CSSM Application Notes

Purpose

This document is designed to aid application developers who want to write applications on top of the Common Security Services Manager (CSSM). It provides explanations of concepts and methods of operation that may not be immediately obvious from reading the CSSM Application Programming Interface. Short code examples are provided, and are assumed to be written in C++, unless otherwise noted. Appendix A contains a *cookbook* for common tasks, such as creating and signing certificates.

1. Incorporating CSSM in an Application

1.1 Include CSSM header files

To incorporate CSSM in an application, include the file `CSSM.H`. It contains all of the definitions for the structures and error codes the application will use.

```
#include "cssm.h"
```

1.2 Initialize CSSM

To initialize CSSM, CSSM_Init() must be called. This should be done at application start-up time or load time of a dynamically-linked library (DLL).

As seen from the CSSM API, this function takes four parameters. The first two parameters are the Major Version and Minor Version numbers of the CSSM release. They are defined in `CSSM.H` as `CSSM_MAJOR` and `CSSM_MINOR`. The third parameter is the tricky one. It asks for a `CSSM_API_MEMORY_FUNCS_PTR`. It may seem confusing, but it is really very simple and ingenious, as described below. The fourth parameter is reserved, so pass in `NULL`.

CSSM lives in a separate process address space. When a piece of data is requested, it must allocate memory for the return object. However, since CSSM is in a different address space, it can't allocate and pass a memory pointer back to the application from CSSM's own space. The pointer would be pointing at something completely different in the application's address space.

How does CSSM get around this problem? Easy, it allocates the memory directly in the application's address space. In order to do this, the application must pass CSSM a structure containing pointers to the application's own memory allocation and deallocation routines. In typical C code, these would be `malloc`, `free`, `realloc`, `calloc`. However, CSSM's flexibility allows any memory allocation routines to be used, including custom-written routines.

Here is an example of a CSSM initialization:

```
CSSM_API_MEMORY_FUNCS fx = {malloc, free, realloc, calloc};

CSSM_RETURN result = CSSM_Init (CSSM_MAJOR, CSSM_MINOR, &fx, NULL);
if (result == CSSM_FAIL)
      return -1;              // Error - CSSM couldn't initialize
```

It is a good idea to declare the CSSM_API_MEMORY_FUNCS object globally, since add-in modules also require this structure to be passed in at attach time.

2. Attaching to Plug-in Modules

2.1 Module Types and Features

CSSM supports four different types of modules.

2.1.1 Cryptographic Services Module

This module contains algorithms for encryption, decryption, hashing, digital signatures, key generation, random number generation, and any other cryptographic services. It is commonly referred to as the CSP (Cryptographic Service Provider).

2.1.2 Certificate Library Services Module

This module provides services for creating, signing, verifying, and managing digital certificates. It is commonly referred to as the CL (Certificate Library).

2.1.3 Data Storage Library Services Module

This module provides the functionality for securely storing and retrieving digital certificates. It is commonly referred to as the DL (Database Library).

2.1.4 Trust Policy Services Module

This module provides a policy for how certificates should be validated and how trust is established. It is commonly referred to as the TP (Trust Policy).

2.2 Attaching to a Module

Attaching to a module is very similar to initializing CSSM. Each module has an *Attach* function and a *List Modules* function. The *List Modules* function takes no arguments and returns a `CSSM_LIST` of the available modules. For example, to list all of the CSP modules available on the system, use the following code can be used:

```
CSSM_LIST_PTR pCSPModuleList = CSSM_CSP_ListModules();
```

This list consists of a number of `CSSM_LIST_ITEM` structures. Each item contains a `CSSM_GUID` (Globally Unique ID) and a string name.

When the application has confirmed that a particular module is present, it calls the module's *Attach* function. This function takes five parameters. The first is a `CSSM_GUID`, which can be obtained from the `CSSM_LIST` of modules. The second and third parameters are the module major and minor version numbers, which must be known at the time the application is written. The fourth parameter is a collection of pointers to memory allocation and deallocation routines, just as was used in `CSSM_Init()`. The final parameter is reserved, so pass `NULL`. If the call to *Attach* is successful, the module is dynamically loaded by CSSM and the application is given back a handle to the module. For example, with the CSP:

```
#define CSSM_CSP_MAJOR_VERSION  1
#define CSSM_CSP_MINOR_VERSION  0
CSSM_GUID CSPModuleGUID;
CSSM_CSP_HANDLE hCSP = NULL;

// Just pick the first module in the list to attach to
CSPModuleGUID = pCSPModuleList->Items[0].GUID;

// Use the same memory functions as CSSM_Init()
hCSP = CSSM_CSP_Attach( &CSPModuleGUID,
                        CSSM_CSP_MAJOR_VERSION,
                        CSSM_CSP_MINOR_VERSION,
                        &fx,
                        NULL )
```

CSSM dynamically loads the CSP module and the application receives a handle if the call is successful. The other modules have similar routines. Just substitute the CSP from these examples for DL, CL, or TP, depending on the type of module required.

2.3 Detaching From a Module

The application should keep track of all the handles created from calls to *Attach*. They should be detached before the application shuts down. Detaching from modules allows CSSM to unload modules after applications are no longer using them. Each module has a *Detach* function which takes a handle as its argument. Continuing with the CSP example, assume a valid `CSSM_CSP_HANDLE` called `hCSP` exists. To detach it, make the following call:

```
if (CSSM_CSP_Detach(hCSP) == CSSM_FAIL)
{
      // Error!  Handle it in here
}
```

The *Detach* functions for other modules work the same way.

3. Connecting to a Database

To store and retrieve certificates, the application must connect to a database. Every DL module contains a list of databases that it owns and can control. To get a list of the available databases , attach your selected DL module and call `CSSM_DL_GetDbNames()` with the DL handle as the argument. It will return a `CSSM_NAME_LIST_PTR` that points to a list of database names. The application can pick a database from the list, or use it to verify that a certain database actually exists on the system.

When the name of the database is known, a handle to it can be obtained by calling `CSSM_DL_DbOpen()`, passing in the DL handle and the database name as arguments. Database handles should be closed when finished, using `CSSM_DL_DbClose()`.

Here is an example of a routine that checks to make sure a database called “Personal Certificate Database” exists and, once found, opens the database, does some work, then closes it again:

```
// Assume CSSM_DL_HANDLE hDL is a valid DL handle
CSSM_NAME_LIST_PTR pDbList = NULL;
const char dbName = “Personal Certificate Database”;
CSSM_DB_HANDLE hDB = NULL;
int found = 0;
int i = 0;

pDbList = CSSM_DL_GetDbNames(hDL);

if (pDbList == NULL) return -1;  // Error! No database names
for (i=0; i<pDbList->NumStrings; i++)
{
      if (!strcmp(dbName, pDbList->Strings[i]))
      {
            found = 1;
            break;
      }
}

if (!found) return -1; // Didn’t find it

hDB = CSSM_DL_DbOpen(hDL, dbName);
if (hDB == NULL)
      return -1;  // Error! Couldn’t connect to the database
// Do some database work here

// Okay, now I’m ready to close the database
if (CSSM_DL_DbClose(hDL, hDB) != CSSM_OK)
      return -1;   // Error!  Couldn’t close the database
```

4. Error Checking

Most CSSM calls return a `CSSM_RETURN` value, which can be either `CSSM_OK` or `CSSM_FAIL`. A few calls return a pointer, which returns `NULL` on failure. If any call to CSSM fails, `CSSM_GetError()` can be called. It will return a `CSSM_ERROR_PTR` that contains the type of error that occurred. The error codes are defined in the file `CSSMERR.H`, which is automatically included in a project when `CSSM.H` is included. Here is an example:

```
// Assume hDL is a DL handle already in existence;
CSSM_DB_HANDLE hDB = NULL;

hDB = CSSM_DL_DbOpen(hDL, “Personal Certificate Database”);
if (hDB == NULL)
{
      CSSM_ERROR_PTR pError = NULL;
      pError = CSSM_GetError();
      if (pError)
      {
            switch (pError->error)
            {
            case CSSM_DL_INVALID_DL_HANDLE:
                  // Invalid DL handle.  Deal with it accordingly
            case CSSM_DL_MEMORY_ERROR:
                  // Memory error.  Free some memory and try again
            }
      }
}
```

5. General Purpose Data Access

5.1 What is a CSSM_DATA object?

CSSM_DATA objects are passed throughout the CSSM API. Wherever a piece of data content is needed, a CSSM_DATA is used. This data can be almost anything, which is what makes the mechanism general purpose.

The CSSM API defines the CSSM_DATA structure as follows.

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

This is simply a 4-byte length field followed by a pointer to the data. It can be used to represent any type of data, such as a string, an integer, a certificate, or a bitmap. When an application fetches certificates from a database, or when particular fields are retrieved from a certificate, they are returned as CSSM_DATA structures. When a new certificate is created, a CSSM_DATA object must be created for each value passed into CSSM.

5.2 What does a CSSM_DATA object represent?

An application developer's first question is probably, "I have this CSSM_DATA structure, but how do I know what it represents? It could be a string, an integer, a bitmap, or an array of function pointers for all I know." This is a very valid question. No type information is stored in a CSSM_DATA. It is only a length and a chunk of data.

The answer is that the type of data a CSSM_DATA object represents is not known. In many cases, this is definitely the correct behavior. For example, certificates can be passed around without regard to the internal implementation of the certificate structure.

However, in some cases the programmer needs to be able to correctly use the data stored in a CSSM_DATA structure. When reading the fields of a certificate, they are returned as CSSM_DATA structures. For example, say an application asks for the Serial Number field of a certificate. Is this serial number supposed to be a representation of a string, or an integer, or a picture of the person's badge? The answer is specific to the particular module being used. One certificate library module (CL) might represent serial numbers as strings, but another might represent them as unsigned long integers. You can deal with this problem three ways:.

1. The pass-through function the module provides. CSSM provides a method for modules to provide extra functionality beyond what the CSSM API defines. For example, the CL pass-through method is CSSM_CL_PassThrough(). The module writer can provide special pass-through functions to unpack CSSM_DATA objects as C style structures. This method is the best, if it is available, since there is no ambiguity. The next section covers pass-through functions.
2. Rely on the module writer's documentation. It is important for any module writer, especially for the CL writer, to include good documentation describing how each field is represented. The module writer may find it takes too much time and effort to implement pass-through functions for all data members, or that it seems unnecessary. In these cases, good documentation will suffice and the translation is left to the application developer.

3. Guess. This is obviously the worst method and should be used only when trying to write code for badly documented modules or for all possible future modules. Many fields can probably be safely guessed from the context. For example, when the name of the issuer for a certificate is requested, it will most likely be a string. This is not guaranteed, of course, but in many cases it will be good enough. Remember, this method is an absolute last resort; pass-through functions and documentation should be relied upon instead, if at all possible.

6. Pass-through Functions

6.1 Purpose

There is a CSSM pass-through function for each type of module. A pass-through function allows a module to provide extra features above and beyond the CSSM API. The passthrough mechanism is very generic and can support a wide variety of functions.

The module documentation is extremely important when using pass-through functions. This documentation is the only guide to determining what pass-through functions a module provides and how to use them. Often, the input parameter is some other data structure simply cast to a `CSSM_DATA_PTR` to be passed through. Likewise, a `CSSM_DATA_PTR` return value may really be a pointer to a different structure that must be cast properly. The only guide to the real data structures that are expected lies in the module documentation.

6.2 Usage

Every pass-through function has at least three parameters:

1. One or more handles. These are the handle for the particular module plus any other handles it may need to perform its operations. Look at the pass-through function for each module type in the CSSM API for details on which handles are required for each module.
2. A `uint32` that is the pass-through ID. It identifies the pass-through function you want to call. The module writer will usually give an additional header file which defines the IDs. Sometimes, however, they may defined only in the documentation.
3. A `CSSM_DATA_PTR` that points to the input parameters of the function. This can be a pointer to a `CSSM_DATA` structure, a pointer to an array of `CSSM_DATA` structures, or a pointer to just about anything simply cast to a `CSSM_DATA_PTR`. A pointer is just a pointer, and the module writer may have some pass-through functions that take pointers to other data structures that can be cast to a `CSSM_DATA_PTR` before you pass them in. Be sure to read the module's documentation very carefully to see what kind of data it expects.

One example of a pass-through is `CSSM_CL_CertPassThrough()`. As defined in the CSSM API, it takes a CL handle that represents what CL module to use, a CC handle that represents the cryptographic context, a pass-through ID, and a CSSM_DATA_PTR to the input parameters.

6.3 Limitations

Pass-through functions are limited to one input pointer and one output pointer. There is no support for inputting multiple parameters. To perform operations that require multiple pieces of data, the application may need to pack all pieces of data into one consolidated data structure that can be passed in to the pass-through function. An array could be passed in, but the length of the array would need to be predefined. Also, an application developer can't tell what pass-throughs are available without extra documentation and/or header files.

7. Using OIDs

7.1 Description

OID stands for Object Identifier or Object ID. The structure called `CSSM_OID` represents an OID. Use OIDs wherever a module needs an identifier for a particular field or value it supports. For example, an application wants to get the "City" field of a certificate. Not every certificate library or certificate format may support a "City" field. This is where OIDs come in handy. Using the CSSM function called `CSSM_CL_CertGetFirstFieldValue()`, you pass in an OID to determine which field you want to get from the certificate.

Each module will define their own OIDs. They will usually place them in an external header file. For example, the Intel CLM, Release 1.0, defines all of its OIDs in a file called `CERTOIDS.H`.

7.2 Usage

OIDs are used whenever a field must be uniquely identified, particularly when dealing with certificates. When you create a new certificate, you pass in an array of OID/Value pairs, where the CSSM_OID defines what field of the certificate the associated data member represents. OID/Value pairs are also used when doing a database search for certificates that match certain criteria. Also, if a particular field of a certificate is desired, the certificate and the OID of the wanted field are passed in, and the module will return a data member for that OID.

7.3 Stumbling Blocks

`CSSM_OID`s are defined in `CSSM.H` as being equivalent to a `CSSM_DATA`. Don't think of them this way though. Think of them as a static ID, much like you would treat a traditional integer ID. When using them, define a `CSSM_OID` on the frame, like so:

```
CSSM_OID myOID;
```

Then, just assign it a statically-defined OID from the module's header file, like this:

```
myOID = CSSMOID_INTEL_SubjectNameOID;
```

Then, you can go ahead and use myOID as an ID just like you would an integer ID. There is no memory allocation and deallocation necessary, and no extra data copying. Just the assignment.

7.4 Example of OID Usage

Let's say we have an inexperienced certificate issuer, John Smith. John has been issuing his certificates with the same serial number. We want to track these certificates down, so they can be revoked and reissued with unique numbers.

Here is a sample piece of code to retrieve and revoke all of the certificates in a database that have a serial number "1234567890" and Issuer Name "John Smith". For definitions of the various structures used, see the file `CSSM.H`.

```
// Assume the following handles are already valid:
// CSSM_DL_HANDLE hDL;
// CSSM_DB_HANDLE hDB;

#include “cssm.h”
#include “certoids.h”

CSSM_HANDLE ResultsHandle;
uint32 numMatched = 0;
CSSM_SELECTION_PREDICATE_PTR pPredicate;
char SerialNumber[] = “1234567890”;
char IssuerName[] = “John Smith”;

// Allocate a array of selection predicates.  We are looking for 2 things, a
Issuer Name and Serial Number
pPredicate = (CSSM_SELECTION_PREDICATE_PTR)
            malloc( 2 * sizeof(CSSM_SELECTION_PREDICATE));

// Both fields must be a perfect match
pPredicate[0].dbOperator = CSSM_EQUAL;
pPredicate[1].dbOperator = CSSM_EQUAL;

// Set up the serial number
pPredicate[0].Field.FieldOid = CSSMOID_SerialNumber;  // This OID is defined
in certoids.h
pPredicate[0].Field.FieldValue.Length = sizeof(SerialNumber);
pPredicate[0].Field.FieldValue.Data = (uint8*)SerialNumber;

// Set up the Subject Name
pPredicate[1].Field.FieldOid = CSSMOID_IssuerName;  // This OID is defined in
certoids.h
pPredicate[0].Field.FieldValue.Length = sizeof(IssuerName);
pPredicate[0].Field.FieldValue.Data = (uint8*)IssuerName;

// Now we can perform the database search
CSSM_DATA_PTR pData = NULL;   // pointer that will retrieve the results
CSSM_RETURN ret;

pData = CSSM_DL_CertGetFirst(hDL, hDB, pPredicate, 2,
                              CSSM_AND, &ResultsHandle, &numMatched);
if (numMatched == 0) return;

for(uint32 i=0; i < numMatched; i++)
{
      if (pData == NULL)
            return;           // Error condition!  Should not happen.

      // Revoke the certificate
      ret = CSSM_DL_CertRevoke(hDL, hDB, pData);
      if (ret == CSSM_FAIL)
      {
            // Handle the error
      }

      // free the allocated memory
      if (pData->Data) free(pData->Data);
      free(pData);

      // Try to get the next matching certificate
      pData = CSSM_DL_CertGetNext(hDL, hDB, ResultsHandle);
```

```
}
```

Appendix A
Developers' Cookbook

Here are some code fragments you can use to perform common CSSM tasks. They will help you get up and running fast, and make writing a CSSM application much less daunting.

This cookbook includes:

- Creating and deleting a certificate database
- Generating a key pair
- Creating a certificate
- Signing, verifying, and unsigning a certificate
- Inserting, retrieving, and removing certificates from a database

In the following code fragments, the handles are assumed to be defined and valid. See *Attaching to Plug-in Modules* for help on obtaining these handles.

```
CSSM_DL_HANDLE hDL;           // Data Storage Library Handle
CSSM_CL_HANDLE hCL;           // Certificate Library Handle
CSSM_CSP_HANDLE hCSP;         // Crypto Service Provider Handle
CSSM_DB_HANDLE hDB;           // Database Handle
```

Also, it is assumed that the following header files are included in the source files.

```
#include "cssm.h"             // Core CSSM functions
#include "certoids.h"         // OIDs for certificate operations.
```

Creating and Deleting a Certificate Database

```
// Going to create a new database called “Test Database” and then
// delete it.

const char *dbName = “Test Database”;
CSSM_DB_HANDLE = newDBHandle;
CSSM_ERROR_PTR pError = NULL;

newDBHandle = CSSM_DL_DbCreate( hDL, hCL, dbName );
if (newDBHandle == NULL)
{
      pError = CSSM_GetError();
      if (pError)
      {
            // Deal with the error condition;
            free(pError);
      }
      return;
}

// Perform database operations here

// Finished with the database... delete it.
CSSM_RETURN ret;

ret = CSSM_DL_DbDelete( hDL, dbName );
if (ret == CSSM_FAIL)
{
      pError = CSSM_GetError();
      if (pError)
      {
            // Deal with the error condition
            free(pError);
      }
      return;
} else {
      // Database deletion successful!
      // Don’t forget that the database handle is now invalid.
      // Set it to NULL for safety.
      newDBHandle = NULL;
}
```

Generating a Key Pair

```
// We will generate a Public/Private Key pair using the DSA algorithm
// Keep in mind, a particular CSP may support all sorts of
// algorithms and key sizes.

CSSM_CC_HANDLE KeyGenContext = NULL;

uint8             SeedBuf[] = “This is a seed, but it should be more random";
const char        *pwd = “testpassword”;
CSSM_CRYPTO_DATA  Seed;
CSSM_DATA         SeedData;
CSSM_CRYPTO_DATA  password;
CSSM_DATA         password_data;
CSSM_KEY          Key;

// Create Key Pair Generation context
SeedData.Length = sizeof(SeedBuf);
SeedData.Data = (uint8*)SeedBuf;

Seed.Param = &SeedData;
Seed.Callback = NULL;

// Initialize the password information
password_data.Length = strlen(pwd);
password_data.Data = (uint8*)pwd;
password.Callback = NULL;
password.Param = &password_data;

// Here we will create a DSA key pair generation context
// See the API document for more detailed explanation
KeyGenContext = CSSM_CSP_CreateKeyGenContext
                  (hCSP, CSSM_ALGID_DSA, &password, 512, 1808, &Seed,
                  &password_data);

if (!KeyGenContext) {
      return;
}

Key.KeyBlobLength = sizeof(CSSM_KEYBLOB);
Key.KeyBlob = (CSSM_KEYBLOB_PTR)malloc(sizeof(CSSM_KEYBLOB));

if (CSSM_FAIL == CSSM_GenerateKey( KeyGenContext, &Key )) {
      free(Key.KeyBlob);
      return;
}

// At this point, Key should have a vaild public DSA public key,
// with the private key securely stored in the CSP and
// password protected.
```

Creating a Certificate

It is not obvious from the CSSM API how to create a certificate. The basic idea is that an array of OID/Value pairs representing all the fields in the certificate is passed into CSSM. CSSM will then return a CSSM_DATA that is the newly-created certificate. In the following example, I use the Intel Certificate Library Module, Release 1.0. If another Certificate Library module is used, consult that documentation for the required OIDs and certificate fields.

```
// Utility function that turns a string into a CSSM_DATA structure
CSSM_DATA_PTR PackString ( const char* pSource )
{
      CSSM_DATA_PTR toReturn = NULL;

      if (pSource == NULL) return NULL;

      toReturn = (CSSM_DATA_PTR)malloc(sizeof(CSSM_DATA));
      toReturn->Length = strlen(pSource);
      char* pData = (char*)malloc(sizeof(char) * (strlen(pSource)));
      memcpy(pData, pSource, strlen(pSource));
      toReturn->Data = (uint8*)pData;
      return toReturn;
}
```

```
// This function will return a CSSM_DATA_PTR to the
// newly created certificate
CSSM_DATA_PTR Create ()
{
const char *m_CommonName = “USA;Mtel Corp;Los Angeles;Marketing;Smith;John”;
const char *m_IssuerName = “USA;Mtel Corp;Los Angeles;Sales;Doe;Jane”;
const char *m_SerialNumber = “111-532-593822”;

// Validity dates must be in X.509 UTC time format.  See the CL
// documentation for further details on UTC time format.  See the
// source code for Certificate Viewer for sample routines to convert
// from MM/DD/YY format to UTC time format
const char *m_StartDate = GetStartingValidityDate();
const char *m_EndDate = GetEndingValidityDate();

// The key should be one you generated earlier
const CSSM_KEY_PTR m_pKey = GetKey();

// Extensions are completely defined at the application level.  They
// simply get appended to the certificate.  You as the application
// developer must define IDs to use for your extensions.
// For this example, I’ll only use 2 extensions, a State field and a
// Zip Code field
#define CSSMEXT_ID_State “220”
#define CSSMEXT_DESC_State “State”
#define CSSMEXT_ID_Zip “221”
#define CSSMEXT_DESC_Zip “Zip Code”

const char *m_State = “CA”;
const char *m_Zip = “91885”;


// The total number of fields to pass in to CSSM will be 1 for each piece
// of data, except each extension will require 4 OID/Value pairs.
uint32 totalFields = 14;

CSSM_FIELD_PTR pFields = (CSSM_FIELD_PTR)malloc(sizeof(CSSM_FIELD)
                         * totalFields);

uint32 index = 0;
CSSM_DATA_PTR pData = NULL;
uint32 NotCritical = 0;

// Set up the Common Name Field
pFields[index].FieldOid = CSSMOID_CommonName;
pData = PackString(m_CommonName);
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Set up the Issuer Name Field
pFields[index].FieldOid = CSSMOID_IssuerName;
pData = PackString(m_IssuerName);
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Set up the Serial Number Field
pFields[index].FieldOid = CSSMOID_SerialNumber;
pData = PackString(m_SerialNumber);
```

```
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Set up the Key Field
pFields[index].FieldOid = CSSMOID_PublicKeyBlob;
pFields[index].FieldValue.Length = sizeof(CSSM_KEYBLOB);
pFields[index].FieldValue.Data = (uint8*)malloc(sizeof(CSSM_KEYBLOB));
memcpy(pFields[index].FieldValue.Data, m_pKey->KeyBlob, sizeof(CSSM_KEYBLOB));
index++;

// Set up the starting validity date field
pFields[index].FieldOid = CSSMOID_ValidityStartDate;
pData = PackString(m_StartDate);
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Set up the ending validity date field
pFields[index].FieldOid = CSSMOID_ValidityEndDate;
pData = PackString(m_EndDate);
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Create 4 fields to represent each extension

// State
// Create a field for the Extension ID
pFields[index].FieldOid = CSSMOID_CSSMCertificateExtensionId;
pData = PackString(CSSM_ID_State);
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Create a field for the Critical Flag
pFields[index].FieldOid = CSSMOID_CSSMCertificateExtensionCritical;
pFields[index].FieldValue.Length = sizeof(uint32);
pFields[index].FieldValue.Data = (uint8*)malloc(sizeof(uint32));
memcpy(pFields[index].FieldValue.Data, &NotCritical, sizeof(uint32));
index++;

// Create a field for the Description
pFields[index].FieldOid = CSSMOID_CSSMCertificateExtensionDescription;
pData = PackString(CSSM_DESC_State);
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Create a field for the Extension data
pFields[index].FieldOid = CSSMOID_CSSMCertificateExtension;
pData = PackString(m_State);
pFields[index].FieldValue = *pData;
free(pData);
index++;
```

```
// Zip Code
// Create a field for the Extension ID
pFields[index].FieldOid = CSSMOID_CSSMCertificateExtensionId;
pData = PackString(CSSM_ID_Zip);
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Create a field for the Critical Flag
pFields[index].FieldOid = CSSMOID_CSSMCertificateExtensionCritical;
pFields[index].FieldValue.Length = sizeof(uint32);
pFields[index].FieldValue.Data = (uint8*)malloc(sizeof(uint32));
memcpy(pFields[index].FieldValue.Data, &NotCritical, sizeof(uint32));
index++;

// Create a field for the Description
pFields[index].FieldOid = CSSMOID_CSSMCertificateExtensionDescription;
pData = PackString(CSSM_DESC_Zip);
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Create a field for the Extension data
pFields[index].FieldOid = CSSMOID_CSSMCertificateExtension;
pData = PackString(m_Zip);
pFields[index].FieldValue = *pData;
free(pData);
index++;

// Now that all the fields are set up, let's actually create the cert
CSSM_DATA_PTR toReturn = NULL;

toReturn = CSSM_CL_CertCreate(hCL, pFields, totalFields));

// Clean up
for (uint32 i=0; i<totalFields; i++) {
      if (pFields[i].FieldValue.Length)
            free(pFields[i].FieldValue.Data);
}

if (pFields) free(pFields);

return toReturn;
}
```

Signing, Verifying, and Unsigning a Certificate

```
// This helper function  will create a Signature Context using
// The Algorithm ID and pass phrase (for signing and unsigning).
CSSM_CC_HANDLE
CreateSignatureContext(uint32 AlgID, const char* password)
{
CSSM_CC_HANDLE hSigContext = NULL;
CSSM_CRYPTO_DATA cspData;
CSSM_DATA   paramData;
CSSM_KEY_PTR pKey = NULL;

// Set up the crypto data
cspData.Callback = NULL;

// The "cspData" is the password for the signer's private key
if (password) {
      paramData.Length = strlen(password);
      paramData.Data = (uint8*)password;
} else {
      paramData.Length = 0;
      paramData.Data = NULL;
}
cspData.Param = &paramData;

// Get the public key data from the certificate
pKey = GetKey();

// Create the signature context
hSigContext = CSSM_CSP_CreateSignatureContext(hCSP, AlgID, &cspData, pKey);

return hSigContext;
}

// Routine to sign a certificate
CSSM_RETURN
Sign ( CSSM_DATA_PTR const pSignee, CSSM_DATA_PTR const pSigner,
       const char* password )
{
      CSSM_CC_HANDLE hCC;
      CSSM_DATA_PTR pSignedCert = NULL;

      // Create a Signature Context with DSA algorithm
      hCC = CreateSignatureContext(CSSM_ALGID_DSA, password);
      if (!hCC)
            return CSSM_FAIL;

      // Sign the input certificate.  Use the default of signing all fields
      pSignedCert = CSSM_CL_CertSign( hCL, hCC, pSignee, pSigner, 0, 0);

      if (!pSignedCert)
            return CSSM_FAIL;

      // Set the signee's data to the newly signed cert and free the old data
      // NOTE: This is in memory only.  You must explicitly insert the newly
      // signed certificate into the database if you want to make the change
      // persistent.
      if (pSignee->Data) free(pSignee->Data);
```

```
	*pSignee = *pSignedCert;
	free(pSignedCert);
	return CSSM_OK;
}

// Routine to unsign a certificate by removing a signature from the cert
CSSM_RETURN
Unsign ( CSSM_DATA_PTR const pUnsignee, CSSM_DATA_PTR const pUnsigner,
	  const char* password )
{
	CSSM_CC_HANDLE hCC;
	CSSM_DATA_PTR pUnsignedCert = NULL;

	// Create a Signature Context with DSA algorithm
	hCC = CreateSignatureContext(CSSM_ALGID_DSA, password);
	if (!hCC)
		return CSSM_FAIL;

	// Unsign the input certificate. Use the default of Unsigning all fields
	pUnsignedCert = CSSM_CL_CertUnsign( hCL, hCC, pUnsignee, pUnsigner,
					    0, 0 );

	if (!pUnsignedCert)
		return CSSM_FAIL;

	// Set the unsignee's data to the newly signed cert
	// NOTE: This is in memory only.  You must explicitly insert the newly
	// signed certificate into the database if you want to make the change
	// persistent.
	if (pUnsignee->Data) free(pUnsignee->Data);
	*pUnsignee = *pUnsignedCert;
	free(pUnsignedCert);
	return CSSM_OK;
}

// Routine to verify the certificate on a signature
CSSM_RETURN
Verify ( CSSM_DATA_PTR const pCert, CSSM_DATA_PTR const pSigner )
{
	CSSM_CC_HANDLE hCC;
	CSSM_RETURN ret;

	// Create a Signature Context with DSA algorithm
	hCC = CreateSignatureContext(CSSM_ALGID_DSA, NULL);
	if (!hCC)
		return CSSM_FAIL;

	// Verify the input certificate. Use the default of Unsigning all fields
	ret = CSSM_CL_CertVerify( hCL, hCC, pCert, pSigner, 0, 0 );

	return ret;
}
```

Inserting, Deleting, and Retrieving Certificates from a Database

```
// Insert and Delete are trivial.  You just have to have a handle to
// the database and a certificate

CSSM_RETURN DbInsert (CSSM_DATA_PTR pCert)
{
      return CSSM_DL_CertInsert( hDL, hDB, pCert );
}

CSSM_RETURN DbDelete (CSSM_DATA_PTR pCert)
{
      retirm CSSM_DL_CertDelete( hDL, hDB, pCert );
}

// Retrieving certificates from a database is much more complicated.
// The fundamental concept is simple though, the rest is
// just setup code.  You have to specify your search
// criteria through an array of OID/Value pairs that represent
// the certificate field values to match on.

// The following will retrieve the last certificate it finds in the
// database that matches the search criteria.  For the search
// criteria in this example, we will use the Common Name and
// Issuer Name from the Create Certificate example

CSSM_DATA_PTR DbRetrieve()
{
const char *m_CommonName = “USA;Mtel Corp;Los Angeles;Marketing;Smith;John”;
const char *m_IssuerName = “USA;Mtel Corp;Los Angeles;Sales;Doe;Jane”;

CSSM_SELECTION_PREDICATE_PTR Filter = NULL;
uint32 FilterSize = 2;
CSSM_HANDLE ResultsHandle;
uint32 NumCerts = 0;
uint32 i = 0;
CSSM_DATA_PTR pTemp = NULL;
CSSM_DATA_PTR toReturn = NULL;

// Make a filter based on the Common Name and Issuer Name
Filter = (CSSM_SELECTION_PREDICATE_PTR)malloc( FilterSize *
          sizeof(CSSM_SELECTION_PREDICATE));

for (i = 0; i<FilterSize; i++) {
      // All fields in the filter must be EQUAL to be a match
      Filter[i].dbOperator = CSSM_EQUAL;
}
```

```
i = 0;

// Create a OID/Value pair for each element we are looking for.
pTemp = PackString(m_CommonName);
Filter[i].Field.FieldOid = CSSMOID_CommonName;
Filter[i].Field.FieldValue = *pTemp;
free(pTemp);
i++;

pTemp = PackString(m_IssuerName);
Filter[i].Field.FieldOid = CSSMOID_IssuerName;
Filter[i].Field.FieldValue = *pTemp;
free(pTemp);
i++;

// Get the first certificate
pTemp = CSSM_DL_CertGetFirst( hDL, hDB, Filter, FilterSize, CSSM_AND,
                              &ResultsHandle, &NumCerts);

// Keep getting the next certificate until you can't get any more
while (pTemp)
{
      toReturn = pTemp;
      pTemp = CSSM_DL_CertGetNext( hDL, hDB, ResultsHandle );
}

// Clean up
for (i=0; i<FilterSize; i++)
{
      if (Filter[i].Field.FieldValue.Data)
            free(Filter[i].Field.FieldValue.Data);
}
free(Filter);

return toReturn;
}
```