NLnet Labs

An introduction to the use of HSM

Jelte Jansen*, *NLnet Labs*

NLnet Labs document 2008-draft May 13, 2008

Abstract

This document describes the use of Hardware Security Modules (HSM).

It contains information and examples on how to get them working in your environment with free software tools.

The main part of this document consists of two tutorials on how to make your software applications capable of using hardware modules, both through OpenSSL's EVP API, and PKCS #11. These should get a software developer started, and provide pointers where to go from there. The programming language used is C.

The appendices contain information about specific hardware modules, like the AEP Keyper and the Eutron ITSEC USB token.

Contents

*jelte@NLnetLabs.nl

1 Introduction

In this document we'll give an introduction to hardware assisted cryptography. We'll talk a bit about PKCS #11, the 'standard' API for smartcards and hardware cryptography modules. We'll also discuss the OpenSSL EVP API, which has, amongst others, a PKCS #11 backend.

We'll also give some pointers about how one would go about getting these magical machines working, with some specific examples of the hardware we've had the honour of playing with. In the main part of this document, we'll give two tutorials on programming your own applications to talk to these machines.

Some knowledge on cryptography is assumed. Mainly, we won't discuss specific algorithms or best practices. They depend largely on the intended use, and there are other, more extensive documents that handle these subjects. In examples given, we keep to a few commonly used algorithms, but this is in no way a reason to choose them over others.

Thanks go out to the gentle people at AEP Networks, for providing a Keyper HSM to test with.

This document is also available on-line in HTML format at
`http://www.NLnetLabs.nl/publications/hsm`.
The code examples can be found at
`http://www.NLnetLabs.nl/publications/hsm/examples`

This is version 1.0 of the document, dated May 2008. If you find an error, have any comments, or would like to see this document extended, please let me know at `jelte@NLnetLabs.nl`

1.1 HSM

A Hardware Security Module (HSM) can come in various shapes and forms; there are smart cards, PCI cards to plug into a PC, usb tokens, separate boxes that communicate over channels like TCP/IP, USB or rs-232, etc. Regardless of shape or package, the main purposes of these modules is usually either:

- Speeding up cryptographic operations
- Keeping keys safe

But of course, there could be other purposes.

Some modules may be able to offer both, but more often than not this is not the case.

1.2 PKCS

The PKCS line of standards was made by RSA laboratories. These supply a wealth of specifications concerning cryptography. What we are mainly interested in here is number 11, also called Cryptoki, which deals with a standardized API for talking to cryptographic tokens.

1.3 FIPS 140

The Federal Information Processing Standard 140 is a series of standards concerning cryptographic modules, both in hardware and software. The current version is FIPS 140-2, but a third is in development.

The latest version can be found on the NIST website at
`http://csrc.nist.gov/publications/PubsFIPS.html`

It defines a number of levels of security, that a certain module can be certified for, in short:

- Level 1: This is the lowest level. A security level 1 cryptographic module does not have to have physical protection, and only need to incorporate one approved algorithm or function.
- Level 2: Security level 2 requires tamper evidence to be added to the module, as well as role-based authentication.
- Level 3: In addition to tamper evidence, for security level 3, a module must also provide tamper resistance. This level also requires identity-based authentication.
- Level 4: This is the highest level specified by FIPS 140. This level requires complete protection around the cryptographic module, detecting and responding to all unauthorized attempts at physical access, as well as environmental anomalies (power fluctuations, extreme temperatures).

2 Tools and Software

In this section, we'll talk about some common open-source tools to manipulate and use data on smart cards and other crypto hardware with a PKCS #11 interface. Before you start writing everything yourself, it might be a good idea to check whether there are other implementations that may suit your needs.

2.1 OpenSC

OpenSC (`http://www.opensc-project.org/`) is a set of libraries and drivers for smartcards and cryptographic tokens. It is designed to work with PKCS #11 supporting cards. OpenCT provides drivers for card readers, and tokens that are comprised of both a card and a reader (ie. usb tokens, and other 'complete' devices).

The project maintains a compatibility list at its web site:
`http://www.opensc-project.org/openct/wiki/`

But even if a device is not on the list, it might work. A good way to check is to just try.

```
% opensc-tool -l
Readers known about:
Nr.    Driver     Name
0      openct     OpenCT reader (detached)
1      openct     OpenCT reader (detached)
2      openct     OpenCT reader (detached)
3      openct     OpenCT reader (detached)
4      openct     OpenCT reader (detached)
```

So we see that it uses the openct package as a backend, but does not see any cards or tokens present at this moment.

So let's see what happen when we insert a token. See the specific appendix on how to get the driver working, if needed.

Let's insert an Eutron ITSEC CryptoIdentity token, and run our program again.

```
% opensc-tool -l
Readers known about:
Nr.    Driver     Name
0      openct     Eutron CryptoIdendity
```

If you get strange errors here, refer to the appendix for more information, and see if your device is listed there.

This tool can be used for a few basic functions, or you could even construct your own data packet and send it to the card, but other than that its use is limited, and it can only see tokens directly supported by openct. Let's take a look at a more flexible tool.

2.2 pkcs11-tool

This tool, which also comes with opensc, gives the user the option to provide a driver module. I.E. the module that exports the PKCS #11 functions. So as long as you have one of those suitable for your operating system, you can use this to access your token.

Actual results may differ though, while I have been able to do basic operations with the AEP Keyper for example, I did notice some minor faults in the tool, and in the drivers. Of course, these will be probably be fixed in the next version, where possible, I'll try to describe problems and workarounds for specific devices in the appendices.

Examples:

For these examples we use the AEP Keyper, whose driver is called pkcs11.GCC4.0.2.so.4.04. See appendix A for more information about getting it running.

We have just initialized it as specified in the documentation.

```
% pkcs11-tool --module=/opt/lib/pkcs11.GCC4.0.2.so.4.04 -l -O
Please enter User PIN:
%
```

The -l option tells the tool to ask for the user PIN and log into the token if necessary. The -O switch asks the token for a list of objects, and prints some information about them.

Since we have just initialized the token, there is no information on it yet. So let's generate a key that we can use.

```
% pkcs11-tool --module=/opt/lib/pkcs11.GCC4.0.2.so.4.04 -l -k \
> -d 01 -a zone_key --key-type rsa:2048
Please enter User PIN:
Key pair generated:
Private Key Object; RSA
  label:      zone_key
  ID:         01
  Usage:      decrypt, sign, unwrap
Public Key Object; RSA 2048 bits
  label:      zone_key
  ID:         01
  Usage:      encrypt, verify, wrap
```

Again, for this operation (as for most) we need to log in to the token, so we provide -l. Then we tell it to create a key pair (-k) with id 1 (-d <hex>) and label 'zone_key' (-a). We want the key to be of type RSA, with a modulus size of 2048 bits (–key-type).

As before, the program asks us for the token user's PIN and then starts generating the key. When it's done, it prints out some information about the created objects.

How you can make a backup of your keys, or move them to another token is out of scope for this document, but detailed instructions about this should be provided in the manual of the better HSM devices.

Some devices cannot create certain kinds of keys, for instance when they are too big, or when they would provide an operation that the device cannot support. In that case, this command will obviously fail.

Let's see whether it has really worked:

```
% pkcs11-tool --module=/opt/lib/pkcs11.GCC4.0.2.so.4.04 -l -O
Please enter User PIN:
Private Key Object; RSA
  label:      zone_key
  ID:         01
  Usage:      decrypt, sign, unwrap
```

The label and id are as we previously specified. The label is just a string where you can place information about this specific key. The ID is a hexadecimal number that you use to specify this specific key when calling the application that needs to use it.

So, now that we have a key on there, we should be able to use any program that talks PKCS #11 to sign data. Here's one that does indirectly (through OpenSSL's EVP API).

```
% OPENSSL_CONF=~/aep_ssl_conf ldns-signzone -E pkcs11 \
> -k 1,5 nlnetlabs.nl
Engine key id: 1, algo 5
PKCS#11 token PIN:
%
```

This program itself is not relevant, except maybe that it signs data, but the way it works is. Through an environment variable we tell it to use a specific configuration file. That file adds the engine 'pkcs11' to the list of possible OpenSSL engines, and the driver we used previously. We talk more about this configuration file in section 4.1.

With that configuration file read, and OpenSSL adding the specified engines to its list, we tell the tool to use 'pkcs11', and 'key 1'. The ',5' is for the tool to know what to do with that exact key, and is not relevant here.

3 PKCS #11

The best way to learn PKCS #11 is, well, to read the specification, available at
`http://www.rsa.com/rsalabs/node.asp?id=2133`

It is a complete API description containing everything you need to know and more. Unfortunately, it might be a bit more than you'll actually need, and finding the information you *do* need can be a bit hard. However, when you understand the basics, it gets a lot easier.

In this chapter we'll give a brief overview of how to use it, and how to read the specification and find information relevant to your situation fast.

The PKCS #11 standard uses Hungarian notation, together with CamelCase for its identifiers. To differentiate between our own example names and PKCS #11 names, we'll use underscore-delimited identifiers in our examples. We do however declare variables in CamelCase when they are used as arguments for PKCS #11 functions.

3.1 Function definitions

A typical function definition in the specification looks like this:

```
CK_DEFINE_FUNCTION(CK_RV, C_GetObjectSize)(
   CK_SESSION_HANDLE hSession,
   CK_OBJECT_HANDLE hObject,
   CK_ULONG_PTR pulSize
);
```

This defines a function named 'C_GetObjectSize', the return value of which is of type 'CK_RV'.

It takes three arguments:

1. a CK_SESSION_HANDLE, which refers to a session
2. a CK_OBJECT_HANDLE, which refers to an object (for instance a key)
3. a CK_ULONG_PTR, which is a pointer to a CK_ULONG type.

An overview of all functions is presented in chapter 11 of the specification.

What functions, mechanisms and operations depends on the module that is used, and what state it is in. Refer to the documentation of your specific module for more information about this.

3.2 Templates

A common way of conversing with PKCS #11 is through templates. You create a template containing the type of information or result you need, and when you call a specific function with that template, the engine will fill it in for you.

Most functions that provide information, or generate data, are called like this:

1. Create template
2. Set template parameters
3. Call function with template
4. Read template results

One possible result, if specified by the caller, is to only set information about the amount of memory needed for the result data, and not set the result data itself. The usual way this is used is this:

1. Create template
2. Set NULL pointers for result data
3. Call function with template
4. Read the result sizes, and allocate memory
5. Set the allocation result pointers in the template
6. Call function with template again

This way, an application does not need to guess how much memory it needs. For simplicity, we do not use this method in our examples later in this chapter, but it is usually a good idea if you do not know exactly how much data to expect.

These are a few of the very basics. It might be a good idea to dive into some actual code. But first, we need to get one thing out of the way.

3.3 Platform specific header

Unfortunately, PKCS #11 is not completely platform-independent. In order for your application to work, you'll have to provide some platform-specific definitions.

For this example, we'll just place them in the main header file. If your application has to be portable, you could provide separate header files and choose the correct one through a configure script.

In this example, we chose for Linux. The specific settings for other platforms can be found in chapter 8 of the specification.

Let's create a header file pkcs11_linux.h:

```
#define CK_PTR *
#define CK_DEFINE_FUNCTION(returnType, name) \
     returnType name
#define CK_DECLARE_FUNCTION(returnType, name) \
     returnType name
#define CK_DECLARE_FUNCTION_POINTER(returnType, name) \
     returnType (* name)
#define CK_CALLBACK_FUNCTION(returnType, name) \
     returnType (* name)
```

```
#ifndef NULL_PTR
#define NULL_PTR 0
#endif

#include "pkcs11.h"
#include <stdio.h>
#include <stdlib.h>
```

3.4 Initialization and cleanup

Let's walk through a typical session. We'll set up the environment, read what keys are present on the device, and print some summary information about them.

First of all, we'll include the header we created in section 3.3

```
#include "pkcs11_linux.h"
```

Most PKCS #11 functions return a status code of the type CK_RV. Because we do not want to make an example that doesn't use error checking at all, but we do not want to clutter up the code with graceful error handling, we'll just provide a function that checks the return value and exits with an error message if there was a problem. The standard specifies exactly what error codes all functions can return, so graceful error handling should not be too much of a problem. A good idea is to at least map the error code back to its name.

```
void
check_return_value(CK_RV rv, const char *message)
{
        if (rv != CKR_OK) {
                fprintf(stderr, "Error at %s: %u\n",
                        message, (unsigned int)rv);
                exit(EXIT_FAILURE);
        }
}
```

So, now that we have that out of the way, let's initialize the environment.

```
CK_RV
initialize()
{
        return C_Initialize(NULL);
}
```

The initialization function has one argument. In it, you can specify some global options. These all have to do with threading. See section 6.6 and section 11.4 of the specification for more information. Since we are not making a threaded application here, we can safely pass NULL.

With the library initialized, it's time to select a slot to use. PKCS providers usually provide multiple slots that can take tokens. In this case, we just want the first one.

```
CK_SLOT_ID
get_slot()
{
     CK_RV rv;
     CK_SLOT_ID slotId;
     CK_ULONG slotCount = 10;
     CK_SLOT_ID *slotIds = malloc(sizeof(CK_SLOT_ID) * slotCount);

     rv = C_GetSlotList(CK_TRUE, slotIds, &slotCount);
     check_return_value(rv, "get slot list");

     if (slotCount < 1) {
          fprintf(stderr, "Error; could not find any slots\n");
          exit(1);
     }

     slotId = slotIds[0];
     free(slotIds);
     printf("slot count: %d\n", (int)slotCount);
     return slotId;
}
```

The C_GetSlotList function takes three arguments, the first one is a boolean value that specifies whether to only return slots that actually have a token present at the moment, or just return any slots. The second is an array to place the slot ID values in, and the third arguments is a pointer to the number of slots we have allocated memory for.

As explained in 3.2, if the second argument would have been NULL, the number of slots found will be set in the third argument. This value could be used to allocate exactly the right amount of memory, after which the function would be called again. This mechanism is used often in PKCS #11.

In this case we'll do it in one step, and assume our reader does not have more than 10 slots. If it does, this function should return the error code CKR_BUFFER_TOO_SMALL. Some implementations might just return the first 10 slots, although you shouldn't count on that.

So, if everything went right, we have a slot with a token now. For that slot, we'll start a session. This session will be the context for all actions we do later.

```
CK_SESSION_HANDLE
start_session(CK_SLOT_ID slotId)
{
        CK_RV rv;
        CK_SESSION_HANDLE session;
        rv = C_OpenSession(slotId,
                           CKF_SERIAL_SESSION,
                           NULL,
```

```
                                NULL,
                                &session);
        check_return_value(rv, "open session");
        return session;
}
```

The second argument is a flags parameter, in which certain settings can be set. CKF_SERIAL_SESSION must always be set. If we would want to have write access to the token (which we don't for just reading keys), we would XOR this with the value CKF_RW_SESSION.

If we would want the library to notify it of certain events in the session, we would provide a callback function as the third argument. See section 11.17 of the specification for more information on this.

The final argument is a pointer to the CK_SESSION_HANDLE variable that we'll use to refer to the session we have opened.

If the token needs a user to provide a PIN, we can use the function C_login. The pin is an array of bytes. Some implementations accept a NULL pin if it wasn't set, although this does not follow the specification.

```
void
login(CK_SESSION_HANDLE session, CK_BYTE *pin)
{
     CK_RV rv;
     if (pin) {
          rv = C_Login(session, CKU_USER, pin, strlen((char *)pin));
          check_return_value(rv, "log in");
     }
}
```

That's the initialization. Now we could go and actually do something. But, for completeness' sake, let's first get the cleanup routines out of the way, these are not much more than the reversed function made so far.

```
void
logout(CK_SESSION_HANDLE session)
{
     CK_RV rv;
     rv = C_Logout(session);
     if (rv != CKR_USER_NOT_LOGGED_IN) {
          check_return_value(rv, "log out");
     }
}
```

We do one additional check on the return value; if we did not pass a PIN to the login function, we would not be logged in at all, resulting in the error CKR_USER_NOT_LOGGED_IN. Rather than keeping track of whether we should have been logged in, we'll just watch out for this error.

Two things remain: we are still in a session, and we need to clean up the library.

```
void
end_session(CK_SESSION_HANDLE session)
{
        CK_RV rv;
        rv = C_CloseSession(session);
        check_return_value(rv, "close session");
}

void
finalize()
{
        C_Finalize(NULL);
}
```

3.5 Example 1: Reading keys

Okay, it's time to do something interesting. In this example, we want to find all the private keys on the token, and print their label and id.

Keys are just one type of object that a token can contain, so if we would have magically acquired a pointer already (which we'll get to in the next function), this is the way to get information about it:

```
void
show_key_info(CK_SESSION_HANDLE session, CK_OBJECT_HANDLE key)
{
     CK_RV rv;
     CK_UTF8CHAR *label = (CK_UTF8CHAR *) malloc(80);
     CK_BYTE *id = (CK_BYTE *) malloc(10);
     size_t label_len;
     char *label_str;

     memset(id, 0, 10);

     CK_ATTRIBUTE template[] = {
          {CKA_LABEL, label, 80},
          {CKA_ID, id, 1}
     };

     rv = C_GetAttributeValue(session, key, template, 2);
     check_return_value(rv, "get attribute value");

     fprintf(stdout, "Found a key:\n");
     label_len = template[0].ulValueLen;
     if (label_len > 0) {
```

```
            label_str = malloc(label_len + 1);
            memcpy(label_str, label, label_len);
            label_str[label_len] = '\0';
            fprintf(stdout, "\tKey label: %s\n", label_str);
            free(label_str);
        } else {
            fprintf(stdout, "\tKey label too large, or not found\n");
        }
        if (template[1].ulValueLen > 0) {
            fprintf(stdout, "\tKey ID: %02x\n", id[0]);
        } else {
            fprintf(stdout, "\tKey id too large, or not found\n");
        }

        free(label);
        free(id);
}
```

First of all, we define a template containing the types of information we want. A template is an array of (type, pointer, size) values. The type specifies what information we want, the pointer points to allocated data that will contain the result, and the size points to the amount of memory we have allocated there. If the 'pointer' is NULL, the size needed will be placed in 'size', so the application can then allocate exactly enough memory and call this function again. This is just like the way C_GetSlotList worked.

If a value can not be extracted, or the amount of memory we have allocated is not enough, the 'size' value will be -1 after the call has completed.

What data you can ask for depends on the type of object and the function you call.

With this template, we call C_GetAttributeValue. Of course we need to provide the session, the object we want data about, and the template. The final argument is the number of entries in our template array. In this case, we have specified 2 data types we want to get.

So after the call completed, and the library has supposedly filled out our template, we check the size values, and if there is data, print it.

Now we'll make a function that finds the keys on the token, and call the previous function for each key.

```
void
read_private_keys(session)
{
        CK_RV rv;
        CK_OBJECT_CLASS keyClass = CKO_PRIVATE_KEY;
        CK_ATTRIBUTE template[] = {
            { CKA_CLASS, &keyClass, sizeof(keyClass) }
        };
```

```
        CK_ULONG objectCount;
        CK_OBJECT_HANDLE object;

        rv = C_FindObjectsInit(session, template, 1);
        check_return_value(rv, "Find objects init");

        rv = C_FindObjects(session, &object, 1, &objectCount);
        check_return_value(rv, "Find first object");

        while (objectCount > 0) {
                show_key_info(session, object);

                rv = C_FindObjects(session, &object, 1, &objectCount);
                check_return_value(rv, "Find other objects");
        }

        rv = C_FindObjectsFinal(session);
        check_return_value(rv, "Find objects final");
}
```

As in the previous function, we create a template. This time it is used as a filter in the C_FindObjectsInit function. Every object that matches the template will be returned with subsequent calls to C_FindObjects. You can ask for more than one object at a time, but for now, lets just do them one at a time.

When we're done, we call C_FindObjectsFinal, this cleans up any memory allocated by C_FindObjectsInit.

Let's tie this all together in an actual application:

```
int
main(int argc, char **argv)
{
        CK_SLOT_ID slot;
        CK_SESSION_HANDLE session;
        CK_BYTE *userPin = NULL;
        CK_RV rv;

        if (argc > 1) {
                if (strcmp(argv[1], "null") == 0) {
                        userPin = NULL;
                } else {
                        userPin = (CK_BYTE *) argv[1];
                }
        }

        rv = initialize();
        check_return_value(rv, "initialize");
```

```
    slot = get_slot();
    session = start_session(slot);
    login(session, userPin);
    read_private_keys(session);
    logout(session);
    end_session(session);
    finalize();
    return EXIT_SUCCESS;
}
```

In case we have a token without a user PIN, we'll let the user specify it on the command line. Normally, it would be a better idea to provide a callback function, or at least do both.

That's it! Compile it, link it, stick it in a stew, and you have your first PKCS #11 application. Wasn't that easy?

On the other hand, this example is only useful when there are actually keys present on the token. What if there aren't, and we want to add them ourselves? Let's do another example.

3.6 Example 2: Generating a key pair

If we use the same initialization code as we did in 3.4, well get an error when we try to generate the key pair; CKR_SESSION_READ_ONLY. So we have to change one thing there:

```
CK_SESSION_HANDLE
start_session(CK_SLOT_ID slotId)
{
    CK_RV rv;
    CK_SESSION_HANDLE session;
    rv = C_OpenSession(slotId,
                       CKF_SERIAL_SESSION | CKF_RW_SESSION,
                       NULL,
                       NULL,
                       &session);
    check_return_value(rv, "open session");
    return session;
}
```

So when we have that, we can create the keypair creation call. Again, we define templates first. Two this time, one for the public key, and one for the private key.

We set the mechanism, the number of bits in the modulus, and the exponent (to RSA_F4, or 65537, or 0x101).

We define a general boolean variable true, which we use a few times to set flags.

```
void
create_key_pair(CK_SESSION_HANDLE session)
{
     CK_RV rv;
     CK_OBJECT_HANDLE publicKey, privateKey;
     CK_MECHANISM mechanism = {
          CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
     };
     CK_ULONG modulusBits = 1024;
     CK_BYTE publicExponent[] = { 1, 0, 1 };
     CK_BYTE subject[] = "mykey";
     CK_BYTE id[] = {0xa1};
     CK_BBOOL true = CK_TRUE;
     CK_ATTRIBUTE publicKeyTemplate[] = {
          {CKA_ID, id, 3},
          {CKA_LABEL, subject, 5},
          {CKA_TOKEN, &true, sizeof(true)},
          {CKA_ENCRYPT, &true, sizeof(true)},
          {CKA_VERIFY, &true, sizeof(true)},
          {CKA_WRAP, &true, sizeof(true)},
          {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
          {CKA_PUBLIC_EXPONENT, publicExponent, 3}
     };
     CK_ATTRIBUTE privateKeyTemplate[] = {
          {CKA_ID, id, sizeof(id)},
          {CKA_LABEL, subject, 5},
          {CKA_TOKEN, &true, sizeof(true)},
          {CKA_PRIVATE, &true, sizeof(true)},
          {CKA_SENSITIVE, &true, sizeof(true)},
          {CKA_DECRYPT, &true, sizeof(true)},
          {CKA_SIGN, &true, sizeof(true)},
          {CKA_UNWRAP, &true, sizeof(true)}
     };

     rv = C_GenerateKeyPair(session,
                            &mechanism,
                            publicKeyTemplate, 8,
                            privateKeyTemplate, 8,
                            &publicKey,
                            &privateKey);
     check_return_value(rv, "generate key pair");
}
```

Most data in these templates is hardcoded. Of course, for an actual application, these need to be provided in a more dynamic way.

The templates themselves should be straightforward, for the public key:

1. We want the id to be a1. A key id is a computer-readable identifier
2. We want the label to be 'mykey'. A key label is a human-readable identifier
3. We want the key to be stored on the token. If this value is not set, the key will not be stored, and will be destroyed once the current session is closed.
4. We want the key to be used for encryption
5. We want the key to be used for verification
6. We want the key to be able to encrypt other key data
7. We want a modulus size of 1024 bits
8. We want the exponent to be 65537

And for the private key:

1. We want the id to be a1
2. We want the label to be 'mykey'
3. We want the key to be stored on the token
4. We want the key to only be usable by logged-in users
5. We want the sensitive key data to always be kept inside the token
6. We want the key to be used for decryption
7. We want the key to be used for signing
8. We want the key to be able to decrypt other key data

There are, of course, a lot of other options. This also depends on the exact mechanism, and what it can do. To find this out, you will have to consult the PKCS #11 specification.

Some tokens (for instance our Eutron ITSEC CryptoIdentity), will not allow keys to be used for multiple purposes. In that case you would get an error code CKR_GENERAL_ERROR, along with a message to specify a specific key usage. In that case, remove the (CKA_)WRAP, UNWRAP, ENCRYPT, and DECRYPT lines from the template. Do not forget to lower the number 8 to 6 in the call to C_GenerateKeyPair.

Since we hardcoded the necessary variables, you should be able to use the same main() function as in example 1, with only the call to read_private_keys replaced by create_key_pair.

3.7 Example 3: Signing data

If the last example worked on your particular device, we should have some keys now. You should be able to verify it by calling the application for example 1 again.

But what good are keys if we don't use them? In this example, we'll create a very simple application to sign the contents of a file, and store the signature data in another. Normally, this data would be enclosed in some form of envelope, depending on the security protocol used. But again, for the sake of simplicity, we'll just write the raw data.

We can use the same initalization functions as in example 1 and 2. Since we will not be writing to the device, the flag CKF_RW_SESSION can be removed again from start_session.

We are going to use Sha1WithRSA, so let's first get the SHA1 digest out of the way. Some devices and PKCS #11 libraries can do this in one go, by using the mechanism CKM_SHA1_RSA_PKCS, but some can only do them separately.

```
CK_RV
digest_data(CK_SESSION_HANDLE session,
            FILE *data_file,
            CK_BYTE *digest,
            CK_ULONG *digestLen)
{
     CK_MECHANISM digest_mechanism;
     CK_RV rv;

     CK_ULONG dataLen;
     CK_BYTE *data = malloc(1024);

     if (!data_file) {
          fprintf(stderr, "Error, no file handle in digest_data\n");
          exit(3);
     }

     digest_mechanism.mechanism = CKM_SHA_1;
     digest_mechanism.pParameter = NULL;
     digest_mechanism.ulParameterLen = 0;

     rv = C_DigestInit(session, &digest_mechanism);
     check_return_value(rv, "digest init");

     printf("read data from %p\n", data_file);
     dataLen = fread(data, 1, 1024, data_file);
     printf("read %u bytes\n", (unsigned int)dataLen);
     while (dataLen > 0) {
          printf("add %u bytes to digest\n", (unsigned int)dataLen);
          rv = C_DigestUpdate(session, data, dataLen);
```

```
            check_return_value(rv, "digest update");
            dataLen = fread(data, 1, 1024, data_file);
        }

        rv = C_DigestFinal(session, digest, digestLen);
        check_return_value(rv, "digest final");

        free(data);
        return rv;
}
```

In this method, we initialize a digest operation with the mechanism CKM_SHA_1. This mechanism does not take any arguments, so we set pParameter to NULL. After we call C_DigestInit, we feed it the data we are reading from the input file.

The actual digest operation is performed when we call C_DigestFinal. If the memory we allocated for the digest is too little (as told to the implementation by the variable dataLen in this case), this would return the error CKR_BUFFER_TOO_SMALL. Otherwise, the hash is calculated, copied to the given buffer, and the operation is cleaned up and finished. After this, the session can start another operation. If the above error was returned, the operation is not finished, and initializing a new operation would result in an error.

So now we have the digest. But if we want to do any signing, we are going to need a specific key, so let's make a function to get one.

```
CK_OBJECT_HANDLE
get_private_key(CK_SESSION_HANDLE session, CK_BYTE *id, size_t id_len)
{
        CK_RV rv;
        CK_OBJECT_CLASS keyClass = CKO_PRIVATE_KEY;
        CK_ATTRIBUTE template[] = {
                { CKA_CLASS, &keyClass, sizeof(keyClass) },
                { CKA_ID, id, id_len }
        };
        CK_ULONG objectCount;
        CK_OBJECT_HANDLE object;

        rv = C_FindObjectsInit(session, template, 1);
        check_return_value(rv, "Find objects init");

        rv = C_FindObjects(session, &object, 1, &objectCount);
        check_return_value(rv, "Find first object");

        if (objectCount > 0) {
                rv = C_FindObjectsFinal(session);
                check_return_value(rv, "Find objects final");
                return object;
```

```
	} else {
		fprintf(stderr, "Private key not found\n");
		exit(2);
	}
}
```

This function is a lot like the one we used in example 1. It initializes a search, and the gets the first object found. The main difference is that we specify more in the filter template. We add a specific key ID. Of course, you could also choose other parameters, like a CKA_LABEL. Don't forget to clean up by calling C_FindObjectsFinal.

While we're at it, we also make a function to get a public key, with which we can verify a signature.

```
CK_OBJECT_HANDLE
get_public_key(CK_SESSION_HANDLE session, CK_BYTE *id, size_t id_len)
{
	CK_RV rv;
	CK_OBJECT_CLASS keyClass = CKO_PUBLIC_KEY;
	CK_ATTRIBUTE template[] = {
		{ CKA_CLASS, &keyClass, sizeof(keyClass) },
		{ CKA_ID, id, id_len }
	};
	CK_ULONG objectCount;
	CK_OBJECT_HANDLE object;

	rv = C_FindObjectsInit(session, template, 1);
	check_return_value(rv, "Find objects init");

	rv = C_FindObjects(session, &object, 1, &objectCount);
	check_return_value(rv, "Find first object");

	if (objectCount > 0) {
		rv = C_FindObjectsFinal(session);
		check_return_value(rv, "Find objects final");
		return object;
	} else {
		fprintf(stderr, "Public key not found\n");
		exit(3);
	}
}
```

This function is exactly the same as the previous one, but for one detail; the CKA_CLASS value. This is a good hint that you might be better of specifying this through a function argument, but this makes it more clear for now.

So when we can retrieve our keys, we can sign data.

```
void
sign_data(CK_SESSION_HANDLE session, FILE *data_file, FILE *signature_file)
{
     CK_RV rv;
     CK_BYTE id[] = { 0x45 };
     CK_OBJECT_HANDLE key = get_private_key(session, id, 1);
     CK_MECHANISM sign_mechanism;

     CK_ULONG digestLen = 20;
     CK_BYTE *digest = malloc(digestLen);

     CK_ULONG signatureLen = 512;
     CK_BYTE *signature = malloc(signatureLen);

     sign_mechanism.mechanism = CKM_RSA_PKCS;
     sign_mechanism.pParameter = NULL;
     sign_mechanism.ulParameterLen = 0;

     rv = digest_data(session, data_file, digest, &digestLen);
     check_return_value(rv, "digest data");

     rv = C_SignInit(session, &sign_mechanism, key);
     check_return_value(rv, "sign init new");

     rv = C_Sign(session, digest, digestLen, signature, &signatureLen);
     check_return_value(rv, "sign final");

     if (signatureLen > 0) {
          fwrite(signature, signatureLen, 1, signature_file);
     }

     free(digest);
     free(signature);
}
```

As you can see, first we get a digest and initialize a signing operation. At the end we get the signature by calling C_SignFinal. If you have do not have a fixed amount of data to sign, you can also replace C_Sign by various calls of the function C_SignUpdate, followed by C_SignFinal. Both C_Sign and C_SignFinal end the signing operation initialized by C_SignInit. In our case, however, we have already performed a digest operation, and so our data will always have a fixed length.

We allocate 512 bytes of data for the signature. This is enough to store a signature made with an 4096-bit RSA key. If the key that is used is smaller, the value of signatureLen will be changed in the call to C_Sign.

In order for the library and device to know exactly what they have to do, we

need to specify a mechanism again. In this case we choose CKM_RSA_PKCS. The resulting signature size depends on the size of the modulus of the RSA key used.

A full list of possible mechanisms can be found in section 12 of the specification. Of course, your device needs to support the mechanism chosen, or you'll get an error code CKR_MECHANISM_INVALID.

If the signing operation succeeds, we write it to the output_file file pointer.

We have hardcoded the key ID here. Of course you'll want to change that to your specific key ID. If you used the key generation function in example 2, you'll need to set it to 0xa1. Better yet would be to let the user specify the key.

For the sake of symmetry, we'll just go ahead and create a function to verify a signature too:

```
void
verify_data(CK_SESSION_HANDLE session, FILE *data_file, FILE *signature_file)
{
     CK_RV rv;
     CK_BYTE id[] = { 0x45 };
     CK_OBJECT_HANDLE key = get_public_key(session, id, 1);
     CK_MECHANISM sign_mechanism;

     CK_ULONG digestLen = 20;
     CK_BYTE *digest = malloc(digestLen);

     CK_ULONG signatureLen = 512;
     CK_BYTE *signature = malloc(signatureLen);

     signatureLen = fread(signature, 1, signatureLen, signature_file);

     sign_mechanism.mechanism = CKM_RSA_PKCS;
     sign_mechanism.pParameter = NULL;
     sign_mechanism.ulParameterLen = 0;

     rv = digest_data(session, data_file, digest, &digestLen);
     check_return_value(rv, "digest data");

     rv = C_VerifyInit(session, &sign_mechanism, key);
     check_return_value(rv, "verify init");

     rv = C_Verify(session,
                   digest,
                   digestLen,
                   signature,
                   signatureLen
                   );
     check_return_value(rv, "verify");
```

```
	printf("The signature is valid\n");

	free(digest);
	free(signature);
}
```

Again, we hardcode the key ID. And since we know it's modulus size, we also hardcode the signature length. Again we reserve 512 bytes for the signature, which is enough for a signature made with a 4096-bit RSA key.

The method to verify a signature looks quite a bit like the method to create one; We initialize the verification procedure, create a digest from the input data, and check the result by calling C_Verify with the signature we read.

If the signature is valid, C_Verify will just return CKR_OK. If it is invalid, it will either return CKR_SIGNATURE_INVALID or CKR_SIGNATURE_LEN_RANGE. That last error code is returned when the signature is obviously invalid because it has the wrong length.

Again, a setup with C_VerifyUpdate and C_VerifyFinal is also possible instead of just C_Verify. Both C_VerifyFinal and C_Verify end the current operation.

Because we do need a bit more input here than in the first two examples, we expand on the main function a bit:

```
int
main(int argc, char **argv)
{
	CK_SLOT_ID slot;
	CK_SESSION_HANDLE session;
	FILE *input_file = NULL;
	FILE *output_file = NULL;
	CK_BYTE *user_pin = NULL;

	if (argc < 3) {
		printf("Usage: pkcs11_example3 <input file> <output file> <pin>\n");
		exit(0);
	}
	if (argc > 3) {
		user_pin = (CK_BYTE *) argv[3];
	}

	initialize();
	slot = get_slot();

	/* signing */
	input_file = fopen(argv[1], "r");
	output_file = fopen(argv[2], "w");
	session = start_session(slot);
```

```
        if (user_pin) {
             login(session, user_pin);
        }
        sign_data(session, input_file, output_file);
        if (user_pin) {
             logout(session);
        }
        end_session(session);
        fclose(input_file);
        fclose(output_file);

        /* verification */
        input_file = fopen(argv[1], "r");
        output_file = fopen(argv[2], "r");
        session = start_session(slot);
        if (user_pin) {
             login(session, user_pin);
        }
        verify_data(session, input_file, output_file);
        if (user_pin) {
             logout(session);
        }
        end_session(session);
        fclose(input_file);
        fclose(output_file);

        finalize();

        return EXIT_SUCCESS;
}
```

Here, we actually perform two operations, which should be completely separate; first we open the two files for signing the first, and then we open them again to verify the signature we've just created. Feel free to comment out either one, or split them up into separate functions. There's no error checking on the file operations, which should obviously be present in a robust application.

One additional thing; some tokens or drivers cannot handle multiple operations within the same session well. Even though the operation has been finalized, it will not start a new one. Some tokens might be silent and stop the first operation if you initialize a second within the same session, but others might return an error even if you did finalize it. Therefore, in this example we just start a completely new session.

That's it!

Please start skimming over the PKCS #11 specification again. Hopefully you can understand it better, now that you have seen some actual code in action.

4 OpenSSL and EVP

OpenSSL is an open source library that implements the SSL and TLS protocols. More than that, OpenSSL also provides an exhaustive general cryptography library, with both high- and lowlevel API's.

EVP is an OpenSSL API that provides a high-level interface to cryptographic functions. While OpenSSL also has direct interfaces for operations like signing data with an RSA key, the EVP library separates the operations from the actual backend used. That way, the actual implementation that is used can be changed, and one can specify an engine to use for the operations.

One of the engines that can be selected in recent versions is a pkcs11 engine.

What does this imply? If your application uses the EVP library, it's very easy to let your users use their HSM for their cryptographic needs, as long as their HSM is supported by a driver.

For an example of usage, see getting things to work 4.1.

If you use this API, it's still a lot like you'd have used the old low-level functions, and you can still use your specific internal cryptographic algorithms. However, a very simple addition makes the code a lot more flexible:

```
OpenSSL_load_config(NULL)
load_engine()
```

Instead of NULL you can provide a filename, but with null the value of the environment variable $OPENSSL\_CONF$ is used. In the provided file, the user can add possible engines. For a better user experience, it is nice to provide a configuration or command-line option that specifies the file, instead of letting the user provide it through an environment variable.

4.1 Configuration file

Here's a simple configuration file that selects the PKCS11 engine, and what actual module to use with the engine.

```
# PKCS11 engine config#####################################################
openssl_conf            = openssl_def

[openssl_def]
engines = engine_section

[engine_section]
pkcs11 = pkcs11_section

[pkcs11_section]
engine_id = pkcs11
dynamic_path = /usr/lib/engines/engine_pkcs11.so
MODULE_PATH = /home/jelte/pkcs11/aep/pkcs11.GCC4.0.2.so.4.04
init = 0
```

The first few sections are some abstractions used. The important section here is `pkcs11_section`. This provides an engine id, a dynamic path for the main driver (in this case a pkcs11 module), and a specific driver for the actual HSM used.

When you now run an openssl application with this config file, the pkcs11 engine is added to the list of available engines.

4.2 The openssl command-line tool

Now that we have a configuration file, it's time to test it;

```
% openssl engine
(padlock) VIA PadLock (no-RNG, no-ACE)
(dynamic) Dynamic engine loading support
%
% OPENSSL_CONF=./my_ssl_conf openssl engine
(padlock) VIA PadLock (no-RNG, no-ACE)
(dynamic) Dynamic engine loading support
(pkcs11) pkcs11 engine
%
```

If the engine does not show up, you might want to check whether you specified the right configuration file. OpenSSL silently fails if it cannot find the file. Syntax errors in the file will be printed though.

Take a look at the name printed here, this is the name you need to use when you are initializing your engine in your application later. The name specified in the configuration file is only used there.

4.3 Code and documentation

In the next few sections, we'll discuss some examples of EVP code.

Using the same structure as for the PKCS examples, we'll first define some general setup and cleanup functions, then some specific signing/verification functions, and finally we'll tie it all together in a calling `main` function. At every defined function, we'll reflect a bit on why we do what we do there, and note some other areas that might be of interest to the application developer.

Before you dive in, you might want to skim the documentation for OpenSSL and EVP. While it might be to much to go trough in one sitting, we advise you at least take a look at $ENGINE(3ssl)$, $EVP\_DigestInit(3ssl)$, $EVP\_SignInit(3ssl)$, and $EVP\_EncryptInit(3ssl)$.

After you have played around for a bit, you might want to read all of these more closely. For now, let's start typing some code.

4.4 Some general functions

First of all, let's do the unavoidable initalization. As mentioned before, to make full use of OpenSSL's possibilities, we need to load a configuration that the user may have set.

```
#include <openssl/conf.h>
#include <openssl/engine.h>
#include <openssl/err.h>

void
initialize()
{
     OPENSSL_config(NULL);
     OpenSSL_add_all_digests();
     ERR_load_crypto_strings();
}
```

Since its argument is NULL, `OpenSSL_config` loads the configuration file set in the environment variable `OPENSSL_CONF`.

We also call `OpenSSL_add_all_digests`. This loads digest algorithm names, so we can later initialize them by name (see example 2, verification, and the call to `EVP_get_cipher_by_name`).

The third call here loads human-readable error messages into memory, this is useful information when something goes wrong.

Like most API's, a lot of OpenSSL methods return a status code that signifies whether the function call succeeded. OpenSSL also provides a feedback mechanism similar to `errno`, so that a user can see some more information than just an error code. Let's make a small helper function that can assist us with that.

```
void
check_ssl_rv(const char *function_name,
             const int return_value,
             const int success_value)
{
     if (return_value != success_value) {
          fprintf(stderr,
                  "Error, return value of %s was %d",
                  function_name,
                  return_value);
          fprintf(stderr, " should have been %d\n", success_value);
          fprintf(stderr, "OpenSSL error messages:\n");
          ERR_print_errors_fp(stderr);
          exit(ERR_get_error());
     }
}
```

For completeness, we do not only pass a return value we received to this function, but also the value that would signify success, just in case this is non-standard. If there is an error, we use `ERR_print_errors_fp`, which, if we have loaded the error strings earlier, will print a nice list of errors.

Finally, if there was an error, we bluntly exit to the system. Naturally, you might want to implement a more graceful error handling routine.

If we perform global initialization, we probably need some cleanup too.

```
void
clean_up()
{
     ERR_remove_state(0);
     ERR_free_strings();

     ENGINE_cleanup();
     EVP_cleanup();

     CONF_modules_finish();
     CONF_modules_free();
     CONF_modules_unload(1);

     CRYPTO_cleanup_all_ex_data();
}
```

`ERR_remove_state` frees up the error message queue for the current thread. With a non-zero argument, you can specify the thread that should have its error queue cleaned up. `ERR_free_strings` removes the strings we loaded in the initialization function.

`ENGINE_cleanup` cleans any memory that is allocated dynamically by engine functions. This happens for instance when certain engines are loaded, even if they are not actually used. See the *engine*(3) manual page for more information.

`EVP_cleanup` removes memory allocated when loading digest and cipher names in the `OpenSSL_add_all_` family of functions, like `OpenSSL_add_all_digests` we called earlier.

Next we free and unload any configuration modules that may have been loaded. The argument to `CONF_modules_unload` is 1, telling OpenSSL to unload both dynamically loaded modules and builtin modules. With this argument set to 0, it would not unload builtin modules.

`CRYPTO_cleanup_all_ex_data` cleans some more generally used memory. Documentation for this function can be hard to find. Just know that it does free some memory, and that this call is not thread-safe.

Now that we have the global tidying up sorted, let's go to some specific engine-related functions.

```
ENGINE *
select_engine(const char *engine_name)
{
     int rv;
     ENGINE *next_engine;
     ENGINE *engine = ENGINE_by_id(engine_name);
```

```
        if (!engine) {
                fprintf(stderr, "Unable to load engine: %s\n", engine_name);
                engine = ENGINE_get_first();
                fprintf(stderr, "Available engines:\n");
                while(engine) {
                        fprintf(stderr, "%s\n", ENGINE_get_id(engine));
                        next_engine = ENGINE_get_next(engine);
                        ENGINE_free(engine);
                        engine = next_engine;
                }
                exit(EXIT_FAILURE);
        }

        rv = ENGINE_init(engine);
        check_ssl_rv("ENGINE_init", rv, 1);

        return engine;
}
```

First we get a structural reference to the engine we need by calling `ENGINE_by_id` with the correct name, in our case 'pkcs11'. If this call fails, apparently the right modules are not loaded or the engine name is unknown. In that case we'll try to be bit userfriendly and print a list of available engines.

Internally, OpenSSL maintains a dynamic list of available engines. You can simply get the first of that list by calling `ENGINE_get_first`, and walk through that list by calling `ENGINE_get_next` until that return `NULL`. These functions create structural references that should be freed.

If it succeeds, we have a structural reference to our engine. However, if we want to actually do anything with it, we need to initialize it. To use the terms of the OpenSSL documentation, we need to create a functional reference to it. This is what `ENGINE_init` does. This function returns 1 on success, so we check for that.

We might want to be ready and add a little engine-specific cleanup too:

```
void
clean_engine(ENGINE *engine)
{
        ENGINE_finish(engine);
        ENGINE_free(engine);
}
```

`ENGINE_finish` removes the functional reference, and `ENGINE_free` removes the structural reference.

So, with our engine up and running, it's time to see whether it has any key data. Since EVP does not provide key-management functionality, we can skip over key enumeration and key creation. You'll need to use specific tools for that, or go straight to PKCS.

We'll just assume there is at least one key present, and load it:

```
EVP_PKEY *
get_private_key(ENGINE *engine, const char *id)
{
     EVP_PKEY *key = NULL;

     key = ENGINE_load_private_key(engine, id, UI_OpenSSL(), NULL);

     if (!key) {
          fprintf(stderr, "Error loading private key with id %s\n", id);
     }
     return key;
}
```

EVP stores its keys in `EVP_PKEY` structures. Private keys, public keys, and secret keys are all stored in this structure. We load a private key with `ENGINE_load_private_key`.

The key id is not just a simple byte string. OpenSSL gives you quite a few options here. It is a null-terminated string which can be one of the following forms:

- $< id >$
- $< slot >:< id >$
- $id\_ < id >$
- $slot\_ < slot > -id\_ < id >$
- $label\_ < label >$
- $slot\_ < slot > -label\_ < label >$

Where:

- id is a hexadecimal value representing the key id.
- slot is the slot number (an integer)
- label is the human-readable label of the key

The third argument provides a way for the application developer to specify how a user can provide the user PIN to the HSM. A nice default is `UI_OpenSSL`, which just uses the command line to enter a PIN when needed. The fourth argument is an optional callback argument for the UI function. UI_OpenSSL does not need an argument, so here it's `NULL`.

Well if we can get private keys that easily, let's just do the public counterpart as well:

```
EVP_PKEY *
get_public_key(ENGINE *engine, const char *id)
{
     EVP_PKEY *key = NULL;
     key = ENGINE_load_public_key(engine, id, UI_OpenSSL(), NULL);

     if (!key) {
          fprintf(stderr, "Error loading public key with id %s\n", id);
     }
     return key;
}
```

4.5 Example 1: Signing

So we have an engine and a private key. Now it's time to do some signing. We'll just blindly try to read from and write to the given files. As in our PKCS examples, correct handling of these files is left as an excercise to the reader.

```
void
sign_data_evp(ENGINE *engine,
              EVP_PKEY *key,
              FILE *data_file,
              FILE *signature_file)
{
     unsigned char *data;
     int data_len;

     unsigned char *sig;
     int sig_len;

     int rv;

     EVP_MD_CTX *ctx = EVP_MD_CTX_create();

     sig = malloc(EVP_PKEY_size(key));
     sig_len = EVP_PKEY_size(key);

     rv = EVP_SignInit_ex(ctx, EVP_sha1(), NULL);
     check_ssl_rv("EVP_SignInit_ex", rv, 1);

     data = malloc(1024);
     data_len = fread(data, 1, 1024, data_file);
     while (data_len > 0) {
          rv = EVP_SignUpdate(ctx, data, data_len);
          check_ssl_rv("EVP_SignUpdate", rv, 1);
          data_len = fread(data, 1, 1024, data_file);
```

```
        }

        rv = EVP_SignFinal(ctx, sig, &sig_len,  key);
        check_ssl_rv("EVP_SignFinal", rv, 1);

        if (sig_len > 0) {
             fwrite(sig, sig_len, 1, signature_file);
        }

        EVP_MD_CTX_destroy(ctx);
        free(sig);
        free(data);
}
```

To perform a signing operation, we first need to create a context. The `EVP_MD_CTX_create` function both allocates memory for that, and initializes it.

We can use `EVP_PKEY_size` function to figure out the size the signature will have.

As with most cryptographic operations, signing takes three steps; initialization, data feeding, and finalization. In `EVP_SignInit_ex` we set up the context with a digest function and possibly a specific engine. In this case, we do not specify a specific engine, OpenSSL can figure out what to use from the key we give it.

We specify that we want to use SHA1 hashing by using `EVP_sha1`, which returns the identifier for the SHA1 algorithm. In the verification method later, we'll use a different approach for this.

If the operation has been initialized successfully, we read our data file in chunks of 1024 bytes, and feed it to the signer by using `EVP_SignUpdate`.

When we are done feeding our data, we call `EVP_SignFinal`, which finalizes the operation and creates the signature.

If this worked and we have a signature, we write it to the specified signature file.

Finally, we need to clean our context, by calling `EVP_MD_CTX_destroy`.

Now to put it all together, we'll create a simple `main`:

```
int
main(int argc, char **argv)
{
        ENGINE *engine;
        const char *engine_name = "pkcs11";


        if (argc < 4) {
             printf("Usage: evp_example1 <key id>");
             printf(" <data file> <signature_file>\n");
             exit(1);
```

```
        }

        initialize();
        engine = select_engine(engine_name);

        sign_data(engine, argv[1], argv[2], argv[3]);

        clean_engine(engine);
        clean_up();

        return EXIT_SUCCESS;
}
```

That's it! Compile it with -lcrypto (or your locally needed compilation options), and run it with $OPENSSL\_CONF$ set to your configuration file, and you can create a signature.

With the signature written out, let's see whether we can verify it again.

4.6 Example 2: Verification

This example looks a lot like the previous one. Of course we will be reading a signature instead of writing it, use some other function calls, and we'll use the public instead of the private key.

```
int
verify_data_evp(ENGINE *engine,
                EVP_PKEY *key,
                FILE *data_file,
                FILE *signature_file)
{
        unsigned char *data;
        int data_len;

        unsigned char *sig;
        int sig_len;

        int result;
        int rv;

        EVP_MD_CTX *ctx = EVP_MD_CTX_create();

        const EVP_MD *md = EVP_get_digestbyname("SHA1");

        if (!md) {
                fprintf(stderr, "Error creating message digest");
                fprintf(stderr, " object, unknown name?\n");
                ERR_print_errors_fp(stderr);
```

```
        exit(1);
    }

    rv = EVP_VerifyInit_ex(ctx, md, NULL);
    check_ssl_rv("EVP_VerifyInit_ex", rv, 1);

    data = malloc(EVP_MD_size(md));
    data_len = fread(data, 1, EVP_MD_size(md), data_file);
    while (data_len > 0) {
        EVP_VerifyUpdate(ctx, data, data_len);
        data_len = fread(data, 1, 1024, data_file);
    }

    sig = malloc(512);
    sig_len = fread(sig, 1, 512, signature_file);

    result = EVP_VerifyFinal(ctx, sig, sig_len,  key);

    EVP_MD_CTX_destroy(ctx);

    free(data);
    free(sig);

    return result;
}
```

You might have noticed that the first few lines of code after the declarations are different than in the signer. We are using a different method to specify a digest algorithm here. Remember when we called `OpenSSL_add_all_digests`? This is why we did it; now we can specify a digest by its name as a string. For this example, this might not be too useful. But when you do not use a fixed algorithm, this gives your user more flexibility; it could technically even specify an algorithm that isn't known when your application was developed.

When we have a message digest structure, we use this to call `EVP_VerifyInit_ex`, which works a lot like `EVP_SignInit_ex`. After initialization, we feed it the same data as before.

Now we read the signature. Again we assume 512 bytes or less (since that is the size a signature from a 4096-bit key would have).

Then we pass the signature we read to the verification operation with `EVP_VerifyFinal`. If the signature verifies this call returns 1. If the signature is bogus it returns 0, and if it cannot perform the verification for any other reason it returns -1.

```
int
verify_data(ENGINE *engine,
            const char *key_id,
            const char *data_file_name,
```

```
            const char *signature_file_name)
{
     FILE *data_file;
     FILE *signature_file;
     EVP_PKEY *public_key = get_public_key(engine, key_id);
     int result;

     if (!public_key) {
          printf("no public key\n");
     } else {
          data_file = fopen(data_file_name, "r");
          if (!data_file) {
          fprintf(stderr, "Error opening %s: %s\n",
                  data_file_name,
                  strerror(errno));
                  exit(errno);
          }
          signature_file = fopen(signature_file_name, "r");
          if (!signature_file) {
               fprintf(stderr, "Error opening %s: %s\n",
                       signature_file_name,
                       strerror(errno));
               exit(errno);
          }

          result = verify_data_evp(engine,
                                   public_key,
                                   data_file,
                                   signature_file);

          fclose(data_file);
          fclose(signature_file);

          EVP_PKEY_free(public_key);
     }

     return result;
}
```

This is mostly the same as the `sign_data` function.

```
int
main(int argc, char **argv)
{
     int rv;
     ENGINE *engine;
```

```
        const char *engine_name = "pkcs11";

        if (argc < 4) {
             printf("Usage: evp_example2 <key id> <data file>");
             printf(" <signature_file>\n");
             exit(1);
        }

        initialize();

        engine = select_engine(engine_name);

        rv = verify_data(engine, argv[1], argv[2], argv[3]);
        check_ssl_rv("Verify data", rv, 1);

        printf("Signature verified: success\n");

        clean_engine(engine);
        clean_up();

        return EXIT_SUCCESS;
}
```

And there we have it. Using this application with the data file you used and the signature file created in example 1 should print the message 'Signature verified: success'.

Encryption and decryption using assymetric cryptography works roughly the same way, see $EVP\_EncryptInit(3SSL)$ for more documentation on functions that perform these operations.

5 Deciding whether to use PKCS #11 or EVP

So you want to create an application that supports HSMs, or you want add support for them to an existing application, should you choose to use EVP or implement PKCS directly?

While there is of course no generally best choice, and personal preferences will vary, here's our take on it.

5.1 PKCS #11

If you are building an application that is specifically meant to be used with one exact hardware module, your best option would probably be to use PKCS directly. You will have less abstraction layers, and less dependency on external libraries.

Furthermore, some modules do not implement PKCS completely, or even correctly. If you are using PKCS calls in your application, you have more flexibility to work around these limitations, assuming that with EVP, you'd be using the PKCS #11 backend. You'll also have the direct ability to add key management functionality to your application, should you desire to do so.

5.2 EVP

If you have an existing application that already uses EVP, or your application is also intended to work with pure software implementations of cryptographic operations, EVP would probably be the better choice. It is said that there are pure software implementations of PKCS #11, but EVP will give your users more flexibility to pick and choose their setup.

Also, while you do have a library dependency on OpenSSL, this library is present on most modern systems, and you have no direct dependencies on specific PKCS implementations.

A AEP Keyper

AEP offers a line of HSM products, including the Keyper Enterprise, the Keyper Professonial, and the Keyper PCI.

They have been so nice as to provide an evaluation version of the AEP Keyper. Some details:

- Works over network
- Key creation, storage and usage on the device
- FIPS-140-2 (see 1.3) level 4 certified.

More information can be found here:
`http://aepnetworks.com/products/key_management/keyper/ent_overview.aspx`

A.1 Driver

The HSM comes with a CD that includes some code samples, a few tools, and a list of drivers for specific operating systems and environments:

- Linux (gcc-2, gcc-3, and gcc-4)
- OSX (PPC-gcc-3, Intel-GCC4)
- Solaris (32-bit and 64-bit)
- Windows

At the time of this writing, there was no linux 64-bit version, nor a BSD version, but both are said to be available shortly. You might be able to get it running on those systems, but you'd probably need a complete stack of compatibiliy libraries. Getting it to run that way is out of scope for this document.

The CD also includes a lot of documentation about the installation and implementation, and what features are provided, as well as some test programs and general examples and header files.

A.2 Utilities

The drivers for each environment come with two tools:

- **inittoken** This can be used to initialize the token, and set a user PIN. When this is run, all generated keys are deleted.
- **test_drv** This is a test tool that can test the machine and your setup.

A.3 Using the driver

If you want to use the tools provided, you only need to place the driver library in you library path (for example LD_LIBRARY_PATH in linux, DYLD_LIBRARY_PATH in OSX), and run the program.

Since this module works over a network, you'll also need to provide its IP address. You do this by specifying it for the name HSM in /etc/hosts (or your local equivalent).

A quick way to use other tools that talk PKCS #11 in Linux is to use the LD_PRELOAD environment variable, thereby overriding any pkcs11 library installed on the system.

Of course you can link against the library when you are compiling your own software.

A.4 Status

Once configured, the driver and the device both seem to run very well. I have been able to do at least all operations shown in this document with the device. Initializing it involves some steps; for security reasons, you'll have to assign Security Officer smart cards, and use them to set the device to Operational, but it is pretty straightforward.

A.5 Caveats and common error messages

An error you might come across when you're just starting out is CKR_DEVICE_ERROR. The first reason that can cause this error is that the device could not be reached over the network, so check whether the HSM value is set in /etc/hosts, whether the address is right and it is operational. In short, when you get this error, check your setup and configuration.

B Eutron ITSEC CryptoIdentity USB-token

Some details:

- USB token (smartcard and reader in one)

Product page:
`http://www.eutronsec.com/infosecurity/Contents/ProductLine/Details.aspx?IDProd=15&IDFamiglia=2`

More information can be found here:
`http://www.opensc-project.org/openct/wiki/eutron`

B.1 Driver

The supported token can be used with the drivers in OpenCT, and should be readable out of the box. However, due to what may be an incompatibility with PKCS in the eutron driver, newer versions of OpenSC might fail to connect to the token.

B.2 Utilities

Since this device works with OpenCT drivers, all tools provided by OpenCT and OpenSC should work with it.

B.3 Status

The only product from the line of CryptoIdentity tokens that works with OpenCT is the ITSEC (the green one). The two other devices are not supported at all.

The device is already in operational state when it is delivered. This means that you do not need to initalize it, but unfortunately, you cannot re-initialize it either. It does not support key removal, or at least our version claims not to, so usage might be a bit limited.

For more information, go to:
`http://www.opensc-project.org/opensc/wiki/CryptoIdentityItsec`

The key present does work, and does what it must do.

B.4 Caveats and common error messages

The card that is used does not support keys for both signing and encryption, but OpenSC has a workaround for that (–key-split).