSM_GetError (void)

This function returns the current error information.

Parameters

None

Return Value

Returns the current error information. If there is no valid error, the error number will be CSSM_OK. A NULL pointer indicates that the CSSM_InitError was not called or that a call to CSSM_DestroyError has been made. No error information is available.

See Also

CSSM_InitError, CSSM_DestroyError, CSSM_ClearError, CSSM_SetError, CSSM_IsCSSMError, CSSM_IsCLError, CSSM_IsTPError, CSSM_IsDLError, CSSM_IsCSPError

9.2.5 CSSM_InitError

CSSM_RETURN CSSMAPI CSSM_InitError (void)

This function initializes the error information for that thread/process and allocates any necessary memory. It should be called by the thread/process initialization function.

Parameters

None

Return Value

CSSM_OK if the error information was successfully initialized. If CSSM_FAIL is returned, no error information will be available.

Notes

CSSM_InitError does not need to be called if you have loaded the KeyWorks DLL.

See Also

CSSM_DestroyError

9.2.6 CSSM_IsCLError

CSSM_BOOL CSSMAPI CSSM_IsCLError (uint32 error_number)

This function determines if *error_number* is within the CL range of errors.

Parameters

error_number (input)
An error number.

Return Value

CSSM_TRUE if the error is a CL error, otherwise CSSM_FALSE.

See Also

CSSM_IsCLError, CSSM_IsCSSMError, CSSM_IsTPError, CSSM_IsCSPError, CSSM_IsKRSPError

9.2.7 CSSM_IsCSPError

CSSM_BOOL CSSMAPI CSSM_IsCSPError (uint32 error_number)

This function determines if *error_number* is within the CSP range of errors.

Parameters

error_number (input)
An error number.

Return Value

CSSM_TRUE if the error is a CSP error, otherwise CSSM_FALSE.

See Also

CSSM_IsCLError, CSSM_IsCSSMError, CSSM_IsTPError, CSSM_IsCSPError, CSSM_IsKRSPError

9.2.8 CSSM_IsCSSMError

CSSM_BOOL CSSMAPI CSSM_IsCSSMError (uint32 error_number)

This function determines if *error_number* is within the KeyWorks range of errors.

Parameters

error_number (input)
An error number.

Return Value

CSSM_TRUE if the error is a KeyWorks error, otherwise CSSM_FALSE.

See Also

CSSM_IsCLError, CSSM_IsCSSMError, CSSM_IsTPError, CSSM_IsCSPError, CSSM_IsKRSPError

9.2.9 CSSM_IsDLError

CSSM_BOOL CSSMAPI CSSM_IsDLError (uint32 error_number)

This function determines if *error_number* is within the DL range of errors.

Parameters

error_number (input)
An error number.

Return Value

CSSM_TRUE if the error is a DL error, otherwise CSSM_FALSE.

See Also

CSSM_IsCLError, CSSM_IsCSSMError, CSSM_IsTPError, CSSM_IsCSPError, CSSM_IsKRSPError

9.2.10 CSSM_IsKRSPError

CSSM_BOOL CSSMAPI CSSM_IsKRSPError (uint32 error_number)

This function determines if *error_number* is within the Key Recovery Service Provider (KRSP) range of errors.

Parameters

error_number (input)
An error number.

Return Value

CSSM_TRUE if the error is a KRSP error, otherwise CSSM_FALSE.

See Also

CSSM_IsCLError, CSSM_IsDLError, CSSM_IsCSSMError, CSSM_IsCSPError

9.2.11 CSSM_IsTPError

CSSM_BOOL CSSMAPI CSSM_IsTPError (uint32 error_number)

This function determines if *error_number* is within the TP range of errors.

Parameters

error_number (input)
An error number.

Return Value

CSSM_TRUE if the error is a TP error; otherwise CSSM_FALSE.

See Also

CSSM_IsCLError, CSSM_IsCSSMError, CSSM_IsTPError, CSSM_IsCSPError, CSSM_IsKRSPError

9.2.12 CSSM_SetError

CSSM_RETURN CSSMAPI CSSM_SetError (CSSM_GUID_PTR guid,
uint32 error)

This function sets the current error information to *error* and *guid*.

Parameters

guid (input)
Pointer to the GUID of the module.

error (input)
An error number. It should fall within one of the valid CSSM, CL, TP, DL, KRSP, or CSP error ranges.

Return Value

CSSM_OK if error was successfully set. A return value of CSSM_FAIL indicates that the error number passed is not within a valid range, the GUID passed is invalid, CSSM_InitError was not called, or CSSM_DestroyError has been called. No error information is available.

See Also

CSSM_InitError, CSSM_DestroyError, CSSM_ClearError, CSSM_GetError

Chapter 10. Application Memory Functions

When IBM KeyWorks or modules return memory structures to applications, that memory is maintained by the application. Instead of using a model where the application passes memory blocks to the modules to work on, the KeyWorks model requires the application to supply memory functions. This has the advantage that applications are not required to know the size of memory blocks they supply to KeyWorks and the add-ins. The memory that the application receives is in its process space, and this prevents the application from walking through the memory of the KeyWorks or the modules. An application that has access to secure memory could supply functions to the Cryptographic Service Provider (CSP) for managing that memory. All data returned from the CSP will be through that secure memory. When the application no longer requires the memory, it is responsible for freeing it.

Applications register their memory functions with the service provider modules during attach time (CSSM_ModuleAttach), and with KeyWorks during initialization (CSSM_Init).

10.1 CSSM_MEMORY_FUNCS and CSSM_API_MEMORY_FUNCS

This structure is used by applications to supply memory functions for KeyWorks and the service provider modules. The functions are used when memory needs to be allocated by KeyWorks or service provider modules for returning data structures to the applications.

```
typedef struct cssm_memory_funcs {
   void * (*malloc_func) (uint32 Size, void *AllocRef);
   void (*free_func) (void *MemPtr, void *AllocRef);
   void * (*realloc_func) (void *MemPtr, uint32 Size, void *AllocRef);
   void * (*calloc_func) (uint32 Num, uint32 Size, void *AllocRef);
   void *AllocRef;
} CSSM_ MEMORY_FUNCS, *CSSM_ MEMORY_FUNCS_PTR;

typedef CSSM_MEMORY_FUNCS CSSM_API_MEMORY_FUNCS;
typedef CSSM_API_MEMORY_FUNCS *CSSM_API_MEMORY_FUNCS_PTR;
```

Definitions:

malloc_func - Pointer to function that returns a void pointer to the allocated memory block of at least *size* bytes from heap *AllocRef.*

free_func - Pointer to function that deallocates a previously-allocated memory block (*memblock*) from heap *AllocRef*

realloc_func - Pointer to function that returns a void pointer to the reallocated memory block (*memblock*) of at least *size* bytes from heap *AllocRef*

calloc_func - Pointer to function that returns a void pointer to an array of *num* elements of length *size* initialized to zero from heap *AllocRef*

AllocRef - Pointer that can be used at the discretion of the application developer to implement additional memory management features such as usage counters.

10.2 Initialization of Memory Structure

The memory structure CSSM_MEMORY_FUNCS requires pointers to functions that implement the memory routines. The example below is an application supplying the C run-time utilities malloc, realloc and free to the memory structure. The memory structure is then used by the CSSM_Init call.

```
    /* Allocating the structure */
MemoryFuncs = (CSSM_MEMORY_FUNCS_PTR) malloc (sizeof (CSSM_MEMORY_FUNCS));

/* Initialize the memory function structure */
MemoryFuncs->malloc_func = HeapMalloc;
MemoryFuncs->realloc_func = HeapRealloc;
MemoryFuncs->free_func = HeapFree;
MemoryFuncs->calloc_func = HeapCalloc;
MemoryFuncs->AllocRef = HeapID;

    /* Initialize the CSSM */
CSSM_Init (Version, MemoryFuncs, NULL);
```

10.3 CSSM_Memory_FUNCS Example

The following two examples are application-defined memory functions. The first example, app_malloc, allocates memory using the system malloc, call and increments a counter palloc_ref, each time the function is called. The memory pointer value returned by malloc is returned to the caller. The second example, app_free, frees memory and decrements the counter palloc_ref.

```
/*****************************************************************************/
void * app_malloc (uint32 size, void *palloc_ref)
{

    if (palloc_ref != NULL)
        *(uint32 *) palloc_ref += 1;
    else
        printf("\tpalloc_ref NULL value passed to allocation function\n");

        return(malloc(size));
}
/*****************************************************************************/

/*****************************************************************************/
void app_free(void * ptr, void *palloc_ref)
{
    if (palloc_ref != NULL)
        * (uint32 *) palloc_ref -= 1;
    else
        printf("\tpallocref NULL value passed to free function\n");
    free(ptr);
    return;
}
/*****************************************************************************/
```

Appendix A. IBM KeyWorks Errors

A.1. Cryptographic Service Provider Module Errors

The following tables provide Cryptographic Service Provider (CSP) module errors.

Table 18. General CSP Messages and Errors

Error Code	Error Name
1001	CSSM_CSP_UNKNOWN_ERROR
1002	CSSM_CSP_REGISTER_ERROR
1003	CSSM_CSP_VERSION_ERROR
1004	CSSM_CSP_CONVERSION_ERROR
1005	CSSM_CSP_NO_TOKENINFO
1006	CSSM_CSP_INTERNAL_ERROR
1007	CSSM_CSP_SERIAL_REQUIRED
1008	CSSM_CSP_NOT_IMPLEMENTED

Table 19. CSP Memory Errors

Error Code	Error Name
1010	CSSM_CSP_MEMORY_ERROR
1011	CSSM_CSP_NOT_ENOUGH_BUFFER
1012	CSSM_CSP_ERR_OUTBUF_LENGTH
1013	CSSM_CSP_NO_OUTBUF
1014	CSSM_CSP_ERR_INBUF_LENGTH
1015	CSSM_CSP_ERR_KEYBUF_LENGTH
1016	CSSM_CSP_NO_SLOT

Table 20. Invalid CSP Parameters

Error Code	Error Name
1020	CSSM_CSP_INVALID_CSP_HANDLE
1021	CSSM_CSP_INVALID_POINTER
1022	CSSM_CSP_INVALID_CERTIFICATE
1023	CSSM_CSP_INVALID_ALGORITHM
1024	CSSM_CSP_INVALID_WINDOW_HANDLE
1025	CSSM_CSP_INVALID_CALLBACK
1026	CSSM_CSP_INVALID_CONTEXT
1027	CSSM_CSP_INVALID_CONTEXT_HANDLE
1028	CSSM_CSP_INVALID_CONTEXT_POINTER
1029	CSSM_CSP_INVALID_DATA_POINTER
1030	CSSM_CSP_INVALID_DATA_COUNT
1031	CSSM_CSP_INVALID_KEY_LENGTH
1032	CSSM_CSP_INVALID_KEY
1033	CSSM_CSP_INVALID_KEY_POINTER
1034	CSSM_CSP_INVALID_ALGORITHM_MODE
1035	CSSM_CSP_INVALID_PADDING
1036	CSSM_CSP_INVALID_KEY_ATTRIBUTE
1037	CSSM_CSP_INVALID_PARAM_LENGTH

Error Code	Error Name
1038	CSSM_CSP_INVALID_IV_SIZE
1039	CSSM_CSP_INVALID_SIGNATURE
1040	CSSM_CSP_INVALID_DEVICE_ID
1041	CSSM_CSP_INVALID_KEYCLASS
1042	CSSM_CSP_INVALID_MODULE_HANDLE
1043	CSSM_CSP_INVALID_KEY_TYPE
1044	CSSM_CSP_INVALID_ITERATION_COUNT

Table 21. File I/O Errors

Error Code	Error Name
1050	CSSM_CSP_FILE_NOT_EXISTS
1051	CSSM_CSP_FILE_NOT_OPEN
1052	CSSM_CSP_FILE_OPEN_FAILED
1053	CSSM_CSP_FILE_CREATE_FAILED
1054	CSSM_CSP_FILE_READ_FAILED
1055	CSSM_CSP_FILE_WRITE_FAILED
1056	CSSM_CSP_FILE_CLOSE_FAILED
1057	CSSM_CSP_FILE_COPY_FAILED
1058	CSSM_CSP_FILE_DELETE_FAILED
1059	CSSM_CSP_FILE_FORMAT_ERROR

Table 22. CSP Cryptographic Errors

Error Code	Error Name
1065	CSSM_CSP_PUBKEY_GET_ERROR
1066	CSSM_CSP_QUERY_SIZE_FAILED
1067	CSSM_CSP_UNKNOWN_ALGORITHM
1068	CSSM_CSP_OPERATION_UNSUPPORTED
1069	CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
1070	CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
1071	CSSM_CSP_KEY_MODULUS_UNSUPPORTED
1072	CSSM_CSP_KEY_LENGTH_UNSUPPORTED
1073	CSSM_CSP_PADDING_UNSUPPORTED
1074	CSSM_CSP_IV_SIZE_UNSUPPORTED
1075	CSSM_CSP_GET_APIMEMFUNC_ERROR
1076	CSSM_CSP_INPUT_LENGTH_OVERSIZE
1077	CSSM_CSP_INPUT_LENGTH_ERROR
1078	CSSM_CSP_INPUT_DATA_ERROR
1079	CSSM_CSP_UNSUPPORTED_STORAGE_MASK
1080	CSSM_CSP_OPERATION_IN_PROGRESS
1081	CSSM_CSP_NO_WRITE_PERMISSIONS
1082	CSSM_CSP_EXCLUSIVE_UNAVAILABLE
1083	CSSM_CSP_UPDATE_WITHOUT_INIT
1084	CSSM_CSP_LOGIN_FAILED
1085	CSSM_CSP_ALREADY_LOGGED_IN
1086	CSSM_CSP_NOT_LOGGED_IN
1087	CSSM_CSP_KEY_PROTECTED
1088	CSSM_CSP_CALLBACK_FAILED
1089	CSSM_CSP_ROUNDS_UNSUPPORTED

Error Code	Error Name
1090	CSSM_CSP_EFFECTIVE_BITS_UNSUPPORTED
1091	CSSM_CSP_INCOMPATIBLE_VERSION
1092	CSSM_CSP_INCOMPATIBLE_KEY_VERSION
1093	CSSM_CSP_ALGORITHM_UNSUPPORTED
1094	CSSM_CSP_OPERATION_FAILED

Table 23. Missing or Invalid CSP Parameters

Error Code	Error Name
1100	CSSM_CSP_PARAM_NO_PARAM
1101	CSSM_CSP_PARAM_NO_PASSWORD
1102	CSSM_CSP_PARAM_NO_SEED
1103	CSSM_CSP_PARAM_NO_KEY
1104	CSSM_CSP_PARAM_NO_SALT
1105	CSSM_CSP_PARAM_NO_MODULUS
1106	CSSM_CSP_PARAM_NO_OUTPUT_SIZE
1108	CSSM_CSP_PARAM_NO_KEY_LENGTH
1109	CSSM_CSP_PARAM_NO_MODE
1110	CSSM_CSP_PARAM_NO_DATA
1111	CSSM_CSP_PARAM_NO_INIT_VECTOR
1112	CSSM_CSP_PARAM_NO_PADDING
1113	CSSM_CSP_PARAM_NO_ROUNDS
1114	CSSM_CSP_PARAM_NO_RANDOM
1115	CSSM_CSP_PARAM_NO_REMAINDATA
1116	CSSM_CSP_PARAM_NO_ALG_PARAMS
1117	CSSM_CSP_PARAM_INVALID_VALUE
1118	CSSM_CSP_PARAM_NO_EFFECTIVE_BITS
1119	CSSM_CSP_PARAM_NO_PRIME
1120	CSSM_CSP_PARAM_NO_BASE
1121	CSSM_CSP_PARAM_NO_SUBPRIME
1122	CSSM_CSP_PARAM_NO_ALG_ID
1123	CSSM_CSP_PARAM_NO_KEY_TYPE
1124	CSSM_CSP_PARAM_NO_ITERATION_COUNT

Table 24. Password Errors

Error Code	Error Name
1130	CSSM_CSP_PASSWORD_INCORRECT
1131	CSSM_CSP_PASSWORD_SAME
1132	CSSM_CSP_PASSWORD_LENGTH_ERROR
1133	CSSM_CSP_PASSWORD_INVALID

Table 25. Key Management Messages and Errors

Error Code	Error Name
1140	CSSM_CSP_PRIKEY_LOAD_ERROR
1141	CSSM_CSP_PRIKEY_NOT_FOUND
1142	CSSM_CSP_PRIKEY_ALREADY_EXIST
1143	CSSM_CSP_PRIKEY_GET_ERROR

Error Code	Error Name
1144	CSSM_CSP_PRIKEY_PUBKEY_INCONSISTENT
1150	CSSM_CSP_KEY_DUPLICATE
1151	CSSM_CSP_KEY_BAD_KEY
1152	CSSM_CSP_KEY_BAD_LENGTH
1153	CSSM_CSP_KEY_NO_PARAM
1154	CSSM_CSP_KEY_ALGID_NOTMATCH
1155	CSSM_CSP_KEY_BLOBTYPE_INCORRECT
1156	CSSM_CSP_KEY_CLASS_INCORRECT
1157	CSSM_CSP_KEY_DELETE_FAILED
1158	CSSM_CSP_KEY_USAGE_INCORRECT
1159	CSSM_CSP_KEY_NOT_PROTECTED
1160	CSSM_CSP_KEY_FORMAT_INCORRECT

Table 26. Random Number Generation (RNG) Messages and Errors

Error Code	Error Name
1200	CSSM_CSP_RNG_FAILED
1201	CSSM_CSP_RNG_UNKNOWN_ALGORITHM
1202	CSSM_CSP_RNG_NO_METHOD

Table 27. Unique ID Generation Messages and Errors

Error Code	Error Name
1220	CSSM_CSP_UIDG_FAILED
1221	CSSM_CSP_UIDG_UNKNOWN_ALGORITHM
1222	CSSM_CSP_UIDG_NO_METHOD

Table 28. Key Generation Messages and Errors

Error Code	Error Name
1210	CSSM_CSP_KEYGEN_FAILED
1211	CSSM_CSP_KEYGEN_UNKNOWN_ALGORITHM
1212	CSSM_CSP_KEYGEN_NO_METHOD

Table 29. Encryption/Decryption Messages

Error Code	Error Name
1230	CSSM_CSP_ENC_UNKNOWN_ALGORITHM
1231	CSSM_CSP_ENC_NO_METHOD
1232	CSSM_CSP_ENC_FAILED
1233	CSSM_CSP_ENC_INIT_FAILED
1234	CSSM_CSP_ENC_UPDATE_FAILED
1235	CSSM_CSP_ENC_FINAL_FAILED
1236	CSSM_CSP_ENC_BAD_IV_LENGTH
1237	CSSM_CSP_ENC_IV_ERROR
1238	CSSM_CSP_ENC_BAD_KEY_LENGTH
1239	CSSM_CSP_ENC_UNKNOWN_MODE
1250	CSSM_CSP_DEC_UNKNOWN_ALGORITHM
1251	CSSM_CSP_DEC_NO_METHOD

Error Code	Error Name
1253	CSSM_CSP_DEC_FAILED
1254	CSSM_CSP_DEC_INIT_FAILED
1255	CSSM_CSP_DEC_UPDATE_FAILED
1256	CSSM_CSP_DEC_FINAL_FAILED
1257	CSSM_CSP_DEC_BAD_IV_LENGTH
1258	CSSM_CSP_DEC_IV_ERROR
1259	CSSM_CSP_DEC_BAD_KEY_LENGTH
1260	CSSM_CSP_DEC_UNKNOWN_MODE

Table 30. Sign/Verify Messages and Errors

Error Code	Error Name
1350	CSSM_CSP_SIGN_UNKNOWN_ALGORITHM
1351	CSSM_CSP_SIGN_NO_METHOD
1352	CSSM_CSP_SIGN_FAILED
1353	CSSM_CSP_SIGN_INIT_FAILED
1354	CSSM_CSP_SIGN_UPDATE_FAILED
1355	CSSM_CSP_SIGN_FINAL_FAILED
1360	CSSM_CSP_VERIFY_FAILED
1361	CSSM_CSP_VERIFY_INIT_FAILED
1362	CSSM_CSP_VERIFY_UPDATE_FAILED
1363	CSSM_CSP_VERIFY_FINAL_FAILED
1365	CSSM_CSP_VERIFY_UNKNOWN_ALGORITHM
1366	CSSM_CSP_VERIFY_NO_METHOD

Table 31. Digest Function Errors

Error Code	Error Name
1380	CSSM_CSP_DIGEST_UNKNOWN_ALGORITHM
1382	CSSM_CSP_DIGEST_NO_METHOD
1383	CSSM_CSP_DIGEST_FAILED
1384	CSSM_CSP_DIGEST_INIT_FAILED
1385	CSSM_CSP_DIGEST_UPDATE_FAILED
1386	CSSM_CSP_DIGEST_CLONE_FAILED
1387	CSSM_CSP_DIGEST_FINAL_FAILED

Table 32. Message Authentication Code (MAC) Function Errors

Error Code	Error Name
1390	CSSM_CSP_MAC_UNKNOWN_ALGORITHM
1392	CSSM_CSP_MAC_NO_METHOD
1393	CSSM_CSP_MAC_FAILED
1394	CSSM_CSP_MAC_INIT_FAILED
1395	CSSM_CSP_MAC_UPDATE_FAILED
1396	CSSM_CSP_MAC_CLONE_FAILED
1397	CSSM_CSP_MAC_FINAL_FAILED

Table 33. Key Exchange Errors

Error Code	Error Name
1410	CSSM_CSP_KEYEXCH_GENPARAM_FAILED
1411	CSSM_CSP_KEYEXCH_PHASE1_FAILED
1412	CSSM_CSP_KEYEXCH_PHASE2_FAILED
1413	CSSM_CSP_KEYEXCH_UNKNOWN_ALGORITHM
1414	CSSM_CSP_KEYEXCH_NO_METHOD

Table 34. PassThrough Custom Errors

Error Code	Error Name
1420	CSSM_CSP_INVALID_PASSTHROUGH_ID
1421	CSSM_CSP_INVALID_PASSTHROUGH_PARAMS

Table 35. Wrap/Unwrap Errors

Error Code	Error Name
1450	CSSM_CSP_WRAP_UNKNOWN_ALGORITHM
1451	CSSM_CSP_WRAP_NO_METHOD
1452	CSSM_CSP_WRAP_FAILED
1456	CSSM_CSP_UNWRAP_UNKNOWN_ALGORITHM
1457	CSSM_CSP_UNWRAP_NO_METHOD
1458	CSSM_CSP_UNWRAP_FAILED

Table 36. Hardware CSP Errors

Error Code	Error Name
1470	CSSM_CSP_DEVICE_ERROR
1471	CSSM_CSP_DEVICE_MEMORY_ERROR
1472	CSSM_CSP_DEVICE_REMOVED
1473	CSSM_CSP_DEVICE_NOT_PRESENT
1474	CSSM_CSP_DEVICE_UNKNOWN
1490	CSSM_CSP_PERMISSIONS_READ_ONLY
1491	CSSM_CSP_PERMISSIONS_WRITE_PROTECT
1492	CSSM_CSP_PERMISSIONS_NOT_EXCLUSIVE

Table 37. Query Size Errors

Error Code	Error Name
1500	CSSM_CSP_QUERY_SIZE_UNKNOWN
1501	CSSM_CSP_QUERY_KEYSIZEINBITS_UNKNOWN

A.2. Certificate Library Module Errors

Table 38. Certificate Library Errors

Error Code	Error Name
3001	CSSM_CL_UNKNOWN_FORMAT
3002	CSSM_CL_UNKNOWN_TAG
3003	CSSM_CL_INVALID_CONTEXT
3004	CSSM_CL_INVALID_CL_HANDLE
3005	CSSM_CL_INVALID_CC_HANDLE
3006	CSSM_CL_INVALID_CERT_POINTER
3007	CSSM_CL_INVALID_FIELD_POINTER
3008	CSSM_CL_INVALID_TEMPLATE
3009	CSSM_CL_INVALID_DATA_POINTER
3010	CSSM_CL_INVALID_SCOPE
3012	CSSM_CL_CERT_CREATE_FAIL
3013	CSSM_CL_CERT_VIEW_FAIL
3014	CSSM_CL_CERT_GET_FIELD_VALUE_FAIL
3015	CSSM_CL_CERT_GET_KEY_INFO_FAIL
3016	CSSM_CL_CERT_IMPORT_FAIL
3017	CSSM_CL_CERT_EXPORT_FAIL
3018	CSSM_CL_PASS_THROUGH_FAIL
3019	CSSM_CL_CERT_DESCRIBE_FORMAT_FAIL
3020	CSSM_CL_UNSUPPORTED_OPERATION
3021	CSSM_CL_MEMORY_ERROR
3022	CSSM_CL_CERT_SIGN_FAIL
3023	CSSM_CL_CERT_UNSIGN_FAIL
3024	CSSM_CL_CERT_VERIFY_FAIL
3025	CSSM_CL_RESULTS_HANDLE
3026	CSSM_CL_INVALID_SIGNER_CERTIFICATE
3027	CSSM_CL_NO_FIELD_VALUES
3028	CSSM_CL_INVALID_CRL_PTR
3029	CSSM_CL_CERT_ABORT_QUERY_FAIL
3030	CSSM_CL_CRL_CREATE_FAIL
3031	CSSM_CL_CRL_SET_FAIL
3032	CSSM_CL_CRL_ADD_CERT_FAIL
3033	CSSM_CL_CRL_REMOVE_CERT_FAIL
3034	CSSM_CL_CRL_SIGN_FAIL
3035	CSSM_CL_CRL_VERIFY_FAIL
3036	CSSM_CL_IS_CERT_IN_CRL_FAIL
3037	CSSM_CL_CRL_GET_FIELD_VALUE_FAIL
3038	CSSM_CL_CRL_ABORT_QUERY_FAIL
3039	CSSM_CL_CRL_DESCRIBE_FORMAT_FAIL
3040	CSSM_CL_INVALID_POINTER
3041	CSSM_CL_INVALID_DATA
3042	CSSM_CL_INITIALIZE_FAIL
3100	CSSM_CL_SIG_NOT_IN_CERT
3101	CSSM_CL_INVALID_REVOKER_CERT_PTR
3102	CSSM_CL_NO_REVOKED_CERTS_IN_CRL
3103	CSSM_CL_CERT_NOT_FOUND_IN_CRL
3104	CSSM_CL_CRL_SIGNSCOPE_NOT_SUPPORTED
3105	CSSM_CL_CRL_VERIFYSCOPE_NOT_SUPPORTED

Error Code	Error Name
3106	CSSM_CL_CRL_NOT_SIGNEDBY_SIGNER
3107	CSSM_CL_CRL_NO_FIELD_OID
3108	CSSM_CL_INVALID_REVOKED_CERT_PTR
3109	CSSM_CL_INVALID_INPUT_PTR
3110	CSSM_CL_KEY_ALGID_NOT_SUPPORTED
3111	CSSM_CL_GET_KEY_ATTRIBUTE_FAIL
3112	CSSM_CL_CERT_ENCODE_FAIL
3113	CSSM_CL_CERT_DECODE_FAIL
3114	CSSM_CL_SIGNATURE_ALGID_NOT_SUPPORTED
3115	CSSM_CL_KEY_FORMAT_UNKNOWN
3116	CSSM_CL_INVALID_CERT_ISSUER_NAME
3117	CSSM_CL_INVALID_CERT_SUBJECT_NAME
3118	CSSM_CL_MISSING_CERT_SUBJECT_NAME
3119	CSSM_CL_MISSING_CERT_ISSUER_NAME
3120	CSSM_CL_MISSING_CERT_VALIDITY
3121	CSSM_CL_MISSING_SUBJECT_PUB_KEY
3122	CSSM_CL_FIELD_NOT_PRESENT

A.3. Data Storage Library Module Errors

Table 39. Data Storage Errors

Error Code	Error Name
5001	CSSM_DL_NOT_LOADED
5002	CSSM_DL_INVALID_DL_HANDLE
5003	CSSM_DL_DATASTORE_NOT_EXISTS
5004	CSSM_DL_MEMORY_ERROR
5005	CSSM_DL_DB_OPEN_FAIL
5006	CSSM_DL_INVALID_DB_HANDLE
5007	CSSM_DL_DB_CLOSE_FAIL
5008	CSSM_DL_DB_CREATE_FAIL
5009	CSSM_DL_DB_DELETE_FAIL
5010	CSSM_DL_INVALID_PTR
5011	CSSM_DL_DB_IMPORT_FAIL
5012	CSSM_DL_DB_EXPORT_FAIL
5013	CSSM_DL_INVALID_CERTIFICATE_PTR
5014	CSSM_DL_CERT_INSERT_FAIL
5015	CSSM_DL_CERTIFICATE_NOT_IN_DB
5016	CSSM_DL_CERT_DELETE_FAIL
5017	CSSM_DL_CERT_REVOKE_FAIL
5018	CSSM_DL_INVALID_SELECTION_PTR
5019	CSSM_DL_NO_CERTIFICATE_FOUND
5020	CSSM_DL_CERT_GETFIRST_FAIL
5021	CSSM_DL_NO_MORE_CERTS
5022	CSSM_DL_CERT_GET_NEXT_FAIL
5023	CSSM_DL_CERT_ABORT_QUERY_FAIL
5024	CSSM_DL_INVALID_CRL_PTR
5025	CSSM_DL_CRL_INSERT_FAIL
5026	CSSM_DL_CRL_NOT_IN_DB
5027	CSSM_DL_CRL_DELETE_FAIL

Error Code	Error Name
5028	CSSM_DL_NO_CRL_FOUND
5029	CSSM_DL_CRL_GET_FIRST_FAIL
5030	CSSM_DL_NO_MORE_CRLS
5031	CSSM_DL_CRL_GET_NEXT_FAIL
5032	CSSM_DL_CRL_ABORT_QUERY_FAIL
5033	CSSM_DL_GET_DB_NAMES_FAIL
5034	CSSM_DL_INVALID_PASSTHROUGH_ID
5035	CSSM_DL_PASS_THROUGH_FAIL
5036	CSSM_DL_INVALID_POINTER
5037	CSSM_DL_NO_DATASOURCES
5038	CSSM_DL_INCOMPATIBLE_VERSION
5039	CSSM_DL_INVALID_FIELD_INFO
5040	CSSM_DL_INVALID_ATTRIBUTE_NAME_FORMAT
5041	CSSM_DL_CONJUNCTIVE_NOT_SUPPORTED
5042	CSSM_DL_OPERATOR_NOT_SUPPORTED
5043	CSSM_DL_NO_MORE_OBJECT
5044	CSSM_DL_INVALID_RESULTS_HANDLE
5045	CSSM_DL_INVALID_ATTRIBUTE_NAME
5046	CSSM_DL_INVALID_ATTRIBUTE
5047	CSSM_DL_UNKNOWN_KEY_TYPE
5048	CSSM_DL_BUFFER_TOO_SMALL
5100	CSSM_DL_INVALID_DATA_POINTER
5101	CSSM_DL_INVALID_DLINFO_POINTER
5102	CSSM_DL_INSTALL_FAIL
5103	CSSM_DL_INVALID_GUID
5104	CSSM_DL_UNINSTALL_FAIL
5105	CSSM_DL_LIST_MODULES_FAIL
5107	CSSM_DL_ATTACH_FAIL
5108	CSSM_DL_DETACH_FAIL
5109	CSSM_DL_GET_INFO_FAIL
5110	CSSM_DL_FREE_INFO_FAIL
5111	CSSM_DL_INVALID_DLINFO_PTR
5112	CSSM_DL_INVALID_CL_HANDLE
5113	CSSM_DL_INVALID_CERTIFICATE_PTR
5114	CSSM_DL_INVALID_CRL
5115	CSSM_DL_INVALID_CRL_POINTER
5116	CSSM_DL_INVALID_RECORD_TYPE
5117	CSSM_DL_DATA_INSERT_FAIL
5118	CSSM_DL_DATA_GETFIRST_FAIL
5119	CSSM_DL_DATA_GETNEXT_FAIL
5120	CSSM_DL_NO_DATA_FOUND
5121	CSSM_DL_INVALID_AUTHENTICATION
5122	CSSM_DL_DATA_ABORT_QUERY_FAIL
5123	CSSM_DL_DATA_DELETE_FAIL

A.4. Trust Policy Module Errors

Table 40. Trust Policy Errors

Error Code	Error Name
7001	CSSM_TP_NOT_LOADED
7002	CSSM_TP_INVALID_TP_HANDLE
7003	CSSM_TP_INVALID_CL_HANDLE
7004	CSSM_TP_INVALID_DL_HANDLE
7005	CSSM_TP_INVALID_DB_HANDLE
7006	CSSM_TP_INVALID_CC_HANDLE
7007	CSSM_TP_INVALID_CERTIFICATE
7008	CSSM_TP_NOT_SIGNER
7009	CSSM_TP_NOT_TRUSTED
7010	CSSM_TP_CERT_VERIFY_FAIL
7011	CSSM_TP_CERTIFICATE_CANT_OPERATE
7012	CSSM_TP_MEMORY_ERROR
7013	CSSM_TP_CERT_SIGN_FAIL
7014	CSSM_TP_INVALID_CRL
7015	CSSM_TP_CERT_REVOKE_FAIL
7016	CSSM_TP_CRL_VERIFY_FAIL
7017	CSSM_TP_CRL_SIGN_FAIL
7018	CSSM_TP_APPLY_CRL_TO_DB_FAIL
7019	CSSM_TP_INVALID_GUID
7020	CSSM_TP_UNISTALL_FAIL
7021	CSSM_TP_INCOMPATIBLE_VERSION
7022	CSSM_TP_INVALID_ACTION
7023	CSSM_TP_VERIFY_ACTION_FAIL
7024	CSSM_TP_INVALID_DATA_POINTER
7025	CSSM_TP_INVALID_ID
7026	CSSM_TP_PASS_THROUGH_FAIL
7027	CSSM_TP_INVALID_CSP_HANDLE
7028	CSSM_TP_ANCHOR_NOT_SELF_SIGNED
7029	CSSM_TP_ANCHOR_NOT_FOUND

A.5. Key Recovery Module Errors

Table 41. Key Recovery Errors

Error Code	Error Name
9001	CSSM_KRSP_AUTHINFO_BUFFER_TOO_SMALL
9002	CSSM_KRSP_COULD_NOT_GET_HOSTINGO
9003	CSSM_KRSP_COULD_NOT_GET_USERID
9004	CSSM_KRSP_CRYPTO_CONTEXT_KEY_NOT_FOUND
9005	CSSM_KRSP_MEMORY_ERROR
9006	CSSM_KRSP_INTEGRITY_CHECK_FAILED
9007	CSSM_KRSP_INTEGRITY_TYPE_NOT_SUPPORTED
9008	CSSM_KRSP_INVALID_AUTHINFO_BUFFER
9009	CSSM_KRSP_INVALID_CRYPTO_CONTEXT
9010	CSSM_KRSP_INVALID_CRYPTO_CONTEXT_KEY
9011	CSSM_KRSP_INVALID_JURIS_PROFILE

Error Code	Error Name
9012	CSSM_KRSP_INVALID_KRCONTEXT
9013	CSSM_KRSP_INVALID_KRSP_CONFIG
9014	CSSM_KRSP_INVALID_KRTYPE
9015	CSSM_KRSP_INVALID_LOCAL_KRPROFILE
9016	CSSM_KRSP_KRPROFILE_ATTRIBUTE_NOT_FOUND
9017	CSSM_KRSP_LEMAN_GEN_REQUIRED
9018	CSSM_KRSP_LEUSE_GEN_REQUIRED
9019	CSSM_KRSP_ENT_GEN_REQUIRED

A.6. KeyWorks Framework Errors

Table 42. Memory Allocation Errors

Error Code	Error Name
10001	CSSM_MALLOC_FAILED
10002	CSSM_CALLOC_FAILED
10003	CSSM_REALLOC_FAILED

Table 43. File I/O Errors

Error Code	Error Name
10010	CSSM_FWRITE_FAILED
10011	CSSM_FREAD_FAILED
10012	CSSM_CANT_FSEEK
10013	CSSM_INVALID_FILE_PTR
10014	CSSM_END_OF_FILE

Table 44. Miscellaneous Errors

Error Code	Error Name
10020	CSSM_CANT_GET_USER_NAME
10021	CSSM_GETCWD_FAILED
10022	CSSM_ENV_VAR_NOT_FOUND

Table 45. Dynamic Library Errors

Error Code	Error Name
10030	CSSM_FREE_LIBRARY_FAILED
10031	CSSM_LOAD_LIBRARY_FAILED
10032	CSSM_CANT_GET_PROC_ADDR
10033	CSSM_CANT_GET_MODULE_HANDLE
10034	CSSM_CANT_GET_MODULE_FILE_NAME
10035	CSSM_INVALID_LIB_HANDLE
10036	CSSM_BAD_MODULE_HANDLE

Table 46. Registry Errors

Error Code	Error Name
10040	CSSM_CANT_CREATE_KEY
10041	CSSM_CANT_SET_VALUE
10042	CSSM_CANT_GET_VALUE
10043	CSSM_CANT_DELETE_SECTION
10044	CSSM_CANT_DELETE_KEY
10045	CSSM_CANT_ENUM_KEY
10046	CSSM_CANT_OPEN_KEY
10047	CSSM_CANT_QUERY_KEY

Table 47. Mutex/Synchronization Errors

Error Code	Error Name
10050	CSSM_CANT_CREATE_MUTEX
10051	CSSM_LOCK_MUTEX_FAILED
10052	CSSM_TRYLOCK_MUTEX_FAILED
10053	CSSM_UNLOCK_MUTEX_FAILED
10054	CSSM_CANT_CLOSE_MUTEX
10055	CSSM_INVALID_MUTEX_PTR

Table 48. Shared Memory File Errors

Error Code	Error Name
10060	CSSM_CANT_CREATE_SHARED_MEMORY_FILE
10061	CSSM_CANT_OPEN_SHARED_MEMORY_FILE
10062	CSSM_CANT_MAP_SHARED_MEMORY_FILE
10063	CSSM_CANT_UNMAP_SHARED_MEMORY_FILE
10064	CSSM_CANT_FLUSH_SHARED_MEMORY_FILE
10065	CSSM_CANT_CLOSE_SHARED_MEMORY_FILE
10066	CSSM_INVALID_PERMS
10067	CSSM_BAD_FILE_HANDLE
10068	CSSM_BAD_FILE_ADDR

Table 49. General Errors

Error Code	Error Name
10100	CSSM_BAD_PTR_PASSED

Table 50. KeyWorks API Errors

Error Code	Error Name
10301	CSSM_INVALID_POINTER
10302	CSSM_EXPIRED
10303	CSSM_MEMORY_ERROR
10304	CSSM_INVALID_ATTRIBUTE
10305	CSSM_NOT_INITIALIZE
10306	CSSM_INSTALL_FAIL

Error Code	Error Name
10307	CSSM_REGISTRY_ERROR
10308	CSSM_INVALID_CONTEXT_HANDLE
10309	CSSM_INVALID_CSP_HANDLE
10310	CSSM_INVALID_TP_HANDLE
10311	CSSM_INVALID_CL_HANDLE
10312	CSSM_INVALID_DL_HANDLE
10313	CSSM_INCOMPATIBLE_VERSION
10314	CSSM_ATTACH_FAIL
10315	CSSM_NO_ADDIN
10316	CSSM_FUNCTION_NOT_IMPLEMENTED
10317	CSSM_INVALID_CONTEXT_POINTER
10318	CSSM_INVALID_MANIFEST_ATTRIB_POINTER
10319	CSSM_MODE_UNSUPPORTED
10320	CSSM_KEY_LENGTH_UNSUPPORTED
10321	CSSM_IV_SIZE_UNSUPPORTED
10322	CSSM_PADDING_UNSUPPORTED
10323	CSSM_KEY_MODULUS_UNSUPPORTED
10324	CSSM_PARAM_NO_KEY
10325	CSSM_INVALID_KRSP_HANDLE
10326	CSSM_KR_FIELDS_NOT_GENERATED
10327	CSSM_ENT_KR_POLICY_MODULE_NOT_FOUND
10328	CSSM_ENT_KR_POLICY_FUNC_NOT_FOUND
10329	CSSM_LE_POLICY_MODULE_CORRUPT
10330	CSSM_ENT_POLICY_MODULE_CORRUPT
10340	CSSM_INVALID_SERVICE_MASK
10341	CSSM_INVALID_SUBSERVICEID
10342	CSSM_INVALID_INFO_LEVEL
10343	CSSM_MULTIPLE_ENCRYPT_ATTEMPT
10344	CSSM_ADDIN_AUTHENTICATION_FAILED
10345	CSSM_EISL_PKCS7_INVALID
10346	CSSM_EISL_SIGROOT_INVALID
10347	CSSM_EISL_MANIFEST_SECTION_NOT_FOUND
10348	CSSM_EISL_MODULE_VERIFICATION_FAILED
10349	CSSM_EISL_MODULE_LOAD_FAILED
10500	CSSM_INVALID_ADDIN_HANDLE
10501	CSSM_INVALID_GUID

Table 51. KeyWorks Privilege Mechanism Errors

Error Code	Error Name
10350	CSSM_INVALID_CREDENTIALS
10351	CSSM_NOT_AUTHORIZED
10352	CSSM_STRONG_CRYPTO_NOT_ALLOWED
10353	CSSM_CANT_GET_THREAD_ID
10354	CSSM_THREAD_EXEMPTION_ERROR
10355	CSSM_CANT_CREATE_CLEANUP_THREAD
10356	CSSM_PRIV_NOT_INITIALIZED
10357	CSSM_INVALID_NAME

Appendix B. List of Acronyms

AI	Authentication Information
API	Application Programming Interface
CA	Certificate Authority
CCA	Common Cryptographic Architecture
CDSA	Common Data Security Architecture
CL	Certificate Library
CPS	Certification Practice Statement
CRL	Certificate Revocation List
CSP	Cryptographic Service Provider
CSSM	Common Security Services Manager
DB	Database
DBMS	Database Management System
DES	Data Encryption Standard
DL	Data Storage Library
DLL	Dynamic Link Library
DSA	Digital Signature Algorithm
ECB	Electronic Code Book
EDI	Electronic Data Interchange
GUID	Globally Unique ID
IDEA	International Data Encryption Algorithm
IPSEC	Internet Protocol Security
ISO	International Organization for Standardization
ISV	Independent Software Vendor
KRA	Key Recovery Agent
KRB	Key Recovery Block
KRF	Key Recovery Field
KRPT	Key Recovery Policy Table
KRS	Key Recovery Server
KRSP	Key Recovery Service Provider
MAC	Message Authentication Code
MSM	Multi-Service Module
OAEP	Optimal Asymmetric Encryption Padding
OID	Object Identifier
PKCS	Public-Key Cryptographic Standard
PKI	Public Key Infrastructure
RNG	Random Number Generation
S/MIME	Secure/Multipurpose Internet Mail Extensions

SDSI	Simple Distributed Security Infrastructure
SET	Secure Electronic Transaction
SPI	Service Provider Interface
SSL	Secure Sockets Layer
TP	Trust Policy
URL	Uniform Resource Locator
UTC	Coordinated Universal Time

Appendix C. Glossary

Asymmetric algorithms	Cryptographic algorithms, where one key is used to encrypt and a second key is used to decrypt. They are often called public-key algorithms. One key is called the public key, and the other is called the private key or secret key. RSA (Rivest-Shamir-Adelman) is the most commonly used public-key algorithm. It can be used for encryption and for signing.
Authentication Information	Information that is verified for authentication. For example, a Key Recovery Officer (KRO) selects a password which will be used for authentication with the Key Recovery Coordinator (KRC). A KRO operator who has identification information must search the Authentication Information (AI) database to locate an AI value that corresponds to the individual who generated the information.
Certificate	See Digital certificate.
Certificate Authority	An entity that guarantees or sponsors a certificate. For example, a credit card company signs a cardholder's certificate to assure that the cardholder is who he or she claims to be. The credit card company is a Certificate Authority (CA). CAs issue, verify, and revoke certificates.
Certificate chain	The hierarchical chain of all the other certificates used to sign the current certificate. This includes the CA who signs the certificate, the CA who signed that CA's certificate, and so on. There is no limit to the depth of the certificate chain.
Certificate signing	The CA can sign certificates it issues or co-sign certificates issued by another CA. In a general signing model, an object signs an arbitrary set of one or more objects. Hence, any number of signers can attest to an arbitrary set of objects. The arbitrary objects could be, for example, pieces of a document for libraries of executable code.
Certificate validity date	A start date and a stop date for the validity of the certificate. If a certificate expires, the CA may issue a new certificate.
Cryptographic algorithm	A method or defined mathematical process for implementing a cryptography operation. A cryptographic algorithm may specify the procedure for encrypting and decrypting a byte stream, digitally signing an object, computing the hash of an object, generating a random number, etc. IBM KeyWorks accommodates Data Encryption Standard (DES), RC2, RC4, International Data Encryption Algorithm (IDEA), and other encryption algorithms.
Cryptographic Service Provider	Cryptographic Service Providers (CSPs) are modules that provide secure key storage and cryptographic functions. The modules may be software only or hardware with software drivers. The cryptographic functions provided may include: • Bulk encryption and decryption • Digital signing

- Cryptographic hash
- Random number generation
- Key exchange

Cryptography

The science for keeping data secure. Cryptography provides the ability to store information or to communicate between parties in such a way that prevents other non-involved parties from understanding the stored information or accessing and understanding the communication. The encryption process takes understandable text and transforms it into unintelligible piece of data (called ciphertext); the decryption process restores the understandable text from the unintelligible data. Both involve a mathematical formula or algorithm and a secret sequence of data called a key. Cryptographic services provide confidentiality (keeping data secret), integrity (preventing data from being modified), authentication (proving the identity of a resource or a user), and non-repudiation (providing proof that a message or transaction was sent and/or received).

There are two types of cryptography:

- In shared/secret key (symmetric) cryptography there is only one key that is a shared secret between two communicating parties. The same key is used for encryption and decryption.
- In public key (asymmetric) cryptography different keys are used for encryption and decryption. A party has two keys: a public key and a private key. The two keys are mathematically related, but it is virtually impossible to derive the private key from the public key. A message that is encrypted with someone's key (obtained from some public directory) can only be decrypted with the associated private key. Alternately, the private key can be used to "sign" a document; the public key can be used as verification of the source of the document.

Cryptoki

Short for cryptographic token interface. See Token.

Data Encryption Standard

In computer security, the National Institute of Standards and Technology (NIST) Data Encryption Standard (DES), adopted by the U.S. Government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementations of the data encryption algorithm.

Digital certificate

The binding of some identification to a public key in a particular domain, as attested to directly or indirectly by the digital signature of the owner of that domain. A digital certificate is an unforgettable credential in cyberspace. The certificate is issued by a trusted authority, covered by that party’s digital signature. The certificate may attest to the certificate holder’s identity, or may authorize certain actions by the certificate holder. A certificate may include multiple signatures and may attest to multiple objects or multiple actions.

Digital signature

A data block that was created by applying a cryptographic signing algorithm to some other data using a secret key. Digital signatures may be used to:

- Authenticate the source of a message, data, or document
- Verify that the contents of a message has not been modified since it was signed by the sender

- Verify that a public key belongs to a particular person

Typical digital signing algorithms include MD5 with RSA encryption, and DSS, the proposed Digital Signature Standard defined as part of the U.S. Government Capstone project.

Enterprise

A company or individual who is authorized to submit on-line requests to the Key Recovery Officer (KRO). In the enterprise key recovery scenario, a process at the enterprise called the KRO is responsible for preparing key recovery requests and communicating them to the KRC. The KRO, acting on behalf of an enterprise or individual, sends an on-line request to the Key Recovery Coordinator (KRC) to recover a key from a Key Recovery Block (KRB).

Graphical User Interface

A type of display format that enables the user to choose commands, start programs, and see lists of files and other options by pointing to pictorial representations (icons) and lists of menu items on the screen. Graphical User Interfaces (GUIs) are used by the Microsoft Windows program for IBM-compatible microcomputers and by other systems.

Hash algorithm

A cryptographic algorithm used to hash a variable-size input stream into a unique, fixed-sized output value. Hashing is typically used in digital signing algorithms. Example hash algorithms include MD and MD2 from RSA Data Security. MD5, also from RSA Data Security, hashes a variable-size input stream into a 128-bit output value. SHA, a Secure Hash Algorithm published by the U.S. Government, produces a 160-bit hash value from a variable-size input stream.

IBM KeyWorks Architecture

A set of layered security services that address communications and data security problems in the emerging PC business space.

IBM KeyWorks Framework

The IBM KeyWorks Framework defines five key service components:

- Cryptographic Module Manager
- Key Recovery Module Manager
- Trust Policy Module Manager
- Certificate Library Module Manager
- Data Storage Library Module Manager

IBM KeyWorks binds together all the security services required by PC applications. In particular, it facilitates linking digital certificates to cryptographic actions and trust protocols.

Key Escrow

The storing of a key (or parts of a key) with a trusted party or trusted parties in case of loss or destruction of the key.

Key Recovery Agent

The Key Recovery Agent (KRA) acts as the back end for a key recovery operation. The KRA can only be accessed through an on-line communication protocol via the Key Recovery Coordinator (KRC). KRAs are considered outside parties involved in the key recovery process; they are analogous to the neighbors who each hold one digit of the combination of the lock box containing the key. The authorized parties (i.e., enterprise or law enforcement) have the freedom to choose the number of specific KRAs that they want to use. The authorized party requests that each KRA decrypt its section of the Key Recovery Fields (KRFs) that is associated with the transmission. Then those pieces of information are used in the process that derives the session key. The KRA will only be able to recover a portion of the key, and reading the original message will require searching the remaining key space in order to find the key that will decrypt the message. The number of KRAs on each end of the communication does not have to be equal.

Key Recovery Block

The Key Recovery Block (KRB) is a piece of encrypted information that is contained within a block. The KRS components (i.e., KRO, KRC, KRA) work collectively to recover a session key from a provided KRB. In the enterprise scenario, the KRO has both the KRB and the credentials that authenticate it to receive the recovered key. This information will be transmitted over the network to the KRC. In the law enforcement scenario, the KRB is presented on a 3.5-inch diskette, and the credentials are in the physical form of a legal warrant. This warrant will specify any information available to the law enforcement agents which can be used to tie the warrant to the identity of the user for whom KRBs were generated (i.e., username, hostname, IP address). The KRC has the ability to check credentials and derive the original encryption key from the KRB with the help of its KRAs.

Key Recovery Coordinator

The Key Recovery Coordinator (KRC) acts as the front end for the key recovery operation. The KRO, acting on behalf of an enterprise or individual, sends an on-line request to the KRC to recover a key from a KRB. The KRC receives the on-line request and services it by interacting with the appropriate set of KRAs as specified within the KRB. The recovered key is then sent back to the KRO by the KRC using an on-line protocol. The KRC consists of one main application which, when started, behaves as a server process. The system, which serves as the KRC, may be configured to start the KRC application as part of system services; alternatively, the KRC operator can start up the KRC application manually. The KRC application performs the following operations:

- Listens for on-line recovery requests from KRO
- Can be used to launch an embedded application that allows manual key recovery for law enforcement
- Monitors and displays the status of the recovery requests being serviced

Key Recovery Field

A Key Recovery Field (KRF) is a block of data that is created from a symmetric key and key recovery profile information. The Key Recovery Service Provider (KRSP) is invoked from the IBM KeyWorks framework to create the KRFs. There are two major pieces of the key recovery fields: block 1 contains information that is unrelated to the session key of the transmitted message, and encrypted with the public keys of the selected key recovery agents; block 2 contains information that is related to the session key of the transmission. The

KRSP generates the KRFs for the session key. This information is *not* the key or any portion of the key, but is information that can be used to recover the key. The KRSP has access to location-unique jurisdiction policy information that controls and modifies some of the steps in the generation of the KRFs. Only once the KRFs are generated, and both the client and server sides have access to them, can the encrypted message flow begin. KRFs are generated so that they can be used by a KRA to recover the original symmetric key, either because the user who generated the message has lost the key, or at the warranted request of law enforcement agents.

Key Recovery Module Manager

The Key Recovery Module Manager enables key recovery for cryptographic services obtained through the IBM KeyWorks. It mediates all cryptographic services provided by the KeyWorks and applies the appropriate key recovery policy on all such operations. The Key Recovery Module Manager contains a Key Recovery Policy Table (KRPT) that defines the applicable key recovery policy for all cryptographic products. The Key Recovery Module Manager routes the KR-API function calls made by an application to the appropriate KR-SPI functions. The Key Recovery Module Manager also enforces the key recovery policy on all cryptographic operations that are obtained through the IBM KeyWorks. It maintains key recovery state in the form of key recovery contexts.

Key Recovery Officer

An entity called the Key Recovery Officer (KRO) is the focal point of the key recovery process. In the enterprise key recovery scenario, the KRO is responsible for preparing key recovery requests and communicating them to the KRC. The KRO has both the KRB and the credentials that authenticate it to receive the recovered key. The KRO is the entity that acts on behalf of an enterprise to initiate a key recovery request operation. An employee within an enterprise who desires key recovery will send a request to the KRO with the KRB that is to be recovered. The actual key recovery phase begins when the KRO operator uses the KRO application to initiate a key recovery request to the appropriate KRC. At this time, the operator selects a KRB to be sent for recovery, enters the Authentication Information (AI) that can be used to authenticate the request to the KRC, and submits the request.

Key Recovery Policy

Key recovery policies are mandatory policies that are typically derived from jurisdiction-based regulations on the use of cryptographic products for data confidentiality. Often, the jurisdictions for key recovery policies coincide with the political boundaries of countries in order to serve the law enforcement and intelligence needs of these political jurisdictions. Political jurisdictions may choose to define key recovery policies for cryptographic products based on export, import, or use controls. Enterprises may define internal and external jurisdictions, and may mandate key recovery policies on the cryptographic products within their own jurisdictions.

	Key recovery policies come in two flavors: *key recovery enablement policies* and *key recovery interoperability policies.* Key recovery enablement policies specify the exact cryptographic protocol suites (e.g., algorithms, modes, key lengths, etc.) and perhaps usage scenarios, where key recovery enablement is mandated. Furthermore, these policies may also define the number of bits of the cryptographic key that may be left out of the key recovery enablement operation; this is typically referred to as the *workfactor*. Key recovery interoperability policies specify to what degree a key recovery enabled cryptographic product is allowed to interoperate with other cryptographic products.
Key Recovery Server	The Key Recovery Server (KRS) consists of three major entities: Key Recovery Coordinator (KRC), Key Recovery Agent (KRA), and Key Recovery Officer (KRO). The KRS is intended to be used by enterprise employees and security personnel, law enforcement personnel, and KRSF personnel. The KRS interacts with one or more local or remote KRAs to reconstruct the secret key that can be used to decrypt the ciphertext.
Key Recovery Server Facility	The Key Recovery Server Facility (KRSF) is a facility room that houses the KRS component facilities, ensuring they operate within a secure environment that is highly resistant to penetration and compromise. Several physical and administrative security procedures must be followed at the KRSF such as a combination keyed lock, limited personnel, standalone system, operating system with security features (Microsoft NT Workstation 4.0), NTFS (Windows NT Filesystem), and account and auditing policies.
Key Recovery Service Provider	Key Recovery Service Providers (KRSPs) are modules that provide key recovery enablement functions. The cryptographic functions provided may include: • Key recovery field generation • Key recovery field processing
Law Enforcement	A type of scenario where key recovery is mandated by the jurisdictional law enforcement authorities in the interest of national security and law enforcement. In the law enforcement scenario, the KRB is presented on a 3.5-inch diskette, and the credentials are in the physical form of a legal warrant. This warrant will specify any information available to the law enforcement agents which can be used to tie the warrant to the identity of the user for whom KRBs were generated (i.e., username, hostname, IP address).
Leaf certificate	The certificate in a certificate chain that has not been used to sign another certificate in that chain. The leaf certificate is signed directly or transitively by all other certificates in the chain.
Message digest	The digital fingerprint of an input stream. A cryptographic hash function is applied to an input message arbitrary length and returns a fixed-size output, which is called the digest value.

Owned certificate	A certificate whose associated secret or private key resides in a local Cryptographic Service Provider (CSP). Digital-signing algorithms require using owned certificates when signing data for purposes of authentication and non-repudiation. A system may use certificates it does not own for purposes other than signing.
Private key	The cryptographic key is used to decipher messages in public-key cryptography. This key is kept secret by its owner.
Public key	The cryptographic key is used to encrypt messages in public-key cryptography. The public key is available to multiple users (i.e., the public).
Random number generator	A function that generates cryptographically strong random numbers that cannot be easily guessed by an attacker. Random numbers are often used to generate session keys.
Root certificate	The prime certificate, such as the official certificate of a corporation or government entity. The root certificate is positioned at the top of the certificate hierarchy in its domain, and it guarantees the other certificates in its certificate chain. Each Certificate Authority (CA) has a self-signed root certificate. The root certificate's public key is the foundation of signature verification in its domain.
Secure Electronic Transaction	A mechanism for securely and automatically routing payment information among users, merchants, and their banks. Secure Electronic Transaction (SET) is a protocol for securing bankcard transactions on the Internet or other open networks using cryptographic services. SET is a specification designed to utilize technology for authenticating parties involved in payment card purchases on any type of on-line network, including the Internet. SET was developed by Visa and MasterCard, with participation from leading technology companies, including Microsoft, IBM, Netscape, SAIC, GTE, RSA, Terisa Systems, and VeriSign. By using sophisticated cryptographic techniques, SET will make cyberspace a safer place for conducting business and is expected to boost consumer confidence in electronic commerce. SET focuses on maintaining confidentiality of information, ensuring message integrity, and authenticating the parties involved in a transaction. The significance of SET, over existing Internet security protocols, is found in the use of digital certificates. Digital certificates will be used to authenticate all the parties involved in a transaction. SET will provide those in the virtual world with the same level of trust and confidence a consumer has today when making a purchase at any of the 13 million Visa-acceptance locations in the physical world. The SET specification is open and free to anyone who wishes to use it or develop SET-compliant software for buying or selling in cyberspace.

Security Context	A control structure that retains state information shared between a CSP and the application agent requesting service from the CSP. Only one context can be active for an application at any given time, but the application is free to switch among contexts at will, or as required. A security context specifies CSP and application-specific values, such as required key length and desired hash functions.
Security-relevant event	An event where a CSP-provided function is performed, a security module is loaded, or a breach of system security is detected.
Session key	A cryptographic key used to encrypt and decrypt data. The key is shared by two or more communicating parties, who use the key to ensure privacy of the exchanged data.
Signature	See Digital signature.
Signature chain	The hierarchical chain of signers, from the root certificate to the leaf certificate, in a certificate chain.
Smart Card	A device (usually similar in size to a credit card) that contains an embedded microprocessor that could be used to store information. Smart cards can store credentials used to authenticate the holder.
S/MIME	Secure/Multipurpose Internet Mail Extensions (S/MIME) is a protocol that adds digital signatures and encryption to Internet MIME messages. MIME is the official proposed standard format for extended Internet electronic mail. Internet e-mail messages consist of two parts, the header and the body. The header forms a collection of field/value pairs structured to provide information essential for the transmission of the message. The body is normally unstructured unless the e-mail is in MIME format. MIME defines how the body of an e-mail message is structured. The MIME format permits e-mail to include enhanced text, graphics, audio, and more in a standardized manner via MIME-compliant mail systems. However, MIME itself does not provide any security services. The purpose of S/MIME is to define such services, following the syntax given in PKCS #7 for digital signatures and encryption. The MIME body part carries a PKCS #7 message, which itself is the result of cryptographic processing on the other MIME body parts.
Symmetric algorithms	Cryptographic algorithms that use a single secret key for encryption and decryption. Both the sender and receiver must know the secret key. Well-known symmetric functions include Data Encryption Standard (DES) and International Data Encryption Algorithm (IDEA). The U.S. Government endorsed DES as a standard in 1977. It is an encryption block cipher that operates on 64-bit blocks with a 56-bit key. It is designed to be implemented in hardware, and works well for bulk encryption. IDEA, one of the best known public algorithms, uses a 128-bit key.

Token	The logical view of a cryptographic device, as defined by a CSP's interface. A token can be hardware, a physical object, or software. A token contains information about its owner in digital form, and about the services it provides for electronic-commerce and other communication applications. A token is a secure device. It may provide a limited or a broad range of cryptographic functions. Examples of hardware tokens are smart cards and Personal Computer Memory Card International Association (PCMCIA) cards.
Verification	The process of comparing two message digests. One message digest is generated by the message sender and included in the message. The message recipient computes the digest again. If the message digests are exactly the same, it shows or proves there was no tampering of the message contents by a third party (between the sender and the receiver).
Web of trust	A trust network among people who know and communicate with each other. Digital certificates are used to represent entities in the web of trust. Any pair of entities can determine the extent of trust between the two, based on their relationship in the web. Based on the trust level, secret keys may be shared and used to encrypt and decrypt all messages exchanged between the two parties. Encrypted exchanges are private, trusted communications.