z/OS

Open Cryptographic Services Facility Service Provider Module Developer's Guide and Reference

Version 2 Release 1

SC14-7514-00

Note
Before using this information and the product it supports, read the information in "Notices" on page 117.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

Preface

The Open Cryptographic Services Facility (OCSF) is a derivative of the IBM Keyworks technology which is an implementation of the Common Data Security Architecture (CDSA) for applications running in the UNIX Services environment. It is an extensible architecture that provides mechanisms to manage service provider security modules, which use cryptography as a computational base to build security protocols and security systems. Figure 1 shows the four basic layers of the OCSF: Application Domains, System Security Services, OCSF Framework, and Service Providers. The OCSF Framework is the core of this architecture. It provides a means for applications to directly access security services through the OCSF security application programming interface (API), or to indirectly access security services via layered security services and tools implemented over the OCSF API. The OCSF Framework manages the service provider security modules and directs application calls through the OCSF API to the selected service provider module that will service the request. The OCSF API defines the interface for accessing security services. The OCSF service provider interface (OCSF SPI) defines the interface for service providers who develop plug-able security service products.

Service providers perform various aspects of security services, including:
- Cryptographic Services[1]
- Trust Policy Libraries
- Certificate Libraries
- Data Storage Libraries.

Cryptographic Service Providers (CSPs) are service provider modules that perform cryptographic operations including encryption, decryption, digital signing, key pair generation, random number generation, and key exchange. Trust Policy (TP) modules implement policies defined by authorities and institutions, such as VeriSign (as a Certificate Authority (CA)) or MasterCard (as an institution). Each TP module embodies the semantics of a trust model based on using digital certificates as credentials. Applications may use a digital certificate as an identity credential and/or an authorization credential. Certificate Library (CL) modules provide format-specific, syntactic manipulation of memory-resident digital certificates and Certificate Revocation Lists (CRLs). Data Storage Library (DL) modules provide persistent storage for certificates and CRLs.

Service provider modules

An OCSF service provider module is a Dynamically Linked Library (DLL) composed of functions that implement some or all of the OCSF module interfaces. Applications directly or indirectly select the modules used to provide security services to the application. Independent Software Vendors (ISVs) and hardware vendors will provide these service providers. The functionality of the service providers may be extended beyond the services defined by the OCSF API, by exporting additional services to applications using an OCSF PassThrough mechanism.

1. If you want to provide a Cryptographic Service Provider, you need to contact IBM. For more information, see "Writing a cryptographic service provider" on page xiv.

The API calls defined for service provider modules are categorized as service operations, module management operations, and module-specific operations. Service operations include functions that perform a security operation such as encrypting data, inserting a CRL into a data source, or verifying that a certificate is trusted. Module management functions support module installation, registration of module features and attributes, and queries to retrieve information on module availability and features.

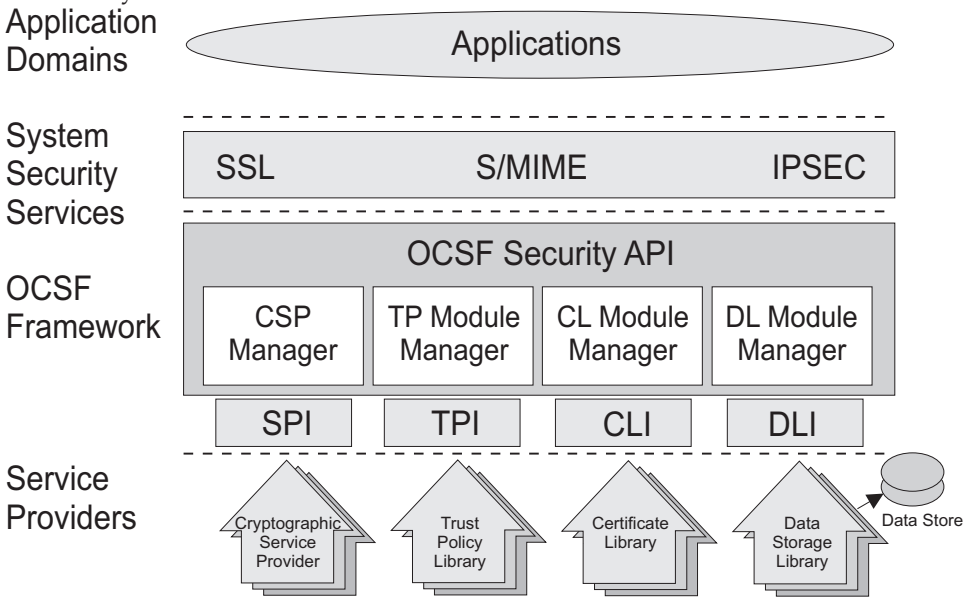

Figure 1. Open Cryptographic Services Facility Architecture.

Module-specific operations are enabled in the API through passthrough functions whose behavior and use is defined by the service provider module developer.

Each module, regardless of the security services it offers, has the same set of module management responsibilities. Every module must expose functions that allow OCSF to indicate events such as module attach and detach. In addition, as part of the attach operation, every module must be able to verify its own integrity, verify the integrity of OCSF, and register with OCSF. Detailed information about service provider module structure, administration, and interfaces are found in this book.

Who should use this information

This book should be used by Independent Software Vendors (ISVs) who want to develop their own service provider modules. These ISVs can be highly experienced software and security architects, advanced programmers, and sophisticated users. The intended audience of this document must be familiar with high-end cryptography and digital certificates. They must also be familiar with local and foreign government regulations on the use of cryptography and the implication of those regulations for their applications and products. We assume that this audience is familiar with the basic capabilities and features of the protocols they are considering.

Conventions used in this information

This book uses the following typographic conventions:

Bold **Bold** words or characters represent system elements that you must enter into the system literally, such as commands.

Italic *Italicized* words or characters represent values for variables that you must supply.

`Example Font`
Examples and information displayed by the system are printed using an example font that is a `constant width typeface`.

Where to find more information

This book describes the features common to all OSCF service provider modules. It defines the interfaces for certificate, trust, and data library service providers. Service provider developers must conform to these interfaces in order for the individual service provider modules to be accessible through the OCSF framework.

The *z/OS Open Cryptographic Services Facility Application Programming* provides an overview of the OCSF. It explains how to integrate OCSF into applications and contains a sample OCSF application. It also defines the interfaces that application developers employ to access security services provided by the OCSF framework and service provider modules. Specific information about the individual service providers is also provided.

For complete titles and order numbers of the books for all products that are part of z/OS see the *z/OS Information Roadmap*, SA22-7500.

Internet sources

The softcopy z/OS publications are also available for web-browsing and for viewing or printing PDFs using the following URL:

`http://www.ibm.com/systems/z/os/zos/bkserv/`

You can also provide comments about this book and any other z/OS documentation by visiting that URL. Your feedback is important in helping to provide the most accurate and high-quality information.

Writing a cryptographic service provider

If you want to write your own Cryptographic Service Provider (CSP) you need to contact IBM using one of the following methods:

- Call the Solution Developer Program Hotline at 1-770-835-9902 (worldwide) or 1-800-627-8363 (US and Canada), ask for the zEnterprise Administrator
- Access the new zEnterprise home page at:

  `http://www.ibm.com/systems/z/`

  and use the feedback form to make a request.

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or provide any other feedback that you have.

Use one of the following methods to send your comments:

1. Send an email to mhvrcfs@us.ibm.com.
2. Send an email from the Contact z/OS.
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   US
4. Fax the comments to us, as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:
  z/OS OCSF Module Developer's Guide and Reference
  SC14-7514-00
- The topic and page number that is related to your comment.
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one of the following actions:

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at IBM support portal.

z/OS Version 2 Release 1 summary of changes

See the following publications for all enhancements to z/OS Version 2 Release 1 (V2R1):

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

Chapter 1. Module structure and administration

Service provider modules are composed of module administration components and implementation of security service interfaces in one or more categories of service. Module administration components include the tasks required during module installation, attach, and detach. The module developer determines the number, categories, and contents of the service implementation. Both the administration components and service interfaces are discussed in the following sections.

Export

Any application you create and export or re-export from the U.S. utilizing the Open Cryptographic Services Facility Cryptographic Services may be subject to special export licensing requirements by the Bureau of Export Administration of the U.S. Department of Commerce.

Security services

The primary components of a service provider module are the security services that it offers. A service provider module may provide one to four categories of service, with each service having one or more available subservices. The service categories are Cryptographic Service Provider (CSP)[2] services, Trust Policy (TP) services, Certificate Library (CL) services, and Data Storage Library (DL) services. A subservice consists of a unique set of capabilities within a certain service. For example, in a CSP service providing access to hardware tokens, each subservice would represent a slot. A TP service may have one subservice that supports the Secure Electronic Transfer (SET) Merchant TP and a second subservice that supports the SET Cardholder TP. A CL service may have different subservices for different encoding formats. A DL service could use subservices to represent different types of persistent storage. In all cases, the subservice implements the basic service functions for its category of service.

Each service category contains a number of basic service functions. A library developer may choose to implement some or all of the functions specified in the service interface. A module developer may also choose to extend the basic interface functionality by exposing pass through operations.

Module-to-Module interaction

Modules may make use of other OCSF service provider modules to implement their functionality. For example, a module implementing a CL may use the capabilities of a CSP module to perform the cryptographic operations of sign and verify. In that case, the CL module could package the certificate or Certificate Revocation List (CRL) fields to be signed or verified, attach to the appropriate CSP module, and call CSSM_SignData or CSSM_VerifyData to perform the operation.

A second form of module-to-module interaction is subservice collaboration. For example, a Public-Key Cryptographic Standard module may require collaborating CSP and DL subservices. Collaborating subservices are assumed to share state. A

2. If you want to provide a Cryptographic Service Provider, you need to contact IBM. For more information, see "Writing a cryptographic service provider" on page xiv .

module indicates that two or more subservices collaborate by assigning them the same subservice ID. When an application attaches one of the collaborating subservices, it will receive a handle that may be used to access any of the subservices having the same subservice ID. This mechanism may be used for collaboration across categories of services, but is not available within a single category of service.

Subservices may make use of other products or services as part of their implementation. For example, an Open Database Connectivity (ODBC) DL subservice may make use of a commercial database product such as DB2. A CL subservice may make use of a Certificate Authority (CA) service, such as the VeriSign DigitalID Center, for filling certification requests. The encapsulation of these products and services is exposed to applications in the CSSM_XX_WRAPPEDPRODUCT_INFO data structure, which is available by querying the OCSF registry.

A module developer may provide additional utility libraries for use by other module developers. Utility libraries are software components that contain functions that may be useful to several modules. For example, a utility library that performs DER encoding might be useful to several modules providing CL services. The utility library developer is responsible for making the definition, interpretation, and usage of their library available to other module developers.

Module administration components

Every module implementation shares certain administrative tasks that must be performed during module installation, attach, and detach. As part of module installation, the module developer must register information about the module's services with OCSF. This information is stored in the OCSF registry and may be queried by applications using the CSSM_GetModuleInfo function.

On attach, the module's administrative responsibilities include module registration, and module initialization.

During attach, the module registers its functions with OCSF and performs any initialization operations. The module uses CSSM_RegisterServices to register a function table with OCSF for each subservice that it supports. The function tables consist of pointers to the subservice functions supported by the module. During future function calls from the application, OCSF will use these function pointers to direct calls to the appropriate module subservice. When the module is detached, it performs any necessary cleanup actions.

Installing a service provider module

Every module must include functions for module initialization and cleanup. The first time the module is attached, OCSF calls the module's Initialize function to allow the module to perform any necessary initialization operations. The last time the module is detached, OCSF calls the module's function that allows the module to perform any necessary cleanup actions. OCSF will call the module's EventNotify function as part of every attach and detach operation.

Before an application can use a module, the module's name, location, and description must be registered with OCSF by an installation application. The name given to a module includes both a logical name and a Globally Unique ID (GUID). The logical name is a string chosen by the module developer to describe the module. The GUID is a structure used to differentiate between service provider

modules in the OCSF registry. GUIDs are discussed in more detail later in this section. The location of the module is required at installation time so the OCSF can locate the module and its credentials when an application requests an attach. The module description indicates to OCSF the security services available within this module.

Each module must have a GUID that the OCSF, applications, and the module itself use to uniquely identify a given module. The GUID is used by the OCSF registry to expose service provider module availability and capabilities to applications. A module uses its GUID to identify itself when it sets an error. When attaching the library, the application uses the GUID to identify the requested module.

A GUID is defined in the following example. GUID generators are publicly available for Windows 95, Windows NT, on many UNIX-based platforms and the UUIDGEN of the DCE on z/OS.

```
typedef struct cssm_guid {
    uint32 Data1;
    uint16 Data2;
    uint16 Data3;
    uint8 Data4[8];
} CSSM_GUID, *CSSM_GUID_PTR;
```

At install time, the installation program must inform OCSF of the ways in which this module can be used. The module usage information includes indicators of the overall module capabilities and descriptions of the security services available from this module. The overall module capabilities include indicators such as the module's threading properties or exportability. The security service descriptions include information on each service, its subservices, and any embedded products or services. For example, a module description might indicate that this is an exportable module containing a DL service and a CSP service, where the CSP service provides one subservice to access a software token and a second subservice to access a hardware token. The module description is made available to applications via queries to the OCSF registry.

Attaching a service provider module

Before an application can use the functions of a specific module subservice, it must use the CSSM_ModuleAttach function to request that OCSF attach to the module's subservice. On the first attach, OCSF verifies the integrity of the service provider module prior to loading the module. Loading the module initiates a call to an operating system (OS-specific) entry point in the module. On registration, the service provider module registers its tables of service function pointers with OCSF and receives the application's memory management upcalls. OCSF then uses the module function table to call the module's Initialize function to confirm version compatibility and calls the module's EventNotify function to indicate that an attach operation is occurring. Once these steps have successfully completed, OCSF returns a module handle to the application that uniquely identifies the pairing of the application thread to the module subservice instance. The application uses this handle to identify the module subservice in future function calls. The module subservice uses the handle to identify the calling application. OCSF notifies the module of subsequent attach requests from the application by using the module's EventNotify function. Subsequent attach operations do not require integrity verification.

Module entry point

When OCSF first attaches to or last detaches from a module, it initiates an OS-specific entry point. The entry points are _init and _fini. On attach, this

function is responsible for calling CSSM_RegisterServices. On detach, it is responsible for calling CSSM_DeregisterServices. To avoid OS-related conflicts, any setup or cleanup operations should be performed in the module's Initialize and Terminate functions.

Module function table registration

On attach, a module must register its function tables with OCSF by calling CSSM_RegisterServices. Its function tables consist of a table of module management function pointers, plus one table of Service Provider Interface (SPI) function pointers for each (service, subservice) pair contained in the module. The module management functions include Initialize, EventNotify, and Terminate. The interface functions reflect the OCSF API for each security service. The function prototypes and their descriptions provide the OCSF SPI specifications. If a subservice does not support a given function in its SPI, the pointer to that function must be set to NULL. These structures are specified in the OCSF header files, cssmspi.h, cssmtpi.h, cssmcli.h, and cssmdli.h.

Memory management upcalls

All memory allocation and deallocation for data passed between the application and a module via OCSF is ultimately the responsibility of the calling application. Since a module needs to allocate memory to return data to the application, the application must provide the module with a means of allocating memory that the application has the ability to free. It does this by providing the module with memory management upcalls.

Memory management upcalls are pointers to the memory management functions used by the calling application. They are provided to a module via OCSF as a structure of function pointers and are passed to the module when it calls the CSSM_RegisterServices function. The functions will be the calling application's equivalent of malloc, free, calloc, and re-alloc, and will be expected to have the same behavior as those functions. The function parameters will consist of the normal parameters for that function. The function return values should be interpreted in the standard manner. A module is responsible for making the memory management functions available to all of its internal functions.

Error handling

When an error occurs inside a module, the function should call CSSM_SetError. The CSSM_SetError function takes the module's GUID and an error number as inputs. The module's GUID is used to identify where the error occurred. The error number is used to describe the error.

The error number set by a module subservice should fall into one of two ranges. The first range of error numbers is predefined by OCSF. These are errors that are common to all modules implementing a given subservice function. They are defined in the header file, cssmerr.h, which is distributed as part of OCSF. The second range of error numbers is used to define module-specific error codes. These module-specific error codes should be in the range of CSSM_XX_PRIVATE_ERROR to CSSM_xx_END_ERROR, where XX stands for the service abbreviation (CSP, TP, CL, DL). CSSM_XX_PRIVATE_ERROR and CSSM_XX_END_ERROR are also defined in the header file cssmerr.h. A module developer is responsible for making the definition and interpretation of their module-specific error codes available to applications.

When no error has occurred, but the appropriate return value from a function is CSSM_FALSE, that function should call CSSM_ClearError before returning. When the application receives a CSSM_FALSE return value, it is responsible for checking whether an error has occurred by calling CSSM_GetError. If the module function has called CSSM_ClearError, the calling application receives a CSSM_OK response from the CSSM_GetError function, indicating no error has occurred.

Install example

An installation program is responsible for registering a module's capabilities with OCSF. A sample code segment for the installation of a CL Module is shown in the following example.

CL module install

```
#include "cssm.h"
CSSM_GUID clm_guid =
{ 0x5fc43dc1, 0x732, 0x11d0, { 0xbb, 0x14, 0x0, 0xaa, 0x0, 0x36, 0x67, 0x2d } };
CSSM_BOOL CLModuleInstall()
{
    CSSM_VERSION        cssm_version = { CSSM_MAJOR, CSSM_MINOR };
    CSSM_VERSION        cl_version = { CLM_MAJOR_VER, CLM_MINOR_VER };
    CSSM_GUID           cl_guid = clm_guid;
    CSSM_CLSUBSERVICE   sub_service;
    CSSM_SERVICE_INFO   service_info;
    CSSM_MODULE_INFO    module_info;
    char                SysDir[_MAX_PATH];

    /* fill subservice information */
    sub_service.SubServiceId = 0;
    strcpy(sub_service.Description, "X509v3 SubService");
    sub_service.CertType = CSSM_CERT_X_509v3;
    sub_service.CertEncoding = CSSM_CERT_ENCODING_DER;
    sub_service.AuthenticationMechanism = CSSM_AUTHENTICATION_NONE;
    sub_service.NumberOfTemplateFields = NUMBER_X509_CERT_OIDS;
    sub_service.CertTemplates = X509_CERT_OIDS_ARRAY;
    sub_service.NumberOfTranslationTypes = 0;
    sub_service.CertTranslationTypes = NULL;
    sub_service.WrappedProduct.EmbeddedEncoderProducts = NULL;
    sub_service.WrappedProduct.NumberOfEncoderProducts = 0;
    sub_service.WrappedProduct.AccessibleCAProducts = NULL;
    sub_service.WrappedProduct.NumberOfCAProducts = 0;

    /* fill service information */
    strcpy(service_info.Description, "CL Service");
    service_info.Type = CSSM_SERVICE_CL;
    service_info.Flags = 0;
    service_info.NumberOfSubServices = 1;
    service_info.ClSubServiceList = &sub_service;
    service_info.Reserved = NULL;

    /* fill module information */
    module_info.Version = cl_version;
    module_info.CompatibleCSSMVersion = cssm_version;
    strcpy(module_info.Description, "Vendor Module");
    strcpy(module_info.Vendor, "Vendor Name");
    module_info.Flags = 0;
    module_info.ServiceMask = CSSM_SERVICE_CL;
    module_info.NumberOfServices = 1;
    module_info.ServiceList = &service_info;
    module_info.Reserved = NULL;

    cssm.init
    /* set dir path for service provider */
    SysDir = "/usr/lpp/ocsf/my_addin";

    /* Install the module */
    if (CSSM_ModuleInstall(clm_fullname_string,
                           clm_filename_string,
                           SysDir,
                           &clm_guid,
                           &module_info,
                           NULL,
                           NULL) == CSSM_FAIL)
    {
```

```
        return CSSM_FALSE;
    }

return CSSM_TRUE;
}
```

Attach/Detach example

A module is responsible for performing certain operations when OCSF attaches to and detaches from it. Modules use **_init** in conjunction with the DLLMain routine to perform those operations, as shown in the following DL Module example.

```
_init   BOOL_init( )
   {
     BOOL rc;
     rc =  DllMain(NULL, DLL_PROCESS_ATTACH, NULL);
     return (rc);
   }
```

DLLMain

```
#include<cssm.h>
CSSM_GUID dl_guid =
{ 0x5fc43dc1, 0x732, 0x11d0, { 0xbb, 0x14, 0x0, 0xaa, 0x0, 0x36, 0x67, 0x2d } };
CSSM_SPI_DL_FUNCS FunctionTable;
CSSM_REGISTRATION_INFO  DLRegInfo;
CSSM_MODULE_FUNCS       Services;
CSSM_SPI_MEMORY_FUNCS   DLMemoryFunctions;

BOOL  DllMain ( HANDLE hInstance, DWORD dwReason, LPVOID lpReserved)
{
switch (dwReason)
{
case DLL_PROCESS_ATTACH:
{

/* Fill in Registration information */
          DLRegInfo.Initialize            = DL_Initialize;
          DLRegInfo.Terminate             = DL_Uninitialize;
          DLRegInfo.EventNotify           = DL_EventNotify;
          DLRegInfo.GetModuleInfo         = NULL;
          DLRegInfo.FreeModuleInfo        = NULL;
          DLRegInfo.ThreadSafe            = CSSM_TRUE;
          DLRegInfo.ServiceSummary        = CSSM_SERVICE_DL;
          DLRegInfo.NumberOfServiceTables = 1;
          DLRegInfo.Services              = &Services;

          /* Fill in Services */

          Services.ServiceType  = CSSM_SERVICE_DL;
          Services.DlFuncs = &FunctionTable;

/* Fill in FunctionTable with function pointers */
FunctionTable.Authenticate = DL_Authenticate;
FunctionTable.DbOpen = DL_DbOpen;
FunctionTable.DbClose = DL_DbClose;
FunctionTable.DbCreate = DL_DbCreate;
FunctionTable.DbDelete = DL_DbDelete;
FunctionTable.DbImport = DL_DbImport;
FunctionTable.DbExport = DL_DbExport;
FunctionTable.DbSetRecordParsingFunctions = DL_DbSetRecordParsingFunctions;
FunctionTable.DbGetRecordParsingFunctions = DL_DbGetRecordParsingFunctions;
   FunctionTable.GetDbNameFromHandle = DL_GetDbNameFromHandle;
   FunctionTable.DataInsert   = DL_DataInsert;
   FunctionTable.DataDelete   = DL_DataDelete;
FunctionTable.DataGetFirst = DL_DataGetFirst;
FunctionTable.DataGetNext = DL_DataGetNext;
FunctionTable.DataAbortQuery = DL_DataAbortQuery;
FunctionTable.FreeUniqueRecord = DL_FreeUniqueRecord;
FunctionTable.PassThrough = DL_PassThrough;

/* Call CSSM_RegisterServices to register the FunctionTable */
/* with CSSM and to receive the application's memory upcall table */
   if (CSSM_ RegisterServices (&dl_guid, &DLRegInfo,
&DLMemoryFunctions,NULL) != CSSM_OK)
return FALSE;

/* Make the upcall table available to all functions in this library */
```

```
break;
}
case DLL_THREAD_ATTACH:
break;
case DLL_THREAD_DETACH:
break;
case DLL_PROCESS_DETACH:
if (CSSM_DeregisterServices (&dl_guid) != CSSM_OK)
return FALSE;
break;
}
return TRUE;
}
```

Service provider module interface functions

These interfaces are used by OCSF service providers to register information with and to provide address of supported function to the OCSF.

Data structures

This section describes the data structures that may be passed to or returned from a service provider module function. They are used by modules to prepare data passing to and from the calling application through the OCSF Framework. These data structures are defined in the header file, cssmspi.h, which is distributed with the OCSF . Data structures that are specific to a particular type of service provider module, such as a Trust Policy (TP) Service Provider or Data Library service provider, are described in the individual OCSF service provider sections of this book.

The data structures used in OCSF are described in the /usr/lpp/ocsf/include/cssmtype.h header. Many of these data structures are compatible with the equivalent cssmtype.h headers on other OCSF platforms. The exceptions are those enclosed in "#ifdef_MVS".

Basic data types

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

The following is used by OCSF data structures to represent a character string inside of a fixed-length buffer. The character string is expected to be NULL-terminated. The string size was chosen to accommodate current security standards.

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

CSSM_ALL_SUBSERVICES

This data type is used to identify that information on all of the subservices is being requested or returned.

```
#define CSSM_ALL_SUBSERVICES (-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.

CSSM_CALLBACK

An application uses this data type to request that a service provider module call back into the application for certain cryptographic information.

```
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.

CSSM_CRYPTO_DATA

This data structure is used to encapsulate cryptographic information, such as the passphrase to use when accessing a private key.

```
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 &tab;&tab;parameter.
CallbackID
A tag that identifies the callback.

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 OCSF. Trust Policy (TP) modules and Certificate Libraries (CLs) use this structure to hold certificates and Certificate Revocation Lists (CRLs). Other service provider modules, such as Cryptographic Service Providers (CSPs), use this same structure to hold general data buffers. Data Storage Library (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

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, the RPC UUIDGEN/uuid_gen program can be used on a number of UNIX-based platforms and the UUIDGEN of the DCE on z/OS can be used to generate a GUID.)

```
typedef struct cssm_guid{
    uint32 Data1;
    uint16 Data2;
    uint16 Data3;
    uint8  Data4[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.

CSSM_HANDLE

A unique identifier for an object managed by OCSF or by a service provider module.

```
typedef uint32 CSSM_HANDLE, *CSSM_HANDLE_PTR
```

CSSM_HANDLEINFO

This structure is used by service provider modules to obtain information about a CSSM_HANDLE.

```
typedef struct cssm_handleinfo {
    uint32 SubServiceID;
    uint32 SessionFlags;
    CSSM_NOTIFY_CALLBACK Callback;
    uint32 ApplicationContext;
} CSSM_HANDLEINFO, *CSSM_HANDLEINFO_PTR;
```

Definitions:

SubserviceID
: An identifier for this subservice.

SessionFlags
: A bit-mask of service options defined by a particular subservice of the module. &tab;Legal values are described in the module-specific documentation. A default set of flags is specified in the CSSM_MODULE_INFO structure for use by the caller.

Callback
: A callback function registered by the application as part of the module attach &tab;operation. This function should be used to notify the application of certain events.

ApplicationContext
: An identifier which should be passed back to the application as part of the Callback function.

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 OCSF service types. Each service may provide one or more subservices. Modules also can have dynamically available services and features.

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

CSSM_MEMORY_FUNCS/ CSSM_API_MEMORY_FUNCS

This structure is used by applications to supply memory functions for the OCSF and the service provider modules. The functions are used when memory needs to be allocated by the OCSF or service providers 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 a function that returns a void pointer to the allocated memory block of at least *Size* bytes from heap *AllocRef*.

Free_func
: Pointer to a function that deallocates a previously allocated memory block (*MemPtr*) from heap *AllocRef*.

Realloc_func
: Pointer to a function that returns a void pointer to the reallocated memory block (*MemPtr*) of at least *Size* bytes from heap *AllocRef*.

Calloc_func
: Pointer to a function that returns a void pointer to an array of *Num* elements of length *Size* initialized to zero from heap *AllocRef*.

AllocRef
: Indicates which memory heap the function operates on

CSSM_MODULE_FLAGS

This bit-mask is used to identify characteristics of the module, such as whether or not it is threadsafe.

```
typedef uint32 CSSM_MODULE_FLAGS;

#define CSSM_MODULE_THREADSAFE  0x1  /* Module is threadsafe */
#define CSSM_MODULE_EXPORTABLE  0x2  /* Module can be exported outside the USA */
```

CSSM_MODULE_FUNCS

This structure is used by service provider modules to pass a table of function pointers for a single service to OCSF.

```
typedef struct cssm_module_funcs {
    CSSM_SERVICE_TYPE ServiceType;
    union {
        void *ServiceFuncs;
        CSSM_SPI_CSP_FUNCS_PTR CspFuncs;
        CSSM_SPI_DL_FUNCS_PTR  DlFuncs;
        CSSM_SPI_CL_FUNCS_PTR  ClFuncs;
        CSSM_SPI_TP_FUNCS_PTR  TpFuncs;
        CSSM_SPI_KRSP_FUNCS_PTR KrspFuncs;
        };
} CSSM_MODULE_FUNCS, *CSSM_MODULE_FUNCS_PTR;
```

Definitions:

ServiceType

The type of service provider module services accessible via the *XX*Funcs function table.

XXFuncs

A pointer to a function table of the type described by ServiceType. These function pointers are used by OCSF to direct function calls from an application to the appropriate service in the service provider module. These function pointer tables are described in the OCSF header files cssmcspi.h, cssmkrspi.h, &tab;cssmdli.h, cssmcli.h, and cssmtpi.h. Table 1 provides the service access tables.

Table 1. Service Access Tables

Value	Description
CSSM_SPI_CSP_FUNCS_PTR CspFuncs	Function pointers to CSP services
CSSM_SPI_KRSP_FUNCS_PTR KrspFuncs	Function pointers to KR services **Note:** This is not supported in z/OS.
CSSM_SPI_DL_FUNCS_PTR DlFuncs	Function pointers to DL services
CSSM_SPI_CL_FUNCS_PTR ClFuncs	Function pointers to CL services
CSSM_SPI_TP_FUNCS_PTR TpFuncs	Function pointers to TP services

CSSM_MODULE_HANDLE

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

```
typedef uint32 CSSM_MODULE_HANDLE
```

CSSM_MODULE_INFO

This structure aggregates all service descriptions about all service types of a module implementation.

```
typedef struct cssm_module_info {
   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;
```

Definitions:

Version
The major and minor version numbers of this service provider module.

CompatibleCSSMVersion
The version of OCSF to which this module was written.

Description
A text description of this module and its functionality.

Vendor The name and description of the module vendor.

Flags Characteristics of this module, such as whether or not it is threadsafe.

ServiceMask
A bit-mask identifying the types of services available in this module.

NumberOfServices
The number of services for which information is provided. Multiple descriptions (as subservices) can be provided for a single service category.

ServiceList
An array of pointers to the service information structures. This array contains *NumberOfServices* entries.

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

CSM_NOTIFY_CALLBACK

The CSSM_NOTIFY_CALLBACK is used by the application to provide a function pointer to a callback routine. It is typically supplied in the CSSM_ModuleAttach API when the application developer wishes something to be called in response to a particular event happening. It is defined as follows:

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

Definitions:
ModuleHandle
The handle of the attached service provider module.
Application
Input data to identify the callback.
Reason The reason for the notification (see Table 2).
Param Any additional information about the event.

Table 2. Notification Reasons

Reason	Description
CSSM_NOTIFY_SURRENDER	The service provider module is temporarily surrendering control of the process.
CSSM_NOTIFY_COMPLETE	An asynchronous operation has completed.
CSSM_NOTIFY_DEVICE_REMOVED	A device, such as a token, has been removed.
CSSM_NOTIFY_DEVICE_INSERTED	A device, such as a token, has been inserted.

CSSM_REGISTRATION_INFO

This structure is used by service provider modules to pass tables of function pointers and module information to OCSF.

```
typedef struct cssm_registration_info {
    /* Loading, Unloading and Event Notifications */
    CSSM_RETURN (CSSMAPI *Initialize) (CSSM_MODULE_HANDLE Handle,
                                       uint32 VerMajor,
                                       uint32 VerMinor);
    CSSM_RETURN (CSSMAPI *Terminate)  (CSSM_MODULE_HANDLE Handle);
    CSSM_RETURN (CSSMAPI *EventNotify)(CSSM_MODULE_HANDLE Handle,
                                        const CSSM_EVENT_TYPE Event,
                                        const uint32 Param);
    CSSM_MODULE_INFO_PTR (CSSMAPI *GetModuleInfo)
                                      (CSSM_MODULE_HANDLE ModuleHandle,
                                       CSSM_SERVICE_MASK ServiceMask,
                                       uint32 SubserviceID,
                                       CSSM_INFO_LEVEL InfoLevel);
    CSSM_RETURN (CSSMAPI *FreeModuleInfo) (CSSM_MODULE_HANDLE ModuleHandle,
                                           CSSM_MODULE_INFO_PTR ModuleInfo);
    CSSM_BOOL ThreadSafe;
    uint32 ServiceSummary;
    uint32 NumberOfServiceTables;
    CSSM_MODULE_FUNCS_PTR Services;
} CSSM_REGISTRATION_INFO, *CSSM_REGISTRATION_INFO_PTR;
```

Definitions:

Initialize
: Pointer to function that verifies compatibility of the requested module version with the actual module version, and which performs module setup operations.

Terminate
: Pointer to function that performs module cleanup operations.

EventNotify
: Pointer to function that accepts event notification from OCSF.

GetModuleInfo
: Pointer to function that obtains and returns dynamic information about the module.

FreeModuleInfo
: Pointer to function that frees the module information structure.

Threadsafe
: &tab;A flag that indicates to OCSF whether or not the module is capable of handling multithreaded access.

ServiceSummary
: A bit-mask indicating the types of services offered by this module. It is the bitwise-OR of the service types described in Table 1.

NumberofServiceTables
: The number of distinct services provided by this module. This is also the length of the *Services* array.

Services
: An array of CSSM_MODULE_FUNCS structures that provide the mechanism for accessing the module's services.

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.

CSSM_SERVICE_FLAGS

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

```
typedef uint32 CSSM_SERVICE_FLAGS

#define CSSM_SERVICE_ISWRAPPEDPRODUCT 0x1
                                /* On  = Contains one or more embedded products
                                   Off = Contains no embedded products */
```

CSSM_SERVICE_INFO

This structure holds a description of a module service. The service described is of the OCSF service type specified by the module type.

```
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 {
        void *SubServiceList;
        CSSM_CSPSUBSERVICE_PTR CspSubServiceList;
        CSSM_DLSUBSERVICE_PTR  DlSubServiceList;
        CSSM_CLSUBSERVICE_PTR  ClSubServiceList;
        CSSM_TPSUBSERVICE_PTR  TpSubServiceList;
        CSSM_KRSUBSERVICE_PTR  KrSubServiceList;
  };
  void *Reserved;
} CSSM_SERVICE_INFO, *CSSM_SERVICE_INFO_PTR;
```

Definitions:

Description
A text description of the service.

Type Specifies exactly one type of service structure, such as CSSM_SERVICE_CSP, &tab;CSSM_SERVICE_CL, etc.

Flags Characteristics of this service, such as whether it contains any embedded products.

NumberOfSubServices
The number of elements in the module *SubServiceList*.

SubServiceList
A list of descriptions of the encapsulated subservices (not of the basic service types).

CspSubServiceList
A list of descriptions of the encapsulated CSP subservices.

DlSubServiceList
A list of descriptions of the encapsulated DL subservices.

ClSubServiceList
A list of descriptions of the encapsulated CL subservices.

TpSubServiceList
A list of descriptions of the encapsulated TP subservices.

KrSubServiceList[3]

A list of descriptions of the encapsulated key recovery subservices.

Reserved

This field is reserved for future use. It should always be set to NULL.

CSSM_SERVICE_MASK

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

```
typedef uint32 CSSM_SERVICE_MASK;

#define CSSM_SERVICE_CSSM  0x1
#define CSSM_SERVICE_CSP   0x2
#define CSSM_SERVICE_DL    0x4
#define CSSM_SERVICE_CL    0x8
#define CSSM_SERVICE_TP    0x10
#define CSSM_SERVICE_KR    0x20
#define CSSM_SERVICE_LAST  CSSM_SERVICE_TP
```

CSSM_SERVICE_TYPE

This data type is used to identify a single service from the CSSM_SERVICE_MASK options defined above.

```
typedef CSSM_SERVICE_MASK CSSM_SERVICE_TYPE
```

CSSM_SPI_FUNC_TBL

This structure is used by service provider modules to reference an application's memory management functions. The functions are used when a service provider module needs to allocate memory for returning data structures to the application, or needs to deallocate memory for a data structure that is passed to it from an application.

```
typedef struct cssm_spi_func_tbl {
    void *(*malloc_func) (CSSM_HANDLE AddInHandle, uint32 Size);
    void  (*free_func)   (CSSM_HANDLE AddInHandle, void *MemPtr);
    void *(*realloc_func)(CSSM_HANDLE AddInHandle, void *MemPtr, uint32 Size);
    void *(*calloc_func) (CSSM_HANDLE AddInHandle, uint32 Num, uint32 Size);
} CSSM_SPI_MEMORY_FUNCS, *CSSM_SPI_MEMORY_FUNCS_PTR;
```

Definitions:

Malloc_func

Pointer to a function that returns a void pointer to the allocated memory block of at least *Size* bytes from the heap of the application associated with *AddInHandle*.

Free_func

Pointer to a function that deallocates a previously allocated memory block (*MemPtr*) from the heap of the application associated with *AddInHandle*.

Realloc_func

Pointer to a function that returns a void pointer to the reallocated memory block (*MemPtr*) of at least *Size* bytes from the heap of the application associated with *AddInHandle*.

Calloc_func

Pointer to function that returns a void pointer to an array of *Num* elements of length *Size* initialized to zero from the heap of the application associated with *AddInHandle*.

3. This is not supported in z/OS.

CSSM_USER_AUTHENTICATION

This structure holds the user's credentials for authentication 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 magic token, or whatever is required by a service provider module 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.

CSSM_USER_AUTHENTICATION_ MECHANISM

The enumerated list of CSSM_User_Authentication_Mechanism 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. OCSF-defined authentication methods include password-based authentication, a login sequence, or a certificate and passphrase. It is anticipated that new mechanisms will be added to this list as required.

```
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;
```

CSSM_VERSION

This structure is used to represent the version of OCSF 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.

Relevant CSSM API functions

Several API functions are particularly relevant to module developers because they are used either by the application to access a module, or by a module to access OCSF services such as the OCSF registry or the error-handling routines. For additional information, module developers are encouraged to reference the *z/OS Open Cryptographic Services Facility Application Programming* book.

Service provider module functions

A service provider module interfaces with OCSF using the functions described in this section.

CSSM_DeregisterServices

Purpose
This function is used by a service provider module to deregister its function table with OCSF

Format
```
CSSM_RETURN CSSMAPI CSSM_DeregisterServices (const CSSM_GUID_PTR GUID)
```

Parameters
Input

GUID A pointer to the CSSM_GUID structure containing the Globablly Unique ID (GUID) for this module.

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

Error codes

Value	Description
CSSM_INVALID_GUID	Invalid GUID
CSSM_DEREGISTER_SERVICES_FAIL	Unable to deregiser services.

Related information
CSSM_RegisterServices

CSSM_GetHandleInfo

Purpose
This function retrieves a CSSM_HANDLEINFO structure which describes the attributes of the service provider module referenced by *hModule*.

Format
```
CSSM_HANDLEINFO_PTR CSSMAPI CSSM_GetHandleInfo (CSSM_HANDLE hModule)
```

Parameters
Input

hModule
Handle of the service provider module.

Return value
A pointer to a CSSM_HANDLEINFO data structure. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

CSSM_ModuleInstall

Purpose
This function registers the module with OCSF. OCSF adds the module's descriptive information to its persistent registry. This makes the service module available for use on the local system. The function accepts as input the name and unique identifier for the module, the location executable code for the module, and a

digitally signed list of capabilities supported by the module. The module name and description are added to the OCSF registry, making the module available for use by applications.

Format

```
CSSM_RETURN CSSMAPI CSSM_ModuleInstall (const char *ModuleName,
                                        const char *ModuleFileName,
                                        const char *ModulePathName,
                                        const CSSM_GUID_PTR GUID,
                                        const CSSM_MODULE_INFO_PTR ModuleDescription,
                                        const void * Reserved1,
                                        const CSSM_DATA_PTR Reserved2)
```

Parameters

Input

ModuleName
The name of the module.

ModuleFileName
The name of the file that implements the module.

ModulePathName
The path to the file that implements the module.

GUID A pointer to the CSSM_GUID structure containing the GUID for the module.

ModuleDescription
A pointer to the CSSM_MODULE_INFO structure containing a description of the module.

Reserved1
Reserve data for the function.

Reserved2
Reserve data for the function.

Return value

A CSSM_OK return value signifies that information has been updated. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information

CSSM_ModuleUninstall

CSSM_ModuleUninstall

Purpose

This function deletes the persistent OCSF internal information about the module and removes it from the name space of available modules in the OCSF system.

Format

```
CSSM_RETURN CSSMAPI CSSM_ModuleUninstall (const CSSM_GUID_PTR GUID)
```

Parameters

Input

GUID A pointer to the CSSM_GUID structure containing the GUID for the module.

Return value
A CSSM_OK return value means the module has been successfully uninstalled. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
CSSM_ModuleInstall

CSSM_RegisterServices

Purpose
This function is used by a service provider module to register its function table with OCSF and to receive a memory management upcall table from OCSF.

Format
```
CSSM_RETURN CSSMAPI CSSM_RegisterServices (const CSSM_GUID_PTR GUID,
                                           const CSSM_REGISTRATION_INFO_PTR FunctionTable,
                                           CSSM_SPI_MEMORY_FUNCS_PTR UpcallTable,
                                           void *Reserved)
```

Parameters
Input

GUID A pointer to the CSSM_GUID structure containint the GUID for the calling module.

FunctionTable
A structure containing pointers to the interface functions implemented by this module, organized by interface type.

Reserved
A reserved input.

Output

UpcallTable
A pointer to the CSSM_SPI_MEMORY_FUNCS structure containing the memory management function pointers to be used by this module

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

Error codes

Value	Desciption
CSSM_INVALID_GUID	Invalid GUID
CSSM_INVALID_FUNCTION_TABLE	Invalid function table
CSSM_REGISTER_SERVICES_FAIL	Unable to register services

Related information
CSSM_DeregisterServices

CSSM_SetModuleInfo

Purpose

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 OCSF registry, make a private copy, and then use the *CSSM_SetModuleInfo* function to update the OCSF registry.

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

Format

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

Parameters

Input

ModuleGUID
: A pointer to the CSSM_GUID structure containing the GUID for the service provider module.

ModuleInfo
: A pointer to the complete structured set of descriptive information about the module.

Return value

A CSSM_OK return value signifies that the module information has been successfully written to the registry. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information

CSSM_GetModuleInfo
CSSM_FreeModuleInfo

EventNotify

Purpose

This function is used by OCSF to notify the module of certain events such as module attach and detach operations.

Format

```
CSSM_RETURN CSSMAPI EventNotify (CSSM_MODULE_HANDLE Handle,
                                 const CSSM_EVENT_TYPE Event,
                                 const uint32 Param)
```

Parameters

Input

Handle The handle that identifies the module to application thread pairing

Event The event that is occurring. The possible events are described in Table 3 on page 21.

Param An event-specific parameter (see Table 4 on page 21).

Table 3. Module Event Types

Event	Description
CSSM_EVENT_ATTACH	The application has requested an attach operation.
CSSM_EVENT_DETACH	The application has requested a detach operation.
CSSM_EVENT_INFOATTACH	An application has requested module info and OCSF wants to obtain the module's dynamic capabilities. The service provider module cannot assume that Initialize or Terminate has been called.
CSSM_EVENT_INFODETACH	OCSF has finished obtaining the module's dynamic capabilities.
CSSM_EVENT_CREATE_CONTEXT	A context has been created.
CSSM_EVENT_DELETE_CONTEXT	A context has been deleted.

Table 4. Module Event Parameters

Event	Parameter
CSSM_EVENT_ATTACH	None
CSSM_EVENT_DETACH	None
CSSM_EVENT_INFOATTACH	None
CSSM_EVENT_INFODETACH	None
CSSM_EVENT_CREATE_CONTEXT	Context handle
CSSM_EVENT_DELETE_CONTEXT	Context handle

Return value

A CSSM_OK return value signifies that the module's event-specific operations were successfully performed. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information

Initialize
Terminate

FreeModuleInfo

Purpose

This function frees the memory allocated to hold all of the info structures returned by GetModuleInfo. All substructures within the info structure are freed by this function.

Format

```
CSSM_RETURN CSSMAPI FreeModuleInfo (CSSM_MODULE_HANDLE ModuleHandle,
                                    CSSM_MODULE_INFO_PTR ModuleInfo)
```

Parameters

Input

ModuleHandle
The handle of the attached service provider module.

ModuleInfo
A pointer to the CSSM_MODULE_INFO structures to be freed

Return value
This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

Error codes

Value	Description
CSSM_INVALID_MODULEINFO_POINTER	Invalid Pointer

Related information
GetModuleInfo

GetModuleInfo

Purpose
This function returns descriptive information about the module identified by the ModuleHandle. The information returned can include all of the capability information for each subservice, and 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 service 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 the module with dynamic capabilities. *InfoLevel* can be used to request static attribute values only or dynamic values.

Format
```
CSSM_MODULE_INFO_PTR CSSMAPI GetModuleInfo (CSSM_MODULE_HANDLE ModuleHandle,
                                            CSSM_SERVICE_MASK ServiceMask,
                                            uint32 SubserviceID,
                                            CSSM_INFO_LEVEL InfoLevel)
```

Parameters
Input

ModuleHandle
The handle of the attached service provider module.

ServiceMask
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
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 the operation may already be limited by a service mask. If so, the subservice ID applies to all service categories &tab;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
Indicates the level of detail returned by this function. Information retrieval can be restricted as follows:

- 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 OCSF to interrogate the module's current capability status.

Return value

A pointer to a module info structure containing a pointer to an array of zero or more service information structures. Each structure contains type information identifying the service description as representing Certificate Library services (CL), Data Storage Library (DL) services, etc. The service descriptions are subclassed into subservice descriptions that describe the attributes and capabilities of a subservice.

Error codes

Value	Description
CSSM_INVALID_POINTER	Invalid pointer
CSSM_INVALID_USAGE_MASK	Invalid bit-mask
CSSM_INVALID_SUBSERVICEID	Invalid subservice ID
CSSM_INVALID_INFO_LEVEL	Invalid info level indicator
CSSM_MEMORY_ERROR	Internal memory error
CSSM_INVALID_GUID	Unknown GUID

Related information

CSSM_SetModuleInfo CSSM_FreeModuleInfo

Initialize

Purpose

This function checks whether the current version of the module is compatible with the input version, and performs any module-specific setup activities

Format

```
CSSM_RETURN CSSMAPI Initialize (CSSM_MODULE_HANDLE Handle, uint32 VerMajor, uint32 VerMinor)
```

Parameters

Input

Handle The handle that identifies the module to application thread pairing

VerMajor
The major version number of the module expected by the calling application.

VerMinor
The minor version number of the module expected by the calling application.

Return value
A CSSM_OK return value signifies that the current version of the module is compatible with the input version numbers, and all setup operations were successfully performed. When CSSM_FAIL is returned, either the current module is incompatible with the requested module version or an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
Terminate
EventNotify

Terminate

Purpose
This function performs any module-specific cleanup activities.

Format
```
CSSM_RETURN CSSMAPI Terminate (CSSM_MODULE_HANDLE Handle)
```

Parameters
Input

Handle The handle that identifies the module to application thread pairing.

Return value
A CSSM_OK return value signifies that all cleanup operations were successfully performed. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
Initialize
EventNotify

Chapter 2. Cryptographic service provider module information

Cryptographic Service Providers (CSPs) are service provider modules which perform cryptographic operations including encryption, decryption, digital signing, key pair generation, random number generation, message digest, and key exchange. Besides the traditional cryptographic functions, CSPs may provide other vendor-specific services. For more information on providing your own CSP, you need to contact IBM by using one of the following methods:

- Send an e-mail note to S390PID@US.IBM.COM
- Call the Solution Developer Program Hotline at 1-770-835-9902 (worldwide) or 1-800-627-8363 (US and Canada), ask for the S/390 Administrator
- Access the S/390 Partners in Development home page at:

  `http://www.s390.ibm.com/s390da`

  and use the feedback form to make a request.

Chapter 3. Trust policy interface

A digital certificate is the binding of some identification to a public key in a particular domain. When a trust domain authority issues (creates and signs) a certificate to a subject, it binds the subject's public key to the identity. This binding obviously can be verified through the signature verification process. The issuing authority also associates a level of trust with the certificate. The actions of the user, whose identity is bound to the certificate, are constrained by the Trust Policy (TP) governing the usage domain of the certificate. A digital certificate is a subject's credential in cyberspace that cannot be forged.

The use of digital certificates is the basic premise of OCSF design. The OCSF assumes the concept of digital certificates in its broadest sense. Applications use digital certificates as credential for:

- Identification
- Authentication
- Authorization.

The applications interpret and manipulate the contents of certificates to achieve these ends based on the real-world trust model they chose as their model for trust and security. The primary purpose of a TP module is to answer the question, "Is this certificate trusted for this action"? The OCSF TP application programming interface (API) determines the generic operations that should be defined for certificate-based trust in every application domain. The specific semantics of each operation is defined by the following:

- Application domain
- Trust model
- Policy statement for a domain
- Certificate type
- Real-world operation the user is trying to perform within the application domain.

The trust model is expressed as an executable policy that is used by all applications that ascribe to that policy and the trust model it represents. As an infrastructure, OCSF is policy neutral; it does not incorporate any single policy. For example, the verification procedure for a credit card certificate should be defined and implemented by the credit company issuing the certificate. Employee access to a lab housing a critical project should be defined by the company whose intellectual property is at risk. Rather than defining policies, OCSF provides the infrastructure for installing and managing policy-specific modules. This ensures complete extensibility of certificate-based trust on every platform hosting OCSF.

Different TPs define different actions that an application may request. Some of these actions are common to every TP, and are operations on objects that all trust models use. The objects common to all trust models are certificates and Certificate Revocation Lists (CRLs). The basic operations on these objects are sign, verify, and revoke.

OCSF defines a set of API calls that should be implemented by TP modules. These calls allow an application to perform basic operations such as verify, sign-on certificates, and CRLs. More extensible operations can be embedded in the implementation of these APIs.

Application developers and trust domain authorities benefit from the ability to define and implement policy-based modules. Application developers are freed from the burden of implementing a policy description and certifying that their implementation conforms. Instead, the application needs only to build in a list of the authorities and certificate issuers it uses.

Trust domain authorities also benefit from an infrastructure that supports TP modules. Trust domain authorities are ensured that applications using their modules adhere to the policies of the domain. Individual functions within the module may combine local and remote processing. This flexibility allows the module developer to implement policies based on the ability to communicate with a remote authority system. This also allows the policy implementation to be decomposed in any convenient distributed manner.

Implementing a TP module may or may not be tightly coupled with one or more Certificate Library (CL) modules or one or more Data Storage Library (DL) modules. The TP embodies the semantics of the domain. The CL and the DL embody the syntax of a certificate format and operations on that format. A TP can be completely independent of certificate format, or it may be defined to operate with one or a small number of certificate formats. A TP implementation may invoke a CL module and/or a DL module to manipulate certificates.

Trust policy services API

OCSF defines eight API calls that TP modules can implement. These calls implement various categories of operations that can be performed on trust objects.

Signing Certificates and Certificate Revocation Lists. Every system should be capable of being a Certificate Authority (CA), if so authorized. CAs are applications that issue and validate certificates and CRLs. Issuing certificates and CRLs include initializing their attributes and digitally signing the result using the private key of the issuing authority. The private key used for signing is associated with the signer's certificate. The TP module must evaluate the trustworthiness of the signer's certificate before performing this operation. Some policies may require that multiple authorities sign an issued certificate. If the TP trusts the signer's certificate, then the TP module may perform the cryptographic signing algorithm by invoking the signing function in a CL module, or by directly invoking the data signing function in a Cryptographic Service Provider (CSP) module. The CL functions that can be used to carry out some of the TP operations are documented in this book.

Verifying Certificates and Certificate Revocation Lists. The TP module determines the trustworthiness of a CRL received from a remote system. The test focuses on the trustworthiness of the agent who signed the CRL. The TP module may need to perform operations on the certificate or CRL to determine trustworthiness. If these operations depend on the data format of the certificate or CRL, the TP module uses the services of a CL module to perform these checks.

Revoking Certificates. When revoking a certificate, the identity of the revoking agent is presented in the form of another certificate. The TP module must determine trustworthiness of the revoking agent's certificate to perform revocation. If the requesting agent's certificate is trustworthy, the TP module carries out the operation directly by invoking a CL module to add a new revocation record to a CRL, marking the certificate as revoked. The OCSF API also defines a reason parameter that is passed to the TP module. The TP may use this parameter as part of its trust evaluation.

PassThrough Function. For operations not defined in the TPI, the passthrough function allows the TP module to provide support for these services to clients. These private services are identified by operation identifiers. TP module developers must provide documentation of these services

Trust policy data structures

This section describes the data structures that may be passed to or returned from a TP function. They will be used by applications to prepare data to be passed as input parameters into OCSF API function calls that will be passed without modification to the appropriate TP. The TP is then responsible for interpreting them and returning the appropriate data structure to the calling application through OCSF. These data structures are defined in the header file, cssmtype.h, which is distributed with OCSF.

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];
```

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.

CSSM_CERTGROUP

This structure contains a set of certificates. It is assumed that the certificates are related based on the signature hierarchy. A typical group is a chain of certificates. The certificate group is a syntactic representation of a trust model. All certificates in the group must be of the same type and issued for the same trust domain.

```
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.

CSSM_DATA

The CSSM_DATA structure associates 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 OCSF.

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

Definitions:

Length The length, in bytes, of the memory block pointed to by *Data*.

Data A pointer to a contiguous block of memory.

CSSM_DL_DB_HANDLE

This data structure holds a pair of handles, one for a DL and another for a data store 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.

CSSM_DL_DB_LIST

This data structure defines a list of handle pairs (DL handle, data store handle).

```
typedef struct cssm_dl_db_list {
    uint32 NumHandles;
    CSSM_DL_DB_HANDLE_PTR DLDBHandle;
} CSSM_DL_DB_LIST, *CSSM_DL_DB_LIST_PTR;
```

Definitions:

NumHandles

Number of pairs in the list (DL handle, data store handle).

DLDBHandle

List of pairs (DL handle, data store handle).

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 &tab;contiguous block of memory.

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
```

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.

CSSM_REVOKE_REASON

This 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;
```

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
```

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 */
```

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,&tab;/* use the pre-defined stopping criteria */
    CSSM_TP_STOP_ON_NONE = 1,&tab;/* 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;
```

Trust policy operations

This section describes the function prototypes expected for the functions in the TPI. The functions will be exposed to OCSF through a function table, so the function names may vary at the discretion of the TP developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications.

TP_CertSign

Purpose

The TP module decides first whether the signer certificate is trusted to sign the subject certificate. Once the trust is established, the TP signs the certificate when given the signer's certificate and the *scope* of the signing process.

Format

```
CSSM_DATA_PTR CSSMTPI TP_CertSign (CSSM_TP_HANDLE TPHandle,
                                   CSSM_CL_HANDLE CLHandle,
                                   CSSM_CC_HANDLE CCHandle,
                                   const CSSM_DL_DB_LIST_PTR DBList,
                                   const CSSM_DATA_PTR CertToBeSigned,
                                   const CSSM_CERTGROUP_PTR SignerCertGroup,
                                   const CSSM_FIELD_PTR SignScope,
                                   uint32 ScopeSize)
```

Parameters

Input

TPHandle
The handle that describes the TP module used to perform this function.

CLHandle
The handle that describes the CL module used to perform this function.

CCHandle
The cryptographic context specifies the handle of the CSP that must be &tab;used to perform the operation.

DBList 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 &tab;objects (such as certificate and CRLs) related to the signer's certificate or a data store for storing a resulting signed CRL.

CertToBeSigned
A pointer to the CSSM_DATA structure containing a certificate to be signed.

SignerCertGroup
A pointer to the CSSM_CERTGROUP structure containing one or more related certificates used to sign the certificate.

SignScope
A pointer to the CSSM_FIELD array containing the tags of the certificate &tab;&tab;&tab;fields to be included in the signing process.

ScopeSize
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 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.

Related information
CSSM_TP_CertVerify
CSSM_CL_CertSign

TP_CertRevoke

Purpose
The TP module determines whether the revoking certificate can revoke the subject certificate. The revoker certificate group is first authenticated and its applicability to perform this operation is determined. Once the trust is established, the TP revokes the subject certificate by adding it to the CRL. The revoker certificate and passphrase is used to sign the resultant CRL.

Format
```
CSSM_DATA_PTR CSSMTPI TP_CertRevoke
                 (CSSM_TP_HANDLE TPHandle,
                  CSSM_CL_HANDLE CLHandle,
                  CSSM_CC_HANDLE CCHandle,
                  const CSSM_DL_DB_LIST_PTR DBList,
                  const CSSM_DATA_PTR OldCrl,
                  const CSSM_CERTGROUP_PTR CertGroupToBeRevoked,
                  const CSSM_CERTGROUP_PTR RevokerCertGroup,
                  CSSM_REVOKE_REASON Reason)
```

Parameters
Input

TPHandle
The handle that describes the TP module used to perform this function.

CLHandle
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
The handle that describes the context for a cryptographic operation. The cryptographic context specifies the handle of the CSP that must be used &tab;to perform the operation

DBList A list of certificate databases containing certificates that may be used to construct the trust structure of the subject and revoker certificate group.

OldCrl A pointer to the CSSM_DATA structure containing an existing CRL. If &tab;this input is NULL, a new list is created.

CertGroupToBeRevoked
A group of one or more certificates that partially or fully represent the

&tab;certificate to be revoked by this operation. The first certificate in the group is the target certificate. The use of subsequent certificates is &tab;specific to the trust domain. For example, in a hierarchical trust model subsequent members are intermediate certificates of a certificate chain.

RevokerCertGroup
A group of one or more certificates that partially or fully represent the revoking entity for this operation. The first certificate in the group is the target certificate representing the revoker. The use of subsequent certificates is specific to the trust domain.

Reason 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.

Related information
CSSM_CL_CrlAddCert

TP_CrlVerify

Purpose
This function verifies the integrity of the CRL and determines whether it is trusted. Some of the checks that may be performed include verifying the signatures on the signer's certificate group, establishing the authorization of the signer to issue CRLs, verification of the signature on the CRL, verifying validity period of the CRL and the date the CRL was issued, etc.

Format
```
CSSM_BOOL CSSMTPI TP_CrlVerify (CSSM_TP_HANDLE TPHandle,
                                CSSM_CL_HANDLE CLHandle,
                                CSSM_CSP_HANDLE CSPHandle,
                                const CSSM_DL_DB_LIST_PTR DBList,
                                const CSSM_DATA_PTR CrlToBeVerified,
                                const CSSM_CERTGROUP_PTR SignerCertGroup,
                                const CSSM_FIELD_PTR VerifyScope,
                                uint32 ScopeSize)
```

Parameters
Input

TPHandle
The handle that describes the TP module used to perform this function.

CSPHandle
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 &tab;verification operation. If no CSP is specified, the TP module uses an&tab;assumed CSP to perform the operations.

DBList 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. &tab;If no DL and database (DB) handle pairs are specified, the TP module &tab;can use an assumed DL module and an assumed data store, if required.

CrlToBeVerified
A pointer to the CSSM_DATA structure containing a signed CRL to be verified.

SignerCertGroup
A group of one or more certificates that partially or fully represent the signer of the CRL. The first certificate in the group is the target certificate representing the CRL signer. Use of subsequent certificates is specific to the trust domain. For example, in a hierarchical trust model subsequent &tab;members are intermediate certificates of a certificate chain.

VerifyScope
A pointer to the CSSM_FIELD array indicating the CRL fields to be included in the CRL signature verification process. A NULL input verifies &tab;the signature assuming the module's default set of fields was used in the &tab;signaturing process (this can include all fields in the CRL).

ScopeSize
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.

Input/optional

CLHandle
The handle that describes the CL module that can be used to manipulate &tab;the certificates to be verified. If no CL module is specified, the TP module uses an assumed CL module, if required.

Return value
A CSSM_TRUE return value means the CRL can be trusted. If CSSM_FALSE is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
CSSM_CL_CrlVerify

TP_CrlSign

Purpose
The TP module decides whether the signer certificate is trusted to sign CRL. The signer certificate group is first authenticated and its applicability to perform this operation is determined. Once the trust is established, this operation signs the CRL.

Format
```
CSSM_DATA_PTR CSSMTPI TP_CrlSign (CSSM_TP_HANDLE TPHandle,
                                  CSSM_CL_HANDLE CLHandle,
                                  CSSM_CC_HANDLE CCHandle,
                                  const CSSM_DL_DB_LIST_PTR DBList,
                                  const CSSM_DATA_PTR CrlToBeSigned,
                                  const CSSM_CERTGROUP_PTR SignerCertGroup,
                                  const CSSM_FIELD_PTR SignScope,
                                  uint32 ScopeSize)
```

Parameters
Input

TPHandle
The handle that describes the TP module used to perform this function.

CLHandle
The handle that describes the CL module used to perform this function.

CCHandle
The handle that describes the context of the cryptographic operation.

DBList 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 &tab;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
A pointer to the CSSM_DATA structure containing a CRL to be signed.

SignerCertGroup
A group of one or more certificates that partially or fully represent the signer for this operation. The first certificate in the group is the target&tab;&tab;&tab;certificate representing the signer. Use of subsequent certificates is specific to the trust domain. For example, in a hierarchical trust model &tab;subsequent members are intermediate certificates of a certificate chain.

SignScope
A pointer to the CSSM_FIELD array containing the tags of the fields to be signed. A NULL input signs a default set of fields in the CRL.

ScopeSize
The number of entries in the sign scope list.

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.

Related information
CSSM_CL_CrlSign

TP_ApplyCrlToDb

Purpose
This function first determines whether the memory-resident CRL is trusted. The CRL is authenticated, its signer is verified, and its authority to update the data sources is determined. If trust is established, this function updates persistent storage to reflect entries in the CRL. This results in designating persistent certificates as revoked.

Format
```
CSSM_RETURN CSSMTPI TP_ApplyCrlToDb (CSSM_TP_HANDLE TPHandle,
                                     CSSM_CL_HANDLE CLHandle,
                                     CSSM_CSP_HANDLE CSPHandle,
                                     const CSSM_DL_DB_LIST_PTR DBList,
                                     const CSSM_DATA_PTR Crl)
```

Parameters
Input

TPHandle
The handle that describes the TP module used to perform this function.

Crl A pointer to the CSSM_DATA structure containing the CRL.

Input/optional

CLHandle
The handle that describes the certificate library module that can be used &tab;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 certificate library module is specified, the TP module uses an assumed CL module, if required. If optional, the caller will set this value to 0.

CSPHandle
The handle referencing a Cryptographic Service Provider to be used to verify signatures on the CRL determining whether to trust the CRL and &tab;apply it to the data store. The TP module is responsible for creating the cryptographic content structures requires for verification operation. If no CSP is specified, the TP module uses an assumed CSP to perform these operations. If optional, the caller will set this value to 0.

DBList A list of handle pairs specifying a DL module and a data store managed by that module. These data stores can contain certificates that might be affected by the CRL, they may contain CRLs, or both. If no DL and DB handle pairs are specified, the TP module must use an assumed DL &tab;module and an assumed data store for this operation. If optional, the caller will set this value to NULL.

Return value
A CSSM_TRUE return value means the CRL has been used to update the revocation status of certificates in the specified database. If CSSM_FALSE is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
CCSM_CL_CrlGetFirstItem
CSSM_CL_CrlGetNextItem
CSSM_DL_CertRevoke

TP_CertGroupConstruct

Purpose
This function builds a collection of certificates that together make up a meaningful credential for a given trust domain. For example, in a hierarchical trust domain, a certificate group is a chain of certificates from an end entity to a top-level CA. The constructed certificate group format (such as ordering) is implementation-specific. However, the subject or end-entity is always the first certificate in the group.

A partially constructed certificate group is specified in *CertGroupFrag*. The first certificate is interpreted to be the subject or end-entity certificate. Subsequent certificates in the *CertGroupFrag* structure may be used during the construction of a certificate group in conjunction with certificates found in *DBList*. The TP defines the certificates that will be included in the resulting set.

The constructed certificate group can be consistent locally or globally. Consistency can be limited to the local system if locally defined anchor certificates are inserted into the group.

Format
```
CSSM_CERTGROUP_PTR CSSMTPI TP_CertGroupConstruct (CSSM_TP_HANDLE TPHandle,
                                                  CSSM_CL_HANDLE CLHandle,
                                                  CSSM_CSP_HANDLE CSPHandle
                                                  CSSM_CERTGROUP_PTR CertGroupFrag,
                                                  CSSM_DL_DB_LIST_PTR DBList)
```

Parameters

Input

TPHandle
The handle to the TP module to perform this operation.

CLHandle
The handle to the CL module that can be used to manipulate and parse &tab;&tab;&tab;values in stored in the certgroup certificates. If no CL module is &tab;&tab;&tab;&tab;specified, the TP module uses an assumed CL module.

CSPHandle
The handle referencing a CSP to be used to perform this operation.

CertGroupFrag
The first certificate in the group represents the target certificate for which a group of semantically related certificates will be assembled. Subsequent intermediate certificates can be supplied by the caller. They&tab;need not be in any particular order.

DBList A list of handle pairs specifying a DL module and a data store managed by that module. These data stores should contain certificates (and &tab;possibly, other security object also). 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.

Related information

CSSM_TP_CertGroupPrune
CSSM_TP_CertGroupVerify

TP_CertGroupPrune

Purpose

This function removes certificates from a certificate group. The prune operation can remove those certificates that have been signed by any local CA, as it is possible that these certificates will not be meaningful on other systems.

This operation can also remove additional certificates that can be added to the certificate group, again using the *CertGroupConstruct* operation. The pruned certificate group should be suitable for transmission to external hosts, which can in turn reconstruct and verify the certificate group.

Format

```
CSSM_CERTGROUP_PTR CSSMTPI TP_CertGroupPrune (CSSM_TP_HANDLE TPHandle,
                                              CSSM_CL_HANDLE CLHandle,
                                              CSSM_CERTGROUP_PTR OrderedCertGroup,
                                              CSSM_DL_DB_LIST_PTR DBList)
```

Parameters

Input/optional

CLHandle
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.

Input

TPHandle
The handle to the TP module used to perform this operation.

OrderedCertGroup
The initial, complete set of certificates from which certificates will be selectively removed.

DBList &tab;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 object 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.

Related information

CSSM_TP_CertGroupConstruct
CSSM_TP_CertGroupVerify

TP_CertGroupVerify

Purpose

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.

Format

```
CSSM_BOOL CSSMTPI TP_CertGroupVerify (CSSM_TP_HANDLE TPHandle,
                                      CSSM_CL_HANDLE CLHandle,
                                      CSSM_DL_DB_LIST_PTR DBList,
                                      CSSM_CSP_HANDLE CSPHandle,
                                      const CSSM_FIELD_PTR  PolicyIdentifiers,
                                      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)
```

Parameters

Input

TPHandle
The handle to the TP module to perform this operation.

CSPHandle
The handle referencing a CSP to be used to perform this operation.

NumberofPolicyIdentifiers
The number of policy identifiers provided in the *PolicyIdentifiers* parameters.

CertToBeVerified
A pointer to the CSSM_CERTGROUP structure containing a certificate &tab;containing at least one signature for verification. An unsigned certificate template cannot be verified.

NumberofAnchorCerts
The number of anchor certificates provided in the *AnchorCerts* parameter.

ScopeSize
The number of entries in the verify scope list. If the verification scope is not specified, the input scope size must be zero.

Input/optional:

CLHandle
The handle to the CL module that can be used to manipulate and parse &tab;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 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.

PolicyIdentifiers
The policy identifier is an 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.

VerificationAbortOn
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 as follows in Table 5.

Table 5. CSSM_TP_STOP_ON Values

Value	Definition
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.

AnchorCerts
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.

VerifyScope
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.

Action An application-specific and application-defined action to be performed under the authority of the input certificate. If no action is specified, the TP &tab;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 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.

Output/optional

Evidence
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 &tab;list of certificates and CRLs that were used to establish the validity of the &tab;*CertToBeVerified*, but other objects may be appropriate for other types of trust policies.

Output

EvidenceSize
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 is verified. CSSM_FALSE if the certificate did not verify or an error condition occurred. Use CSSM_GetError to obtain the error code.

Related information

CSSM_TP_CertGroupConstruct
CSSM_TP_CertGroupPrune

Trust policy extensibility functions

The TP_PassThrough function is provided to allow TP developers to extend the certificate of the OCSF API. Because it is only exposed to OCSF as a function pointer, its name internal to the TP can be assigned at the discretion of the TP module developer. However, its parameter list and return value must match.

TP_PassThrough

Purpose

The TP module allows clients 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.

Format

```
CSSM_DATA_PTR CSSMTPI 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)
```

Parameters

Input

TPHandle
The handle that describes the TP module used to perform this function.

CLHandle
The handle that describes the CL module used to perform this function.

DLHandle
The handle that describes the DL module used to perform this function.

DBHandle
The handle that describes the data storage used to perform this function.

CCHandle
The handle that describes the context of the cryptographic operation.

PassThroughId
An identifier assigned by the TP module to indicate the exported function to perform.

InputParams
A pointer to the CSSM_DATA structure containing parameters to be interpreted in a function-specific manner by the TP module.

Return value

A pointer to the CSSM_DATA 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.

Trust policy Attach/Detach example

TPHandle &tab;The Trust Policy (TP) module performs certain operations when OCSF attaches to or detaches from it. TP modules use **_init** in conjuction with the DLLMain routine to perform those operations, as shown in the following example.

```
_init
   BOOL_init( )
   {
     BOOL rc;
     rc =  DllMain(NULL, DLL_PROCESS_ATTACH, NULL);
     return (rc);
   }
```

DLLMain

```
#include<cssm.h>
CSSM_GUID tp_guid =
{ 0x83bafc39, 0xfac1, 0x11cf, { 0x81, 0x72, 0x0, 0xaa, 0x0, 0xb1, 0x99, 0xdd } };

BOOL  DllMain (HANDLE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    switch (dwReason)
    {
        case DLL_PROCESS_ATTACH:
        {
           CSSM_SPI_TP_FUNCS_PTR FunctionTable;
           CSSM_SPI_MEMORY_FUNCS_PTR UpcallTable;

         /* Allocate TP memory for pointers */
         FunctionTable = (CSSM_SPI_TP_FUNCS_PTR)malloc (sizeof
                              (CSSM_SPI_TP_FUNCS));
         UpcallTable = (CSSM_SPI_MEMORY_FUNCS_PTR)malloc (sizeof
                              (CSSM_SPI_MEMORY_FUNCS);

         /* Initialize TP callback functions */
         FunctionTable->CertSign = CertSign;
         FunctionTable->CertRevoke = CertRevoke;
         FunctionTable->CrlVerify = CrlVerify;
         FunctionTable->CrlSign = CrlSign;
         FunctionTable->ApplyCrlToDb = ApplyCrlToDb;
         FunctionTable->CertGroupConstruct = CertGroupConstruct;
         FunctionTable->CertGroupPrune = CertGroupPrune;
         FunctionTable->CertGroupVerify = CertGroupVerify;
         FunctionTable->PassThrough = NULL;

          /* Call CSSM_RegisterServices to register the FunctionTable */
          /* with OCSF and to receive the application's memory upcall table */
          if (CSSM_RegisterServices (&tp_guid, FunctionTable, UpcallTable)
                                != CSSM_OK)
             return FALSE;
          break;
   }
case DLL_THREAD_ATTACH:
   break;
case DLL_THREAD_DETACH:
   Break;
case DLL_PROCESS_DETACH:
   if (CSSM_DeregisterServices (&tp_guid) != CSSM_OK)
           return FALSE;
   break;
   }
return TRUE;
}
```

Trust policy OCSF errors

This section defines the error code range that is defined by OCSF for use by all Trust Policies (TPs) in describing common error conditions. A TP may also define and return vendor-specific error codes. The error codes defined by OCSF are considered to be comprehensive and few if any vendor-specific codes should be required. Applications must consult vendor-supplied documentation for the specification and description of any error codes defined outside of this specification.

All Trust Policy service provider interface (TP SPI) 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 - OCSF 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. 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 OCSF 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 OCSF, those errors have a set of predefined numeric values that are reserved by OCSF, 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 6 lists the range of error numbers defined by OCSF for TP modules and those available for use individual Trust Policy (TP) modules. See the *z/OS Open Cryptographic Services Facility Application Programming* book for a list of all the error codes and descriptions for TP.

Table 6. Trust Policy Module Error Numbers

Error Number Range	**Description**
7000-7999	TP errors defined by OCSF
8000-8999	TP errors reserved for individual TP modules

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.

Chapter 4. Certificate library interface

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. These manipulations encapsulate the complete life cycle of a certificate and the key pair associated with that certificate. Certificate and CRLs are related by the life cycle model and by the data formats used to represent them. For this reason, a single, cohesive library should manipulate these objects.

The CL encapsulates format-specific knowledge into a library that an application can access through OCSF. These libraries allow applications and service provider modules to interact with Certificate Authorities (CAs) and to use certificates and CRLs for services such as signing, verification, creation and revocation without requiring knowledge of the certificate and CRL formats.

CLs 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 Trust Policy (TP) module to use data storage service provider modules to make objects persistent (if appropriate).

Certificate life cycle

The CL provides support for the certificate life cycle and for format-specific certificate or CRL manipulation, services that an application can access through OCSF. These libraries allow applications and service provider modules to create, sign, verify, and revoke certificates without requiring knowledge of certificate and CRL format and encoding.

A certificate is a form of credential. Under current certificate models, such as X.509, Simple Distributed Security Infrastructure (SDSI), Simple Public Key Infrastructure (SPKI), etc., a single certificate represents the identity of an entity (in the form of a binding between a name and a public key) and optionally associates authorizations with that entity. When a certificate is issued, the issuer includes a digital signature on the certificate. Verification of this signature is the mechanism used to establish trust in the identity and authorizations recorded in the certificate. Certificates can be signed by one or more other certificates. Root certificates are self-signed. The syntactic process of signing corresponds to establishing a trust relationship between the entities identified by the certificates.

Figure 2 presents the certificate life cycle. It begins with the registration process. During registration, the authenticity of a user's identity is verified. This can be a two-part process beginning with manual procedures requiring physical presence, followed by backoffice procedures to register results for use by the automated system. The level of verification associated with the identity of the individual will depend on the Security Policy and Certificate Management Practice Statements that apply to the individual who will receive a certificate, and the domain in which that certificate will be issued and used.

After registration, keying material is generated and a certificate is created. Once the private key material and public key certificate are issued to a user, and backed up if appropriate, the active phase of the certificate management life cycle begins. The active phase includes:

- Retrieval - Retrieves a certificate from a remote repository such as an X.500 directory.
- Verification - Verifies the validity dates and signatures on a certificate and revocation status.
- Revocation - Asserts that a previously legitimate certificate is no longer a valid certificate.
- Recovery - When an end user can no longer access encryption keys (e.g., forgotten password).
- Update - Issues a new public/private keypair when a legitimate pair has or will expire soon.

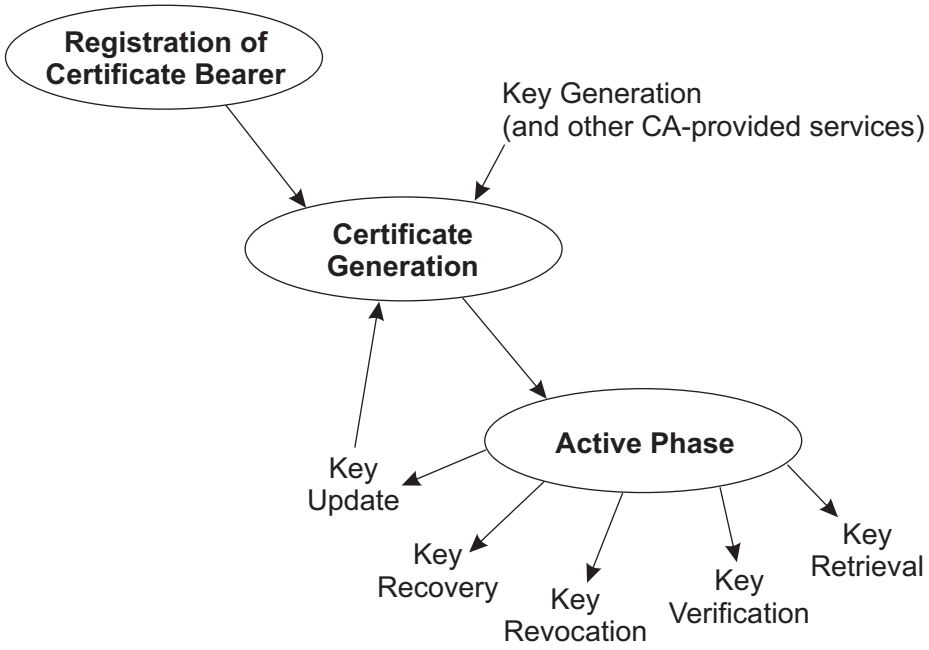

Figure 2. Certificate Life Cycle States and Actions

Certificate library interface specification

The Certificate Library Interface (CLI) specifies the functions that a CL may make available to applications via OCSF in order to support a certificate and a CRL format. These functions mirror the OCSF API for certificates and CRLs. These functions include the basic areas of functionality expected of a CL, which include certificate operations, CRL operations, extensibility functions, and module management functions. The CL developer may choose to implement some or all of these CLI functions. The available functions are made known to OCSF at module attach time when it receives the CL's function table. In the function table, any unsupported function must have a NULL function pointer. The CL module developer is responsible for making the certificate format and general functionality known to application developers.

Certificate operations fall into three general areas, including:

- **Cryptographic Operations -** These operations include signing a certificate and verifying the signature on a certificate. It is expected that the CL will determine the certificate fields to be signed or verified, and will manage the interaction with a Cryptographic Service Provider (CSP) to perform the signing or verification.

- **Certificate Field Management -** Fields are added to a certificate when it is created. After the certificate is signed, the fields cannot be modified in any way. However, they can be queried for their values using the OCSF certificate interface.
- **Certificate Format Translation -** In the heterogeneous world of multiple certificate formats, CL modules may want to provide the service of translating between certificate formats. This translation would involve mapping the fields from one certificate format into another certificate format, while maintaining the original format for integrity verification purposes. For example, an X.509 Version 1 certificate may be exported to a Simple Distributed Security Infrastructure (SDSI) format or imported into an X.509 Version 3 certificate, but the original data and signature must somehow be maintained. The supported import and export types are registered with OCSF as part of CL installation.

To support new certificate types and new uses of certificates, the sign and verify operations in the CLI support a scope parameter. The scope parameter enables an application to sign a portion of the certificate, namely, the fields identified by the scope. This provides support for certificate models that permit field signing. CL modules that support existing certificate formats, such as X.509 Version 1, which sign and verify a predefined portion of the certificate, will ignore this parameter.

The CL module's certificate format is exposed via its fields. These fields will consist of tag/value pairs, where the tag is an object identifier (OID). These OIDs reference specific data types or data structures within the certificate or CRL. OIDs are defined by the CL developer at a granularity appropriate for the expected usage of the CL.

Operations on CRLs are comprised of cryptographic operations and field management operations on the CRL, as a whole, and on individual revocation records. The entire CRL can be signed or verified. This will ensure the integrity of the CRL's contents as it is passed between systems. Individual revocation records are signed when they are revoked and verified when they are queried. Certificates may be revoked and unrevoked by adding or removing them from the CRL at any time prior to its being signed. The contents of the CRL can be queried for all of its revocation records, specific certificates, or individual CRL fields.

A pass-through function is included in the CLI to allow CLs to expose additional services beyond what is currently defined in the OCSF API. These services should be syntactic in nature, meaning that they should be dependent on the data format of the certificates and CRLs manipulated by the library. OCSF will pass an operation identifier and input parameters from the application to the appropriate CL. Within the CL_PassThrough function in the CL, the input parameters will be interpreted and the appropriate operation performed. The CL developer is responsible for making known to the application the identity and parameters of the supported passthrough operations.

Certificate library 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 OCSF API function calls that will be passed without modification to the appropriate CL. The CL is then responsible for interpreting the data structures and returning the appropriate data structure to the calling application through the OCSF Framework. These data structures are defined in the header file, cssmtype.h, which is distributed with OCSF.

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.

CSSM_CS_SERVICES

This bit-mask defines the additional certificate-creation-related services that an issuing CA (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. Additional services can be defined over time.

```
typedef uint32 CSSM_CA_SERVICES;
/* bit masks for additional CA services at cert enroll */
#define CSSM_CA_KEY_ARCHIVE        0x0001 /* archive cert & keys */
#define CSSM_CA_CERT_PUBLISH       &tab;0x0002 /* cert in directory service */
#define CSSM_CA_CERT_NOTIFY_RENEW  &tab;0x0004 /* notify at renewal time */
#define CSSM_CA_CRL_DISTRIBUTE     0x0010 /* push CRL to everyone */
```

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;
```

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.

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 Intrastructure (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 uint32 CSSM_CERT_TYPE,*CSSM_CERT_TYPE_PTR;
/* bit masks for supported cert types */
#define  CSSM_CERT_UNKNOWN        0x00000000
#define  CSSM_CERT_X_509v1        0x00000001
#define  CSSM_CERT_X_509v2        0x00000002
#define  CSSM_CERT_X_509v3        0x00000004
#define  CSSM_CERT_Fortezza       0x00000008
#define  CSSM_CERT_PGP            0x00000010
#define  CSSM_CERT_SPKI           0x00000020
#define  CSSM_CERT_SDSIv1         0x00000040
#define  CSSM_CERT_Intel          0x00000080
#define  CSSM_CERT_ATTRIBUTE_BER  0x00000100
#define  CSSM_CERT_X509_CRL       0x00000200
#define  CSSM_CERT_LAST           0x00007fff

/* 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 */
#define  CSSM_CL_CUSTOM_CERT_TYPE 0x08000
```

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;
```

Definitions:
CertClassName
Name of a certificate class issued by this CA.
CSCert CA certificate for this cert class.

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.

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
&tab;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 &tab;manages.

ProductFlags
A bit-mask indicating any selectable features of the embedded product that the CL module selected for use.

CSSM_CL_HANDLE

The CSSM_CL_HANDLE is used to identify the association between an application thread and an instance of a CL module. CSSM_CL_HANDLE is assigned when an application causes OCSF to attach to a CL. It is freed when an application causes OCSF 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
```

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 OCSF registry when the CL module is installed with OCSF. 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 sub-service.

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.

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.

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 OCSF.

```
typedef struct cssm_data {
    uint32 Length;
    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.

CSSM_FIELD

This structure contains the 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.

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 OCSF.

```
typedef uint32 CSSM_HEADERVERSION

#define CSSM_KEYHEADER_VERSION (2)
```

CSSM_KEY

This structure is used to represent keys in OCSF.

```
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.

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 keyheader structure.

Cspld 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 7.

Table 7. Keyblob Type Identifiers

Keyblob Type 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	Other keyblob type.

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 &tab;CSSM_KEYBLOB_RAW_FORMAT identifier must be used, otherwise a CSSM_KEYBLOB_REF_FORMAT identifier is used. Any of the following values in Table 8 are valid as format identifiers.

Table 8. Keyblob Format Identifiers

Keyblob Format Identifiers	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
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

Table 8. Keyblob Format Identifiers (continued)

Keyblob Format Identifiers	Description
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 OCSF algorithm IDs may be used.

KeyClass
Class of key contained in the key blob. Valid key classes are as follows in Table 9.

Table 9. 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 Digital Signature Algorithm (DSA) key would be represented by the size of its prime. Symmetric key sizes describe the &tab;actual number of bits in the key. For example, Data Encryption Standard (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 10).

KeyUsage
A bit-mask representing the valid uses of the key. Any of the following values in Table 11 are valid.

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 if the date is 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.

Table 10. KeyAttribute Flags

Attribute	Desccription
CSSM_KEYATTR_PERMANENT	Key is stored persistently in the CSP, i.e., PKCS11 token object.
CSSM_KEYATTR_PRIVATE	Key is a private object and protected by either user login, a password, or both.
CSSM_KEYATTR_MODIFIABLE	The key or its attributes can be modified.
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.

Table 11. 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 Message Authentication Codes (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.

Table 11. Key Usage Flags (continued)

Usage Mask	Description
CSSM_KEYUSE_VERIFY_RECOVER	Key can be used to verify signatures with message recovery. This form of a signature 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.

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.

CSSM_KEY_TYPE

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

CSSM_SPI_MEMORY_FUNCS

This structure is used by OCSF to pass an application's memory function table to the service provider modules. The functions are used when memory needs to be allocated by the service provider module for returning data structures to the applications.

```
typedef struct cssm_spi_func_tbl {
void *(*malloc_func) (CSSM_HANDLE AddInHandle, uint32 Size);
void (*free_func) (CSSM_HANDLE AddInHandle, void *MemPtr);
void  *(*realloc_func) (CSSM_HANDLE AddInHandle, void *MemPtr, uint32 Size);
void  *(*calloc_func) (CSSM_HANDLE AddInHandle, uint32 Num, uint32 Size);
} CSSM_SPI_MEMORY_FUNCS, *CSSM_SPI_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.

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;
```

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;
```

CSSM_REVOKE_REASON

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

```
typedef enum cssm_revoke_reason {
    CSSM_REVOKE_CUSTOM,
    CSSM_REVOKE_UNSPECIFIC,
    CSSM_REVOKE_KEYCOMPROMISE,
    CSSM_REVOKE_CACOMPROMISE,
    CSSM_REVOKE_AFFILIATIONCHANGED,
    CSSM_REVOKE_SUPERCEDED,
    CSSM_REVOKE_CESSATIONOFOPERATION,
    CSSM_REVOKE_CERTIFICATEHOLD,
    CSSM_REVOKE_CERTIFICATEHOLDRELEASE,
    CSSM_REVOKE_REMOVEFROMCRL
} CSSM_REVOKE_REASONrtificate Operations
```

Certificate library operations

This section describes the function prototypes and error codes expected for the functions in the CLI. The functions will be exposed to OCSF via a function table, so the function names may vary at the discretion of the CL developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications.

CL_CertAbortQuery

Purpose

This function terminates the query initiated by CL_CertGetFirstFieldValue and allows the CL to release all intermediate state information associated with the query.

Format

```
CSSM_RETURN CSSMAPI CL_CertAbortQuery CSSM_CL_HANDLE CLHandle, CSSM_HANDLE ResultsHandle)
```

Parameters

Input

CLHandle

The handle that describes the service provider CL module used to perform this function.

ResultsHandle

The handle that identifies the results of a certificate 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.

Related information

CL_CertGetFirstFieldValue
CL_CertGetNextFieldValue

CL_CertCreateTemplate

Purpose

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.

Format

```
CSSM_DATA_PTR CSSMAPI CL_CertCreateTemplate (CSSM_CL_HANDLE CLHandle,
                                             const CSSM_FIELD_PTR CertTemplate,
                                             uint32 NumberOfFields)
```

Parameters

Input

CLHandle

The handle that describes the service provider CL module used to perform this function.

CertTemplate

A pointer to an array of OID/value pairs that identify the field values to initialize a new certificate.

NumberOfFields

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.

Related information
CL_CertRequest
CL_CertGetFirstFieldValue

CL_CertDescribeFormat

Purpose
This function returns a list of the OIDs used to describe the certificate format supported by the specified CL.

Format
```
CSSM_OID_PTR CSSMAPI CL_CertDescribeFormat (CSSM_CL_HANDLE CLHandle, uint32 *NumberOfFields)
```

Parameters
Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

Output

NumberOfFields
The length of the output OID array.

Return value
A pointer to the array of CSSM_OID structures which are supported for certificate operations in the specified CL module. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
CL_CertGetFirstFieldValue

CL_CertExport

Purpose
This function exports a certificate from the native format of the specified CL into the specified target certificate format.

Format
```
CSSM_DATA_PTR CSSMAPI CL_CertExport (CSSM_CL_HANDLE CLHandle,
                                     CSSM_CERT_TYPE TargetCertType,
                                     const CSSM_DATA_PTR NativeCert)
```

Parameters
CLHandle
The handle that describes the service provider CL module used to perform this function.

TargetCert
A unique value that identifies the target type of the certificate being exported.

NativeCert
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.

Related information
CL_CertImport

CL_CertGetAllFields

Purpose
This function returns a list of the fields in the input certificate, as described by their OID/value pairs.

Format
```
CSSM_FIELD_PTR CSSMAPI CL_CertGetAllFields (CSSM_CL_HANDLE CLHandle,
                                            const CSSM_DATA_PTR Cert,
                                            uint32 *NumberOfFields)
```

Parameters
Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

Cert A pointer to the CSSM_DATA structure containing the certificate whose fields will be returned.

Output

NumberOfFields
The length of the output CSSM_FIELD array.

Return value
A pointer to an array of CSSM_FIELD structures that describe the contents of the certificate using OID/value pairs. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
CL_CertGetFirstFieldValue

CL_CertGetFirstFieldValue

Purpose
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.

Format
```
CSSM_DATA_PTR CSSMAPI CL_CertGetFirstFieldValue (CSSM_CL_HANDLE CLHandle,
                                                 const CSSM_DATA_PTR Cert,
                                                 const CSSM_OID_PTR CertField,
                                                 CSSM_HANDLE_PTR ResultsHandle,
                                                 uint32 *NumberOfMatchedFields)
```

Parameters

Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

Cert A pointer to the CSSM_DATA structure containing the certificate.

CertField
A pointer to an OID that identifies the field value to be extracted from the *Cert*.

Output

ResultsHandle
A pointer to the CSSM_HANDLE that should be used to obtain any additional matching fields.

NumberOfMatchedFields
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.

Related information

CL_CertGetNextFieldValue
CL_CertAbortQuery
CL_CertGetAllFields
CL_CertDescribeFormat

CL_CertGetKeyInfo

Purpose

This function obtains information about the certificate's public key. Ideally, this information comprises the key fields the application needs to create a cryptographic context that uses this certificate's key.

Format

```
CSSM_KEY_PTR CSSMAPI CL_CertGetKeyInfo (CSSM_CL_HANDLE CLHandle, const CSSM_DATA_PTR Cert)
```

Parameters

Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

Cert 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.

CL_CertGetNextFieldValue

Purpose
This function returns the next certificate field that matched the OID in a call to CL_CertGetFirstFieldValue.

Format
```
CSSM_DATA_PTR CSSMAPI CL_CertGetNextFieldValue (CSSM_CL_HANDLE CLHandle, CSSM_HANDLE ResultsHandle)
```

Parameters
Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

ResultsHandle
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.

Related information
CL_CertGetFirstFieldValue
CL_CertAbortQuery

CL_CertImport

Purpose
This function imports a certificate from the input format into the native format of the specified CL.

Format
```
CSSM_DATA_PTR CSSMAPI CL_CertImport (CSSM_CL_HANDLE CLHandle,
                                     CSSM_CERT_TYPE ForeignCertType,
                                     const CSSM_DATA_PTR ForeignCert)
```

Parameters
CLHandle
The handle that describes the service provider CL module used to perform this function.

ForeignCertType
A unique value that identifies the type of the certificate being imported.

Cert A pointer to the CSSM_DATA structure containing the certificate to be imported into the native type.

Return value
A pointer to the CSSM_DATA structure containing the native-type certificate imported from the foreign certificate. Use CSSM_GetError to obtain the error code.

Related information
CL_CertExport

CL_CertSign

Purpose

This function signs the fields of the input certificate as indicated by the SignScope array.

Format

```
CSSM_DATA_PTR CSSMAPI CL_CertSign (CSSM_CL_HANDLE CLHandle,
                                   CSSM_CC_HANDLE CCHandle,
                                   const CSSM_DATA_PTR SubjectCert,
                                   const CSSM_DATA_PTR SignerCert,
                                   const CSSM_FIELD_PTR SignScope,
                                   uint32 ScopeSize)
```

Parameters

Input

CLHandle
: The handle that describes the service provider CL module used to perform this function.

CCHandle
: The handle that describes the context of this cryptographic operation.

SubjectCert
: A pointer to the CSSM_DATA structure containing the certificate to be signed.

SignerCert
: A pointer to the CSSM_DATA structure containing the certificate to be used to sign the subject certificate.

SignScope
: 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
: 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.

Related information

CL_CertVerify

CL_CertVerify

Purpose

This function verifies that the signed certificate has not been altered since it was signed by the designated signer. It does this by verifying the digital signature on the VerifyScope fields.

Format

```
CSSM_BOOL CSSMAPI CL_CertVerify (CSSM_CL_HANDLE CLHandle,
                                 CSSM_CC_HANDLE CCHandle,
                                 const CSSM_DATA_PTR SubjectCert,
                                 const CSSM_DATA_PTR SignerCert,
                                 const CSSM_FIELD_PTR VerifyScope,
                                 uint32 ScopeSize)
```

Parameters

Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

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

SubjectCert
A pointer to the CSSM_DATA structure containing the signed certificate

SignerCert
A pointer to the CSSM_DATA structure containing the certificate used to sign the subject certificate.

VerifyScope
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be verified. A NULL input verifies all the fields in the certificate.

ScopeSize
The number of entries in the verify scope list.

Return value

CSSM_TRUE if the certificate verified. CSSM_FALSE if the certificate did not verify or an error condition occurred. Use CSSM_GetError to obtain the error code.

Related information

CL_CertSign

Certificate revocation list operations

This section describes the function prototypes supported by a CL module for operations on CRLs. The functions will be exposed to OCSF through a function table, so the function names may vary at the discretion of the CL developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications.

CL_CrlAbortQuery

Purpose

This function terminates the query initiated by CL_CrlGetFirstFieldValue and allows the CL to release all intermediate state information associated with the query.

Format

```
CSSM_RETURN CSSMAPI CL_CrlAbortQuery (CSSM_CL_HANDLE CLHandle, CSSM_HANDLE ResultsHandle)
```

Parameters

Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

ResultsHandle
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.

Related information
CL_CrlGetFirstFieldValue
CL_CrlGetNextFieldValue

CL_CrlAddCert

Purpose
This function revokes the input certificate by adding a record representing the certificate to the CRL. It uses the revoker's certificate to sign the new record in the CRL. The reason for revoking the certificate may also be stored in the revocation record.

Format
```
CSSM_DATA_PTR CSSMAPI CL_CrlAddCert (CSSM_CL_HANDLE CLHandle,
                                     CSSM_CC_HANDLE CCHandle,
                                     const CSSM_DATA_PTR Cert,
                                     const CSSM_DATA_PTR RevokerCert,
                                     CSSM_REVOKE_REASON RevokeReason,
                                     const CSSM_DATA_PTR OldCrl)
```

Parameters
Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

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

Cert A pointer to the CSSM_DATA structure containing the certificate to be revoked.

RevokerCert
A pointer to the CSSM_DATA structure containing the revoker's certificate.

RevokeReason
The reason for revoking the certificate.

OldCrl 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.

Related information
CL_CrlRemoveCert

CL_CrlCreateTemplate

Purpose
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, though the module developer may specify a set of fields that must be or cannot be set using this operation. Subsequent values may be set using the CL_CrlSetFields operation.

Format

```
CSSM_DATA_PTR CSSMAPI CL_CrlCreateTemplate (CSSM_CL_HANDLE CLHandle,
                                            const CSSM_FIELD_PRT CrlTemplate,
                                            uint32 NumberOfFields)
```

Parameters

Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

CrlTemplate
An array of OID/value pairs specifying the initial values for descriptive data fields of the new CRL.

NumberOfFields
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.

CL_CrlDescribeFormat

Purpose

This function returns a list of the OIDs used to describe the CRL format supported by the specified CL.

Format

```
CSSM_OID_PTR CSSMAPI CL_CrlDescribeFormat (CSSM_CL_HANDLE CLHandle, uint32 *NumberOfFields)
```

Parameters

Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

Output

NumberOfFields
The length of the output array.

Return value

A pointer to the array of CSSM_OID structures which are supported for CRL operations in the specified CL module. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

CL_CrlGetFirstFieldValue

Purpose

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 is an output parameter, as is the ResultsHandle to be used to retrieve the remaining matching fields.

Format

```
CSSM_DATA_PTR CSSMAPI CL_CrlGetFirstFieldValue (CSSM_CL_HANDLE CLHandle,
                                                const CSSM_DATA_PTR Crl,
                                                const CSSM_OID_PTR CrlField,
                                                CSSM_HANDLE_PTR ResultsHandle,
                                                uint32 *NumberOfMatchedFields)
```

Parameters

Input

CLHandle
: The handle that describes the service provider CL module used to perform this function.

Crl
: A pointer to the CSSM_DATA structure that contains the CRL from which the first revocation record will be retrieved.

CrlField
: A pointer to an OID that identifies the field value to be extracted from the *Crl*.

Output

ResultsHandle
: A pointer to the CSSM_HANDLE, which should be used to obtain any additional matching fields.

NumberOfMatchedFields
: 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.

Related information

CL_CrlGetNextFieldValue
CL_CrtAbortQuery

CL_CrlGetNextFieldValue

Purpose

This function returns the next CRL field that matched the OID in a call to CL_CrlGetFirstFieldValue.

Format

```
CSSM_DATA_PTR CSSMAPI CL_CrlGetNextFieldValue (CSSM_CL_HANDLE CLHandle, CSSM_HANDLE ResultsHandle)
```

Parameters

Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

ResultsHandle
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, which 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.

Related information
CL_CrlGetFirstFieldValue
CL_CrlAbortQuery

CL_CrlRemoveCert

Purpose
This function unrevokes a certificate by removing it from the input CRL.

Format
```
CSSM_DATA_PTR CSSMAPI CL_CrlRemoveCert (CSSM_CL_HANDLE CLHandle,
                                        const CSSM_DATA_PTR Cert,
                                        const CSSM_DATA_PTR OldCrl)
```

Parameters
Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

Cert A pointer to the CSSM_DATA structure containing the certificate to be unrevoked.

OldCrl A pointer to the CSSM_DATA structure containing the CRL from which the certificate will 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.

Related information
CL_CrlAddCert

CL_CrlSetFields

Purpose
This function will set the fields of the input CRL to the new values specified by the input OID/value pairs. The module developer may specify a set of fields that must be or cannot be set using this operation. This operation is valid only if the CRL has not been closed by the process of signing the CRL (i.e., execution of the function CL_CrlSign). Once the CRL has been signed, fields cannot be changed.

Format

```
CSSM_DATA_PTR CSSMAPI CL_CrlSetFields (CSSM_CL_HANDLE CLHandle,
                                       const CSSM_FIELD_PRT CrlTemplate,
                                       uint32 NumberOfFields,
                                       const CSSM_DATA_PTR OldCrl)
```

Parameters

Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

CrlTemplate
Any array of field OID/value pairs containing the values to initialize the CRL attribute fields.

NumberOfFields
The number of OID/value pairs specified in the CrlTemplate input parameter.

OldCrl 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

CL_CrlSign

Purpose

This function signs, in accordance with the specified cryptographic context, the fields of the CRL indicated in the *SignScope* parameter.

Format

```
CSSM_DATA_PTR CSSMAPI 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)
```

Parameters

Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

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

UnsignedCrl
A pointer to the CSSM_DATA structure containing the CRL to be signed.

SignerCert
A pointer to the CSSM_DATA structure containing the certificate to be used to sign the CRL.

SignScope
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 CRL.

ScopeSize
The number of entries in the sign scope list.

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.

Related information
CL_CrlVerify

CL_CrlVerify

Purpose
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 on the VerifyScope fields.

Format
```
CSSM_BOOL CSSMAPI CL_CrlVerify (CSSM_CL_HANDLE CLHandle,
                                CSSM_CC_HANDLE CCHandle,
                                const CSSM_DATA_PTR SubjectCrl,
                                const CSSM_DATA_PTR SignerCert,
                                const CSSM_FIELD_PTR VerifyScope,
                                uint32 ScopeSize)
```

Parameters
Input

CLHandle
The handle that describes the service provider CL module used to perform this function.

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

SubjectCrl
A pointer to the CSSM_DATA structure containing the CRL to be verified.

SignerCert
A pointer to the CSSM_DATA structure containing the certificate used to sign the CRL.

VerifyScope
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be verified. A NULL input verifies all the fields in the CRL.

ScopeSize
The number of entries in the verify scope list.

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.

Related information
CL_CrlSign

CL_IsCertInCrl

Purpose

This function searches the CRL for a record corresponding to the certificate.

Format

```
CSSM_BOOL VSSMAPI CL_IsCertInCrl (CSSM_CL_HANDLE CLHandle,
                                  const CSSM_DATA_PTR Cert,
                                  const CSSM_DATA_PTR Crl)
```

Parameters

Input

CLHandle
: The handle that describes the service provider CL module used to perform this function.

Cert
: A pointer to the CSSM_DATA structure containing the certificate to be located.

Crl
: 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.

Certificate library extensibility functions

The CL_PassThrough function is provided to allow CL developers to extend the certificate and CRL format-specific functionality of the OCSF API. Because it is only exposed to OCSF as a function pointer, its name internal to the CL can be assigned at the discretion of the CL module developer. However, its parameter list and return value must match.

CL_PassThrough

Purpose

This function allows applications to call CL module-specific operations.

Format

```
void * CSSMAPI CL_PassThrough (CSSM_CL_HANDLE CLHandle,
                               CSSM_CC_HANDLE CCHandle,
                               uint32 PassThroughID,
                               const void * InputParams)
```

Parameters

Input

CLHandle
: The handle that describes the service provider CL module used to perform this function.

CCHandle
: The handle that describes the context of the cryptographic operation.

PassThroughId
: An identifier assigned by the CL module to indicate the function to perform.

InputParams

A pointer to a module, implementation-specific structure containing parameters to &tab;be interpreted in a function-specific manner by the requested CL module. This parameter can be used as a pointer to an array of void pointers.

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.

Certificate library Attach/Detach example

The Certificate Library (CL) module is responsible for performing certain operations when OCSF attaches to and detaches from it. CL modules use **_init** in conjunction with the DLLMain routine to perform those operations, as shown in the following example.

```
_init
   BOOL_init( )
   {
     BOOL rc;
     rc =  DllMain(NULL, DLL_PROCESS_ATTACH, NULL);
     return (rc);
   }
```

DLLMain

```
#include <cssm.h>
CSSM_GUID my_clm_guid =
{ 0x83bafc39, 0xfac1, 0x11cf, { 0x81, 0x72, 0x0, 0xaa, 0x0, 0xb1, 0x99, 0xdd } };

BOOL  DllMain ( HANDLE hInstance, DWORD dwReason, LPVOID lpReserved)
{
switch (dwReason)
{
case DLL_PROCESS_ATTACH:
{
CSSM_SPI_CL_FUNCS FunctionTable;
CSSM_SPI_FUNC_TBL_PTR UpcallTable;

/* Fill in FunctionTable with function pointers */
FunctionTable.CertSign                  = CL_CertSign;
          FunctionTable.CertVerify               = CL_CertVerify;
          FunctionTable.CertCreateTemplate       = CL_CertCreateTemplate;
          FunctionTable.CertGetFirstFieldValue   = CL_CertGetFirstFieldValue;
          FunctionTable.CertGetNextFieldValue    = CL_CertGetNextFieldValue;
         FunctionTable.CertAbortQuery           = CL_CertAbortQuery;
         FunctionTable.CertGetKeyInfo           = CL_CertGetKeyInfo;
         FunctionTable.CertGetAllFields         = CL_CertGetAllFields;
         FunctionTable.CertImport               = NULL;
         FunctionTable.CertExport               = NULL;
         FunctionTable.CertDescribeFormat       = CL_CertDescribeFormat;
         FunctionTable.CrlCreateTemplate        = CL_CrlCreateTemplate;
         FunctionTable.CrlSetFields             = CL_CrlSetFields;
         FunctionTable.CrlAddCert               = CL_CrlAddCert;
         FunctionTable.CrlRemoveCert            = CL_CrlRemoveCert;
         FunctionTable.CrlSign                  = CL_CrlSign;
         FunctionTable.CrlVerify                = CL_CrlVerify;
         FunctionTable.IsCertInCrl              = CL_IsCertInCrl;
          FunctionTable.CrlGetFirstFieldValue    = CL_CrlGetFirstFieldValue;
         FunctionTable.CrlGetNextFieldValue     = CL_CrlGetNextFieldValue;
         FunctionTable.CrlAbortQuery            = CL_CrlAbortQuery;
         FunctionTable.CrlDescribeFormat        = CL_CrlDescribeFormat;
         FunctionTable.PassThrough              = CL_PassThrough;

   /* Call CSSM_RegisterServices to register the FunctionTable */
   /* with OCSF and to receive the application's memory upcall table */
   if (CSSM_RegisterServices (&my_clm_guid, FunctionTable, &UpcallTable) !=    CSSM_OK)
      return FALSE;
```

```
   /* Make the upcall table available to all functions in this library */

   break;
   }

case DLL_THREAD_ATTACH:

   break;
case DLL_THREAD_DETACH:
   break;
case DLL_PROCESS_DETACH:
   if (CSSM_CL_DeregisterServices (&my_clm_guid) != CSSM_OK)
      return FALSE;
   break;
}
return TRUE;
```

Certificate operations examples

This section contains sample implementations of certificate functions in the CL.

CL_CertCreateTemplate

```
/*-----------------------------------------------------------------------
 * Name: CL_CertCreateTemplate
 *
 * Description:
 * This function allocates and initializes memory for a certificate
 * based on the input tag/values pairs. The returned certificate
 * must be signed using the CSSM_CL_CertSign function.
 *
 * Parameters:
 * CLHandle (input)       : A handle to a CL module.
 * CertTemplate (input)   : A pointer to an array of tag/value pairs
 *                          which identify the fields of the new certificate
 * NumberOfFields (input) : The length of the CertTemplate array
 *
 * Return value:
 * The new certificate
 *
 * Error codes:
 * CSSM_CL_INVALID_CL_HANDLE
 * CSSM_CL_INVALID_FIELD_POINTER
 * CSSM_CL_INVALID_TEMPLATE
 * CSSM_CL_MEMORY_ERROR
 * CSSM_CL_UNSUPPORTED_OPERATION
 * CSSM_CL_CERT_CREATE_FAIL
*-----------------------------------------------------------------------*/
CSSM_DATA_PTR CSSMAPI CL_CertCreateTemplate (CSSM_CL_HANDLE CLHandle,
                                             const CSSM_FIELD_PTR CertTemplate,
                                             uint32 NumberOfFields)
{
    /* Initializations */
    CSSM_CERTIFICATE_PTR cert_ptr = NULL;
    CSSM_DATA_PTR  packed_cert_ptr = NULL;
    CSSM_ERROR_PTR err_ptr = NULL;
    uint32 i=0;

    /* Check inputs */
      /* Check that this is a valid CLHandle */
    if (CLHandle == 0)
    {
        CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_CL_HANDLE);
        return NULL;
    }
        /* Check that the NumberOfFields is greater than 0
           and that the CertTemplate pointer is not NULL */
    if ( !NumberOfFields || !CertTemplate)
    {
        CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_TEMPLATE);
        return NULL;
    }
        /* Check that CertTemplate is a valid pointer */
  if (cssm_IsBadReadPtr (CertTemplate, NumberOfFields*sizeof(CSSM_FIELD)) ||
        cssm_IsBadReadPtr(CertTemplate[NumberOfFields-1].FieldValue.Data,
            CertTemplate[NumberOfFields-1].FieldValue.Length) ||
        cssm_IsBadReadPtr(CertTemplate[NumberOfFields-1].FieldOid.Data,
```

```
            CertTemplate[NumberOfFields-1].FieldOid.Length) )
    {
        CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_TEMPLATE);
        return NULL;
    }

    /* Allocate a new certificate structure */
    cert_ptr = UpcallTable.malloc_func(CLHandle, sizeof(CSSM_CERTIFICATE));
    if (cert_ptr == NULL)
    {
        CSSM_SetError(&my_clm_guid, CSSM_CL_MEMORY_ERROR);
        return NULL;
    }
    memset(cert_ptr, 0, sizeof(CSSM_CERTIFICATE));

    /* Loop through the CertTemplate array */
    for( i=0; i < NumberOfFields; i++ )
    {
        /* Check that this field contains a valid data pointer */
        if ( !cl_IsBadReadPtr (CertTemplate[i].FieldValue.Data,
                           CertTemplate[i].FieldValue.Length))
        {
            /* If so, copy the data into the certificate structure */
            /* Add CL module-specific code here */
        }
        else
        {
            CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_FIELD_POINTER);
            /* Free the certificate structure */
            return NULL;
        }
    }

    /* Add internal, CL-generated certificate information */
    /* Add CL module-specific code here */

    /* If there are signatures on this cert, delete them */
    /* A newly created cert is assumed to be unsigned */
    /* Add CL module-specific code here */

    /* Pack the new certificate */
    /* The pack routine will allocate memory for the new cert using the
       application's memory allocation routines */
    packed_cert_ptr = cl_PackCertificate(cert_ptr);

    /* Cleanup */
        /* Free the certificate structure */

    /* Return the packed certificate */
    return packed_cert_ptr;
};
```

CRL operations examples

This section contains sample implementations of Certificate Revocation List (CRL) functions in the CL.

CL_CrlAddCert

```
/*-----------------------------------------------------------------------
* Name: CL_CrlAddCert
*
* Description:
* This function revokes the input certificate by adding a record
* representing the certificate to the CRL. It uses the revoker's certificate
* to sign the new record in the CRL. The reason for revoking the certificate
* may also be stored in the revocation record.
*
* Parameters:
* CLHandle (input)     : Handle to the CL module
* CCHandle (input)     : Handle to the cryptographic context
* 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
* RevokeReason (input) : The reason for revoking the certificate
* OldCrl (input)        : A pointer to the CSSM_DATA structure containing the
```

```
*                          CRL to which the newly revoked certificate will be
*                          added
*
* Return value:
* The updated CRL
*
* Error codes:
* CSSM_CL_INVALID_CL_HANDLE
* CSSM_CL_INVALID_CC_HANDLE
* CSSM_CL_INVALID_CERTIFICATE_PTR
* CSSM_CL_INVALID_CRL
* CSSM_CL_MEMORY_ERROR
* CSSM_CL_CRL_ADD_CERT_FAIL
*-------------------------------------------------------------------------*/
CSSM_DATA_PTR   CSSMAPI CL_CrlAddCert    (CSSM_CL_HANDLE CLHandle,
                           CSSM_CC_HANDLE CCHandle,
                           const CSSM_DATA_PTR Cert,
                           const CSSM_DATA_PTR RevokerCert,
                          CSSM_REVOKE_REASON RevokeReason,
                           const CSSM_DATA_PTR OldCrl)
{
   CSSM_REVOCATION_LIST_PTR new_crl_ptr = NULL;
   CSSM_DATA_PTR new_crl_data_ptr = NULL;
   CSSM_DATA_PTR sign_data_ptr = NULL;
   CSSM_REVOKED_CERT_PTR new_revoked_cert_ptr = NULL;
   CSSM_REVOKED_CERT_PTR temp_revoked_cert_ptr = NULL;
   CSSM_REVOKED_CERT_PTR prev_revoked_cert_ptr = NULL;

   CSSM_CERTIFICATE_PTR revoker_cert_ptr = NULL;
   CSSM_CERTIFICATE_PTR cert_ptr = NULL;
   uint32 signature_size;
   CSSM_DATA_PTR signature_data_ptr = NULL;
   CSSM_CONTEXT_PTR context_ptr = NULL;
   CSSM_RETURN ret;

   /* Check inputs */
   if(CLHandle == 0)
   {
      CSSM_SetError(&my_clm_guid,CSSM_CL_INVALID_CL_HANDLE);
      return NULL;
   }
   if(CCHandle == 0)
   {
      CSSM_SetError(&my_clm_guid,CSSM_CL_INVALID_CC_HANDLE);
      return NULL;
   }
   if(Cert == NULL)
   {
      CSSM_SetError(&my_clm_guid,CSSM_CL_INVALID_CERT_POINTER);
      return NULL;
   }
   if(Cert != NULL && cssm_IsBadReadPtr(Cert, sizeof(CSSM_DATA)) )
   {
      CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_DATA_POINTER);
      return NULL;
   }
   if(Cert->Length != 0 && cssm_IsBadReadPtr(Cert->Data,Cert->Length))
   {
      CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_CERT_POINTER);
      return NULL;
   }

   if(RevokerCert == NULL)
   {
      CSSM_SetError(&my_clm_guid,CSSM_CL_INVALID_REVOKER_CERT_PTR);
      return NULL;
   }
   if(RevokerCert->Length != 0 && cssm_IsBadReadPtr(RevokerCert->Data,RevokerCert->Length))
   {
      CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_REVOKER_CERT_PTR);
      return NULL;
   }
   if(OldCrl == NULL)
   {
      CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_CRL_PTR);
      return NULL;
   }
   if(cssm_IsBadReadPtr(OldCrl, sizeof(CSSM_DATA)))
   {
   CSSM_SetError(&my_clm_guid, CSSM_CL_INVALID_CRL_PTR);
```

```
   return NULL;
   }
   if(OldCrl->Length != 0 && !cssm_IsBadReadPtr(OldCrl->Data, OldCrl->Length))
   {
   /* Unpack the CRL */
   new_crl_ptr = cl_UnPackCrl(CLHandle,&MemoryFunctions,OldCrl);
   if(new_crl_ptr == NULL)
   {
      CSSM_SetError(&my_clm_guid, CSSM_CL_MEMORY_ERROR);
      return NULL;
   }

   /* remove the crl signature, if necessary */
   /* unpack the revoker's certificate */
   revoker_cert_ptr =
   cl_UnpackCertificate(CLHandle,&MemoryFunctions,RevokerCert);
   if(revoker_cert_ptr == NULL)
   {
      /* Cleanup */
         CSSM_SetError(&my_clm_guid, CSSM_CL_MEMORY_ERROR);
         return NULL;
      }
      /* unpack the certificate to be revoked */
      cert_ptr = cl_UnpackCertificate(CLHandle,&MemoryFunctions,Cert);;
      if(cert_ptr == NULL)
      {
      /* Cleanup */
         CSSM_SetError(&my_clm_guid, CSSM_CL_MEMORY_ERROR);
         return NULL;
      }

      /* Create the revoked certificate structure to be placed in the CRL */
      /* Add any revocation record specific information,
          such as the time of revocation and the revocation reason */
      /* Sign the revoked certificate structure using the revoker's certificate */
      }

   /* Add the new revocation record to the CRL */

   /* Pack the new CRL */
   new_crl_data_ptr = cl_PackCrl(CLHandle,&MemoryFunctions,new_crl_ptr);

   /* Cleanup & Return */
   return new_crl_data_ptr;
}
```

Certificate library extensibilty functions example

In this example, the pack and unpack routines that are used internally to the CL module are exposed for use by applications through the passthrough mechanism.

```
typedef enum cl_custom_function_id {
    CL_CUSTOMID_PACK_CERTIFICATE&tab;= 0,
    CL_CUSTOMID_UNPACK_CERTIFICATE&tab;= 1,
} CL_CUSTOM_FUNCTION_ID;

/*--------------------------------------------------------------------------
 * Name: CL_PassThrough
 *
 * Description:
 * This function allows applications to call OCSF CL module-specific operations.
 * The OCSF CL module-specific operations include:
 *    cl_PackCertificate
 *    cl_UnpackCertificate
 *
 * Parameters:
 * CCHandle (input)       : Handle identifying a Cryptographic Context which
 *                          may be used by the passthrough function
 * PassThroughId (input)  : An identifier assigned by the OCSF CL module
 *                          to indicate the exported function to perform.
 * InputParams (input)    : Parameters to be interpreted in a
 *                          function-specific manner by the OCSF CL module.
 *
 * Return value:
 * Output from the passthrough function.
 * The output data must be interpreted by the calling application
 * based on externally available information.
 *
```

```
 * Error codes:
 * CSSM_CL_INVALID_CL_HANDLE
 * CSSM_CL_INVALID_CC_HANDLE
 * CSSM_CL_INVALID_DATA_POINTER
 * CSSM_CL_UNSUPPORTED_OPERATION
 * CSSM_CL_PASS_THROUGH_FAIL
 *--------------------------------------------------------------------------*/
CSSM_DATA_PTR CSSMAPI CL_PassThrough   (CSSM_CL_HANDLE CLHandle,
                                        CSSM_CC_HANDLE CCHandle,
                                        uint32 PassThroughId,
                                        const CSSM_DATA_PTR InputParams)
{
    /* Initializations */
    /* Check inputs */
        /* Check that this is a recognized PassThroughId */
    /* Call the requested function */
    switch ( PassThroughId ) {
 case CL_CUSTOMID_PACK_CERTIFICATE:&tab;
return cl_PackCertificate( InputParams );
 case CL_CUSTOMID_UNPACK_CERTIFICATE:
return cl_UnpackCertificate( InputParams );
 default:


       CSSM_SetError(&my_clm_guid, CSSM_CL_UNSUPPORTED_OPERATION);
       return NULL;
```

Certificate library OCSF errors

This section defines the error code range that is defined by OCSF for use by all Certificate Libraries (CLs) in describing common error conditions. A CL may also define and return vendor-specific error codes. The error codes defined by OCSF are considered to be comprehensive and few if any vendor-specific codes should be required. Applications must consult vendor-supplied documentation for the specification and description of any error codes defined outside of this specification.

All CL service provider interface (SPI) 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 - OCSF 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. 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, including:

- Errors defined by OCSF 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 OCSF, those errors have a set of predefined numeric values that are reserved by OCSF, 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 12 lists the range of error numbers defined by OCSF for CL modules and those available for use in individual Certificate Library

(CL) modules. See the *z/OS Open Cryptographic Services Facility Application Programming* for a complete listing of the error numbers and their descriptions.

Table 12. Certificate Library Module Error Numbers

Error Number Range	**Description**
3000 – 3999	CL errors defined by OCSF
4000 – 4999	CL errors reserved for individual CL modules

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.

Chapter 5. Data storage library interface

A module with Data Storage Library (DL) services provides access to persistent data stores of certificates, Certificate Revocation Lists (CRLs), keys, policies, and other security-related objects. Stable storage can be provided by a:

- Commercially available database management system (DBMS) product
- Directory service
- Custom hardware-based storage device
- Native file system.

The implementation of DL operations should be semantically free. For example, a DL operation that inserts a trusted X.509 certificate into a data store should not be responsible for verifying the trust on that certificate. The semantic interpretation of security objects should be implemented in Trust Policy (TP) services, layered services, and applications.

The DL provides access to persistent stores of security-related objects by translating calls from the Data Storage Library Interface (DLI) into the native interface of the data store. The native interface of the data store may be that of a DBMS package, a directory service, a custom storage device, or a traditional local or remote file system. Applications are able to obtain information about the available DL services by using the CSSM_GetModuleInfo function to query the OCSF registry. The information about the DL service includes the following:

- Vendor information - Information about the module vendor, a text description of the DL and the module version number.
- Types of supported data stores - The module may support one or more types of persistent data stores as separate subservices. For each type of data store, the DL provides information on the supported query operators and optionally provides specific information on the accessible data stores.

The DL may choose to provide information about the data stores that it has access to. Applications can obtain information about these data stores by using the CSSM_GetModuleInfo function call. The information about the data store includes the following:

- Types of persistent security objects - The types of security objects that may be stored include certificates, CRLs, keys, policy objects, and generic data objects. A single data store can contain a single object type in one format, a single object type in multiple formats, or multiple object types.
- Attributes of persistent security objects - The stored security object may have attributes which must be included by the calling application on data insertion, and which are returned by the DL on data retrieval.
- Data store indexes - These indexes are high-performance query paths constructed as part of data store creation and maintained by the data store.
- Secure access mechanisms - A data store may restrict a user's ability to perform certain actions on the data store or on the data store's contents. This structure exposes the mechanism required to authenticate to the data store.
- Record integrity capabilities - Some data stores will insure the integrity of the data store's contents. To insure the integrity of the data store's contents, the data store is expected to sign and verify each record.
- Data store location - The persistent repository can be local or remote.

To build indexes or to satisfy an application's request for record retrieval, the data store may need to parse the stored security objects. If the application has invoked CSSM_DL_DbSetRecordParsingFunctions for a given security object type, those functions will be used to parse that security object as the need arises. If the application has not explicitly set record-parsing functions, the default service provider modules set by the data store creator will be used for parsing.

Secured access to the data store and to the data store's contents may be enforced by the DL, the data store, or both. The partitioning of authentication responsibility is exposed via the DL and data store authentication mechanisms.

Data stores may be added to a DL in one of three ways:

- Using DL_DbCreate - This creates and opens a new, empty data store with the specified schema.
- Using DL_DbImport with information and data - If the specified data store does not exist, a new data store is created with the specified schema and the exported data records.
- Using DL_DbImport with information only - In this case, the data store's native format is the same as that managed by the DL service. Importing its information makes it accessible via this DL service.

In all cases, it is the responsibility of the DL service to update the OCSF registry with information about the new data store. This can be accomplished by making use of the CSSM_GetModuleInfo and CSSM_SetModuleInfo functions.

Categories of operations

The DL service provider interface (SPI) defines four categories of operations:

- DL operations
- Data store operations
- Data record operations
- Extensibility operations.

DL operations are used to control access to the DL library. They include:

- Authentication to the DL Module - A user may be required to present valid credentials to the DL prior to accessing any of the data stores embedded in the DL module. The DL module will be responsible for insuring that the access privileges of the user are not exceeded.

The data store functions operate on a data store as a single unit. These operations include:

- Opening and closing data stores - A DL service manages the mapping of logical data store names to the storage mechanisms it uses to provide persistence. The caller uses logical names to reference persistent data stores. The open operation prepares an existing data store for future access by the caller. The close operation terminates current access to the data store by the caller.
- Creating and deleting data stores - A DL creates a new, empty data store and opens it for future access by the caller. An existing data store may be deleted. Deletion discards all data contained in the data store.
- Importing and exporting data stores - Occasionally a data store must be moved from one system to another, or a DL service may need to provide access to an existing data store. The import and export operations may be used in

conjunction to support the transfer of an entire data store. The export operation prepares a snapshot of a data store. (Export does not delete the data store it snapshots.)

- The import operation accepts a snapshot (generated by the export operation) and includes it in a new or existing data store managed by a DL. Alternately, the import operation may be used independently to register an existing data store with a DL.

The data record operations operate on a single record of a data store. They include:

- Adding new data objects - A DL adds a persistent copy of data object to an open data store. This operation may or may not include the creation of index entries. The mechanisms used to store and retrieve persistent data objects are private to the implementation of a DL module.
- Deleting data objects - A DL removes single data object from the data store.
- Retrieving data objects - A DL provides a search mechanism for selectively retrieving a copy of persistent security objects. Selection is based on a selection criterion.

Data store extensibility operations include:

Pass through for unique, module-specific operations - A passthrough function is included in the DLI to allow data store libraries to expose additional services beyond what is currently defined in the OCSF API. OCSF passes an operation identifier and input parameters from the application to the appropriate DL. Within the DL_PassThrough function in the DL, the input parameters are interpreted and the appropriate operation performed. The DL developer is responsible for making known to the application the identity and parameters of the supported passthrough operations.

Data storage library data structures

This section describes the data structures that may be passed to or returned from a DL function. Applications use these data structures to prepare and then pass input parameters into OCSF API function calls, which are passed without modification to the appropriate DL. The DL is responsible for interpreting them and returning the appropriate data structure to the calling application via OCSF. These data structures are defined in the header file, cssmtype.h, which is distributed with OCSF.

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.

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 OCSF. DL modules use this structure to hold persistent security-related objects.

```
typedef struct cssm_data {
    uint32 Length;
    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

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.

PriviledgedMode

A Boolean indicating that the user requests privileged operations.

Asynchronous

A Boolean indicating that the user requests asynchronous access.

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.

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.

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;
```

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.

```
#define CSSM_DB_CERT_USE_ROOT    0x00000001 /* a self-signed root cert */
#define CSSM_DB_CERT_USE_TRUSTED 0x00000002 /* re-issued locally */
#define CSSM_DB_CERT_USE_SYSTEM  0x00000004 /* contains CSSM system cert */
#define CSSM_DB_CERT_USE_OWNER   0x00000008 /* private key, owned by the system's user
*/ #define CSSM_DB_CERT_USE_REVOKED &tab;0x00000010 /* revoked cert - used w\ CRL APIs */
#define CSSM_DB_CERT_SIGNING     0x00000011 /* use cert for signing only */
#define CSSM_DB_CERT_PRIVACY     0x00000012 /* use cert for encryption only */
```

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;
```

CSSM_DB_HANDLE

A unique identifier for an open data store.&tab;

```
typedef uint32 CSSM_DB_HANDLE    /* data store Handle */
```

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 OCSF 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.

CSSM_DB_INDEX_TYPE

This enumerated list defines two types of indexes: indexes with unique values (i.e., primary database keys) and indexes with non-unique 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;
```

CSSM_DB_INDEXED_DATA_LOCATION

This enumerated list defines where within a record the indexed data values reside. Indexes can be constructed on atributes or on fields wihin 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.

```
typdef enum cssm_db_indexed_data_location {
   CSSM_DB_INDEX_ON_UNKOWN = 0
   CSSM_DB_INDEX_ON_ATTRIBUTE = 1
   CSSM_DB_INDEX_ON_RECORD = 2
} CSSM_DB_INDEXED_DATA_LOCATION
```

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 pairs (record-type, GUID) 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
&tab;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 Universal Resource Locator (URL), a file system path name, a remote directory service name, etc.

Reserved
Reserved for future use

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;
```

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_RECORDTYPE RecordType;
    CSSM_GUID Module;
} 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.

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

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 information (schema) for each of the attributes.

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.

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.

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;
```

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 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

A module-specific identifier which will allow the DL to locate this record.

CSSM_DL_DB_HANDLE

This data structure holds a pair of handles, one for a DL and another for a data store 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*.

CSSM_DL_DB_LIST

This data structure defines a list of handle pairs (DL handle, data store handle).

```
typedef struct cssm_dl_db_list {
    uint32 NumHandles;
    CSSM_DL_DB_HANDLE_PTR DLDBHandle;
} CSSM_DL_DB_LIST, *CSSM_DL_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.

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;
```

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;
```

CSSM_DL_HANDLE

A unique identifier for an attached module that provides DL services.

```
typedef uint32 CSSM_DL_HANDLE/* Data Storage Library Handle */
```

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;
```

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;
```

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 data storage device.

```
typedef struct cssm_dl_pkcs11_attributes {
    uint32 DeviceAccessFlags;
} *CSSM_DL_PKCS11_ATTRIBUTES;
```

Definitions:

DeviceAccessFlags
: Specifies the access modes applicable for accessing persistent objects in a data store.

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 OCSF registry when the DL module is installed with OCSF. 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&tab;
```

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 &tab;authentication mechanism is distinct from the authentication mechanism (specified in a cssm_dbInfo structure) required to access a specific data store.

NumberOfRelOperatorsTypes
: 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 operators 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.

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;
```

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.

CSSM_NAME_LIST

```
typedef struct cssm_name_list {
    uint32 NumStrings;
    char** String;
} CSSM_NAME_LIST, *CSSM_NAME_LIST_PTR;
```

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 OCSF

format. When CSSM_QUERY_RETURN_DATA is 0, the information is returned in raw format (a format native to the individual module, BSAFE, or PKCS11).

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.

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.

Data storage operations

This section describes the function prototypes and error codes defined for the data source operations in the DLI. The functions are exposed to OCSF through a function table, so the function names may vary at the discretion of the DL developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications.

DL_Authenticate

Purpose

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 credential applies 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 credential is to applied to a specific data store, the handle for that data store must be specified in the *DLDBHandle* pair.

Format

```
CSSM_RETURN DL_Authenticate (const CSSM_DL_DB_HANDLE DLDBHandle,
                             const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
                             const CSSM_USER_AUTHENTICATION_PTR UserAuthentication)
```

Parameters

Input

DLDBHandle
: 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
: An indicator of the requested access mode for the data store or DL module in general.

UserAuthentication
: 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.

DL_DbClose

Purpose

This function closes an open data store.

Format

```
CSSM_RETURN DL_DbClose (CSSM_DL_DB_HANDLE DLBHandle)
```

Parameters

DLDBHandle
: A handle structure containing the DL handle for the attached DL module and the database (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.

Related information
DL_DbOpen

DL_DbCreate

Purpose
This function creates a new, empty data store with the specified logical name.

Format
```
CSSM_DB_HANDLE 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)
```

Parameters
Input

DLHandle
The handle that describes the DL module to be used to perform this function.

DbName
The general, external name for the new data store.

DBInfo A pointer to a structure describing the format/schema of each record type that will be stored in the new data store.

AccessRequest
An indicator of the requested access mode for the data store, such as read-only or read/write.

Input/optional

UserAuthentication
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
A pointer to a module-specific set of parameters required to open the data store.

Return value
Returns the CSSM_DB_HANDLE of the newly created data store. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
DL_DbOpen
DL_DbClose
DL_DbDelete

DL_DbDelete

Purpose
This function deletes all records from the specified data store and removes all state information associated with that data store.

Format

```
CSSM_RETURN  DL_DbDelete (CSSM_DL_HANDLE DLHandle,
                          const char *DbName,
                          const CSSM_USER_AUTHENTICATION_PTR UserAuthentication)
```

Parameters

Input

DLHandle
The handle that describes the DL module to be used to perform this function.

DbName
A pointer to the string containing the logical name of the data store.

Input/optional

UserAuthentication
The caller's credential 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.

Related information

DL_DbCreate
DL_DbClose
DL_DbOpen

DL_DbExport

Purpose

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 DL_DbImport function. The DL module may require additional user authentication to determine authorization to snapshot a copy of an existing data store.

Format

```
CSSM_RETURN  DL_DbExport  (CSSM_DL_HANDLE DLHandle,
                           const char *DbDestinationName,
                           const char *DbSourceName,
                           const CSSM_BOOL InfoOnly,
                           const CSSM_USER_AUTHENTICATION_PTR UserAuthentication)
```

Parameters

Input

DLHandle
The handle that describes the DL module to be used to perform this function.

DbSourceName
The name of the data store from which the records are to be exported.

DbDestinationName
The name of the destination data container which will contain a copy of the source data store's records.

InfoOnly
A Boolean value indicating what to export. If CSSM_TRUE, export only the DBInfo that describes the data store. If CSSM_FALSE, export both the DBInfo and all of the records in the specified data store.

Input/optional

UserAuthentication
The caller's credential as required for authorization to snapshot/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.

Related information
DL_DbImport

DL_GetDbNameFromHandle

Purpose
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.

Format
```
char *  DL_GetDbNameFromHandle  (CSSM_DL_DB_HANDLE DLDBHandle)
```

Parameters
Input

DLDBHandle
The handle pair that describes the DL module used to perform this function and the data store to which access is being requested.

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.

DL_DbGetRecordParsingFunctions

Purpose
This function gets the records parsing function table, that operates on records of the specified type, in the specified data store. Three record-parsing functions can be returned in the table. The functions can be implemented to parse multiple record types. In this case, multiple calls to 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.

Format
```
CSSM_DB_RECORD_PARSING_FNTABLE_PTR DL_DbGetRecordParsingFunctions (CSSM_DL_HANDLE DLHandle,
                                                                   const char* DbName,
                                                                   CSSM_DB_RECORDTYPE RecordType)
```

Parameters

DLHandle
The handle that describes the DL module to be used to perform this function.

DbName
The name of the data store with which the parsing functions are associated.

RecordType
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 CSSM_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.

Related information

DL_SetRecordParsingFunctions

DL_DbImport

Purpose

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 DL_DbExport.

The *DbDestinationName* 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. The required form is documented in the capabilities and feature descriptions for this module. The resulting data store is not opened as a result of this operation.

Format

```
CSSM_RETURN DL_DbImport (CSSM_DL_HANDLE DLHandle,
                         const char *DbDestinationName,
                         const char *DbSourceName,
                         const CSSM_DBINFO_PTR DBInfo,
                         const CSSM_BOOL InfoOnly,
                         const CSSM_USER_AUTHENTICATION_PTR UserAuthentication)
```

Parameters

Input

DLHandle
The handle that describes the DL module to be used to perform this function.

DbDestinationName
The name of the destination data store in which to insert the records.

DbSourceName
The name of the data source from which to obtain the records that are added to the data store.

InfoOnly
A Boolean value indicating what to import. If CSSM_TRUE, import only the DBInfo that describes the a data store. If CSSM_FALSE, import both the DBInfo and all of the records exported from a data store.

Input/optional

DBInfo 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.

UserAuthentication
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. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
DL_DbExport

DL_DbOpen

Purpose
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. In addition, additional open parameters may be required to open a given data store and are supplied in the *OpenParameters*.

Format
```
CSSM_DB_HANDLE 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)
```

Parameters
Input

DLHandle
The handle that describes the DL module to be used to perform this function

DbName
A pointer to the string containing the logical name of the data store.

AccessRequest
An indicator of the requested access mode for the data store, such as read-only or read/write.

Input/ptional

UserAuthentication
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
A pointer to a module-specific set of parameters required to open the data store.

Return value
Returns the CSSM_DB_HANDLE of the opened data store. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

Related information
DL_DbClose

DL_DbSetRecordParsingFunctions

Purpose
This function sets the records 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 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.

Format
```
CSSM_RETURN DL_DbSetRecordParsingFunctions (CSSM_DL_HANDLE DLHandle,
                                            const char* DbName,
                                            CSSM_DB_RECORDTYPE RecordType,
                                            const CSSM_DB_RECORD_PARSING_FNTABLE_PTR FunctionTable)
```

Parameters
Input

DLHandle
The handle that describes the DL module to be used to perform this function.

DbName
The name of the data store with which to associate the parsing functions.

RecordType
One of the record types parsed by the functions specified in the function table.

FunctionTable
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.

Related information
DL_GetRecordParsingFunctions

Data record operations

This section describes the function prototypes and error codes defined for the data record operations in the DLI. The functions are exposed to OCSF through a function table, so the function names may vary at the discretion of the DL developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications.

DL_DataAbortQuery

Purpose
This functions 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.

Format
```
CSSM_RETURN DL_DataAbortQuery (CSSM_DL_DB_HANDLE DLDBHandle, CSSM_HANDLE ResultsHandle)
```

Parameters
Input

DLDBHandle
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
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.

Related information
DL_DataGetFirst
DL_DataGetNext

DL_DataDelete

Purpose
This function removes from the specified data store the data record specified by the unique record identifier.

Format
```
CSSM_RETURN DL_DataDelete (CSSM_DL__DB_HANDLE DLDBHandle,
                           CSSM_DB_RECORDTYPE RecordType,
                           const CSSM_DB_UNIQUE_RECORD_PTR UniqueRecordIdentifier)
```

Parameters
Input

DLDBHandle
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.

UniqueRecordIdentifier
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 cannot be used in future references.

Input/optional

RecordType
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

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.

Related information
DL_DataInsert

DL_DataGetFirst

Purpose
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, satisfying 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 satisfying the query. 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.

Format
```
CSSM_DB_UNIQUE_RECORD_PTR  DL_DataGetFirst (CSSM_DL_DB_HANDLE DLDBHandle,
                                            const CSSM_QUERY_PTR Query,
                                            CSSM_HANDLE_PTR  ResultsHandle,
                                            CSSM_BOOL  *EndOfDataStore,
                                            CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
                                            CSSM_DATA_PTR Data)
```

Parameters
Input

DLDBHandle
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.

Input/optional

Query 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.

Output

ResultsHandle
This handle should be used to retrieve subsequent records that satisfied this query.

EndOfDataStore
A flag indicating whether a record satisfying this query was available to be retrieved in the current operation. If CSSM_FALSE, then a record was available and was retrieved unless an error condition occurred. If CSSM_TRUE, then all records satisfying the query have been previously retrieved and no record has been returned by this operation.

Attributes
A list of attributes values (and corresponding meta-information) from the retrieved record.

Data 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.

Related information

DL_DataGetNext
DL_DataAbortQuery

DL_DataGetNext

Purpose

This function returns the next data record referenced by the *ResultsHandle*. The *ResultsHandle* parameter references a set of records selected by an invocation of the 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.

Format

```
CSSM_DB_UNIQUE_RECORD_PTR DL_DataGetNext (CSSM_DL_DB_HANDLE DLDBHandle,
                                          CSSM_HANDLE ResultsHandle,
                                          CSSM_BOOL *EndOfDataStore,
                                          CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
                                          CSSM_DATA_PTR Data)
```

Parameters

Input

DLDBHandle
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.

Output

ResultsHandle
The handle identifying a set of records retrieved by a query executed by the DL_DataGetFirst function.

EndOfDataStore
A flag indicating whether a record satisfying this query was available to be retrieved in the current operation. If CSSM_FALSE, then a record was available and was retrieved unless an error condition occurred. If CSSM_TRUE, then all records satisfying the query have been previously retrieved and no record has been returned by this operation.

Attributes
A list of attributes values (and corresponding meta-information) from the retrieved record

Data 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.

Related information
DL_DataGetFirst
DL_DataAbortQuery

DL_DataInsert

Purpose
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* parameters. 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* parameter is an opaque object to be stored in the new data record.

Format
```
CSSM_DB_UNIQUE_RECORD_PTR DL_DataInsert (CSSM_DL_DB_HANDLE DLDBHandle,
                                         const CSSM_DB_RECORDTYPE RecordType,
                                         const CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
                                         const CSSM_DATA_PTR Data)
```

Parameters
Input

DLDBHandle
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
Indicates the type of data record being added to the data store.

Input/optional

Attributes
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 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.

Related information
DL_DataDelete

DL_FreeUniqueRecord

Purpose
This function frees the memory associated with the data store unique record structure.

Format
```
CSSM_RETURN DL_FreeUniqueRecord (CSSM_DL_DB_HANDLE DLDBHandle, CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord)
```

Parameters
Input

DLDBHandle
The handle pair that describes the DL module to be used to perform this function.

UniqueRecord
The pointer to the memory that describes the data store unique record structure.

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.

Related information
DL_DataInsert
DL_DataGetFirst
DL_DataGetNext

Data storage library extensibility functions

The DL_PassThrough function is provided to allow DL developers to extend the certificate and CRL format-specific storage functionality of the OCSF API. Because it is exposed to OCSF as only a function pointer, its name internal to the DL can be assigned at the discretion of the DL module developer. However, its parameter list and return value must match. The error codes listed in this section are the generic codes all data storage libraries may use to describe common error conditions.

DL_PassThrough

Purpose

This function allows applications to call additional module-specific operations that have been exported by the DL. Such operations may include queries or services specific to the domain represented by the DL module.

Format

```
void * DL_PassThrough (CSSM_DL_DB_HANDLE DLDBHandle, uint32 PassThroughId, const void *InputParams)
```

Parameters

Input

DLDBHandle

The handle pair that describes the DL module to be used to perform this function and the open data store upon which the function is to be performed.

PassThroughId

An identifier assigned by a DL module to indicate the exported function to be performed.

InputParams

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 void pointers.

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.

Data storage library Attach/Detach example

The DL module is responsible for performing certain operations when OCSF attaches to and detaches from it. DL modules use **_init** in conjunction with the DLLMain routine to perform those operations, as shown in the following example:

```
_init
   BOOL_init( )
   {
     BOOL rc;
     rc =  DllMain(NULL, DLL_PROCESS_ATTACH, NULL);
     return (rc);
   }
```

DLLMain

```
#include<cssm.h>
CSSM_GUID dl_guid =
{ 0x5fc43dc1, 0x732, 0x11d0, { 0xbb, 0x14, 0x0, 0xaa, 0x0, 0x36, 0x67, 0x2d } };
CSSM_FUNCTIONTABLE FunctionTable;
CSSM_SPI_FUNC_TBL_PTR UpcallTable;

BOOL  DllMain ( HANDLE hInstance, DWORD dwReason, LPVOID lpReserved)
{
     switch (dwReason)
     {
     case DLL_PROCESS_ATTACH:
        {

        /* Fill in FunctionTable with function pointers */
        FunctionTable.Authenticate = DL_Authenticate;
        FunctionTable.DbOpen = DL_DbOpen;
        FunctionTable.DbClose = DL_DbClose;
        FunctionTable.DbCreate = DL_DbCreate;
        FunctionTable.DbDelete = DL_DbDelete;
        FunctionTable.DbImport = DL_DbImport;
        FunctionTable.DbExport = DL_DbExport;
        FunctionTable.DbSetRecordParsingFunctions =
           DL_DbSetRecordParsingFunctions;
        FunctionTable.DbGetRecordParsingFunctions =
           DL_DbGetRecordParsingFunctions;
        FunctionTable.GetDbNameFromHandle = DL_GetDbNameFromHandle;
FunctionTable.DataInsert = DL_DataInsert;
FunctionTable.DataDelete = DL_DataDelete;
FunctionTable.DataGetFirst = DL_DataGetFirst;
FunctionTable.DataGetNext = DL_DataGetNext;
FunctionTable.DataAbortQuery = DL_DataAbortQuery;
FunctionTable.FreeUniqueRecord = DL_FreeUniqueRecord;
FunctionTable.PassThrough = DL_PassThrough;

* Call CSSM_RegisterServices to register the FunctionTable */
/* with CSSM and to receive the application's memory upcall table */
if (CSSM_RegisterServices (&dl_guid, FunctionTable, &UpcallTable) != CSSM_OK)
                return FALSE;

 /* Make the upcall table available to all functions in this library */

 break;
   }
case DLL_THREAD_ATTACH:
   break;
case DLL_THREAD_DETACH:
   break;
case DLL_PROCESS_DETACH:
   if (CSSM_DeregisterServices (&dl_guid) != CSSM_OK)
      return FALSE;
   break;
   }
return TRUE;
}
```

Data store operations example

This section contains a template for the DL_DbOpen function.

```
/*---------------------------------------------------------------------------
 * Name: DL_DbOpen
 *
 * Description:
 * This function opens a Data store and returns a handle back to the
 * caller which should be used for further access to the data store.
 *
 * Parameters:
 * DLHandle(input)        : Handle identifying the DL module.
 * DbName                 : String containing the logical Data store name.
 * AccessRequest          : Requested access mode for the data store
 * UserAuthentication     : Caller's credentials
 * OpenParameters         : Module-specific parameters
 *
 * Return value:
 * Handle to the Opened Data store.
 * If NULL, use CSSM_GetError to get the following return codes
```

```
*
* Error codes:
* CSSM_DL_INVALID_DL_HANDLE
* CSSM_DL_DATASTORE_NOT_EXISTS
* CSSM_DL_INVALID_AUTHENTICATION
* CSSM_DL_MEMORY_ERROR
* CSSM_DL_DB_OPEN_FAIL
*-------------------------------------------------------------------------*/
              CSSM_DB_HANDLE  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)
{
   if(DLHandle == NULL)
   {
      CSSM_SetError(&dl_guid, CSSM_DL_INVALID_DL_HANDLE);
      return NULL;
   }
   if(DbName == NULL)
   {
      CSSM_SetError(&dl_guid, CSSM_DL_INVALID_DATASTORE_NAME);
      return NULL;
   }
   if(!dl_IfDataStoreExists(DLHandle, DbName))
   {
      CSSM_SetError(&dl_guid, CSSM_DL_DATASTORE_NOT_EXISTS);
      return NULL;
   }

   /*DL specific internal implementation of DbOpen*/

   CSSM_DB_Handle Handle = dl_OpenDataStore(DbName);
              return Handle;
}
```

Data storage library OCSF errors

This section defines the error code range in OCSF that provides a consistent mechanism across all layers of OCSF for returning errors to the caller. All Data Storage Library (DL) service provider interface (SPI) 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 - OCSF 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. 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 OCSF 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 OCSF, those errors have a set of predefined numeric values that are reserved by OCSF, 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 13 lists the range of error numbers defined by

OCSF for DL modules and those available for use individual DL modules. See the *z/OS Open Cryptographic Services Facility Application Programming* book for a list of error codes and their descriptions for DL.

Table 13. Data Storage Library Module Error Numbers

Error Number Range	**Description**
5000 – 5999	DL errors defined by OCSF
6000 – 6999	DL errors reserved for individual DL modules

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.

Appendix. Accessibility

Accessible publications for this product are offered through the z/OS® Information Center, which is available at www.ibm.com/systems/z/os/zos/bkserv/.

If you experience difficulty with the accessibility of any z/OS information, please send a detailed message to mhvrcfs@us.ibm.com or to the following mailing address:

IBM® Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the z/OS Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually

exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1!

(KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

  Note:

  1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
  2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
  3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS™, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted

for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: http://www.ibm.com/software/support/systemsz/lifecycle/
- For information about currently-supported IBM hardware, contact your IBM representative.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml (http://www.ibm.com/legal/copytrade.shtml).

Glossary

This glossary defines technical terms and abbreviations used in Open Cryptographic Services Facility documentation. If you do not find the term you are looking for, refer to the index of the appropriate OCSF manual or view IBM Glossary of Computing Terms, located at:

http://www.ibm.com/ibm/terminology

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.

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. OCSF accommodates Data Encryption Standard (DES), RC2, RC4, International Data Encryption Algorithm (IDEA), and other encryption algorithms.

cryptographic service provier
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 an 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 send 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 the 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 public 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.

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.

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.

Open Cryptographic Services Facility (OCSF) Framework
Open Cryptographic Services Facility (OCSF) Framework. The Open

Cryptographic Services Facility (OCSF) framework defines four key service components:

- Cryptographic Module Manager
- Trust Policy Module Manager
- Certificate Library Module Manager
- Data Storage Library Module Manager

The OCSF binds together all the security services required by applications. In particular, it facilitates linking digital certificates to cryptographic actions and trust protocols.

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.

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 carries a PKCS #7 message, which itself is the result of cryptographic processing on other MIME body parts.

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.

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.

symmetric algorithm
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.

Index

A

C

D

E

F

G

H

I

K

M

N

O

P

R

S

T

U

Product Number: 5650-ZOS

Printed in USA

SC14-7514-00