A Brief Guide to OpenSSL: Part 1

1 Introduction

1.1 What is OpenSSL?

From the OpenSSL Web page:

> The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. The project is managed by a worldwide community of volunteers that use the Internet to communicate, plan, and develop the OpenSSL toolkit and its related documentation.
>
> OpenSSL is based on the excellent SSLeay library developed by Eric A. Young and Tim J. Hudson. The OpenSSL toolkit is licensed under an Apache-style license, which basically means that you are free to get and use it for commercial and non-commercial purposes subject to some simple license conditions.

In short, OpenSSL is a software library that provides a full-featured cryptographic toolkit as well as an implementation of SSL. The Unix version, which we will be using, provides two libraries, one for SSL/TLS (`ssl`) and one for the cryptographic toolkit (`crypto`) as well as command-line tool (`openssl`).

1.2 Getting More Information

In addition to the documentation in this handout, there are a number of other sources of information on using OpenSSL you may wish to consider.

1.2.1 The Web

The OpenSSL Project's Web page, `<http://www.openssl.org/>`, contains HTML versions of much of the OpenSSL documentation, as well as links to FAQs and other documents available on the Web about using OpenSSL. You can also download your own copy of OpenSSL and its source code, if you so desire.

Since OpenSSL's documentation of parts of the Crypto library are lacking, Ariel Glenn's documentation of SSLeay (the precursor to OpenSSL) at `<http://www.columbia.edu/~ariel/ssleay/>` may be of particular help.

1.2.2 Man Pages

Most of OpenSSL's library interfaces have Unix manual pages, accessible via the Unix `man` program. If you are using a Leland Systems cluster computer, however, you should be aware that these manual pages are located in a non-standard location. If you want to access the man page for `crypto`, for example, which provides an overview of the OpenSSL Crypto library, you should use the following command:

```
> man -M /usr/pubsw/apps/openssl/man crypto
```

If you find yourself accessing OpenSSL manual pages on a regular basis, you may wish to define a shell alias to save typing.

1.2.3 Header Files

Unfortunately, OpenSSL's documentation is not as comprehensive as the library itself. If you cannot find information about a particular function in the OpenSSL documentation, inspecting the C header files may prove useful. These are installed on the Leland cluster computers in `/usr/pubsw/include/openssl`. For example, to see the interface declarations for all the functions related to MD5, you might look in `md5.h`.

1.2.4 Source Code

Since OpenSSL is an open source project, the source code is freely available and you may find it interesting or useful to peruse it. However, you should not need to go to this much effort for the CS 255 projects. If you are having trouble finding documentation on a particular feature of OpenSSL, please feel free to contact the course staff.

2 Programming With The OpenSSL Crypto Library

This section contains some general information on using the OpenSSL Crypto library that may be of use before examining the rest of the library.

2.1 Error Checking

Most OpenSSL functions use their return value to indicate whether the call succeeded or not. Many functions return 1 for success and 0 for failure. Others that return a pointer to a structure will return `NULL` upon failure. You should always check each OpenSSL function to see if it has succeeded. If an OpenSSL function fails, an error code is stored in a queue kept by the library.

To retrieve this error code in numeric form, you can use the *ERR_get_error()* function. For example:

```
unsigned long error_code;
if (SSL_function() == 0) {
```

```
    error_code = ERR_get_error();
}
```

Once you have retrieved the error, you can copy it into a string using the *ERR_error_string()* function:

```
char buf[120];
ERR_error_string(error_code, &buffer);  /* buf now describes the error */
```

If you are interested in the error codes only for debugging purposes, you may find it simpler to simply use the following function to print out all errors currently in the error queue:

```
ERR_print_errors_fd(stderr);
```

For more information on OpenSSL error codes, see the *err* man page.

2.2 Generating Randomness

Often you will wish to have a source of random bits, for example, to generate a new key. OpenSSL provides a cryptographically secure pseudo-random number generator for this purpose.

Once you have seeded the random number generator (more on this later), obtaining randomness is straightforward, using the *RAND_bytes()* function. For example, copying 16 bytes (128 bits) of randomness into an array:

```
char buffer[16];
if (RAND_bytes(buffer, 16) == 1) {
    /* use randomness */
}
```

The important caveat is that OpenSSL's PRNG must have a random source of data from which obtain randomness. Some operating systems (e.g., Linux) have a built-in source of randomness, and OpenSSL will transparently use this if available. However, the Leland cluster machines run Solaris 8, which does not provide a built-in source of randomness. Therefore, OpenSSL's random number generator will fail unless your application first provides a random seed using the *RAND_seed* function.

For the CS 255 projects, however, you do not have to worry about this. The starter code provided seeds the OpenSSL random number generator for you. It uses the current date and time and some statistics about the program's resource usage, which are not truly random but will be a good enough substitute for your project.

For more information on OpenSSL's random number generation, see the *rand(3)* man page.

3 Private Key Cryptography

3.1 Symmetric Ciphers

Symmetric encryption in OpenSSL is done using a library called EVP, which provides a high-level interface to the cryptographic functions. It abstracts from you the details of a particular algorithm, so you can interchange algorithms while still using the same code. OpenSSL includes support for a wide range of ciphers, including DES, IDEA, RC4, and others.

EVP uses a data type called `EVP_CIPHER_CTX` to keep track of what cipher you are using, how much data you have encrypted, the current IV, etc... You will pass a pointer to this context to each EVP function that is dealing with the same encryption or decryption.

3.1.1 Initializing a Cipher

To initialize a cipher context for a particular cipher and mode, key and IV, use *EVP_EncryptInit()*. For example:

```
EVP_CIPHER_CTX ctx;
char key[8] = { 34, 56, 78, 00, 24, 12, 111, 24 };  /* a "random" key */
char iv[8] = { 12, 55, 124, 99, 201, 37, 12, 177 };
int success;

success = EVP_EncryptInit(&ctx, EVP_des_cbc(), key, iv);
```

Here, we initialized our context for a particular key and IV for using single-DES to encrypt in CBC mode. The second argument, a pointer to an `EVP_CIPHER` type, tells EVP what cipher and mode to use. These types are obtained by calling the appropriate function. See the *EVP_EncryptInit* man page for a complete listing of cipher types.

Here, we passed in an explicit 64-bit IV and 64-bit key. Note that the DES implementation requires an 8-byte key, even though it only uses seven of the bytes. To find out the requires key and IV lengths, you can use the *EVP_CIPHER_key_length()* and *EVP_CIPHER_iv_length()* functions, respectively, passing in a cipher type. Note that some algorithms (e.g., stream ciphers) do not require an IV; in that case, you may pass in NULL. You may also find useful *EVP_CIPHER_block_size()*, which returns the cipher's block size.

3.1.2 Using a Cipher

To start encrypting data, use the *EVP_EncryptUpdate()* function. For example, to encrypt a 32-byte block using an already-initialized DES cipher context:

```
char plaintext[32] = { ... };
int plaintext_length = 32;
char ciphertext[39];
int ciphertext_length = 39;
```

```
success = EVP_EncryptUpdate(&ctx, ciphertext, &ciphertext_length,
                            plaintext, plaintext_length);
```

This will add the data from `plaintext` to the encryption, and put some ciphertext into `ciphertext`. Some important things to note:

- We passed a pointer to the size of our output buffer. EVP will modify this to reflect how many bytes were actually inserted. Depending on the mode of encryption and algorithm, not all of the ciphertext corresponding to a given update will be generated in that same call. Some may be encoded in a later update.
- For this reason, the ciphertext may be more or less in size than the plaintext. The output buffer needs to be as large as the input, plus ($cipher\_block\_size - 1$) bytes. In our case, since DES uses 8-byte blocks, we need an extra 7 bytes. **Make sure you have enough room**.

Once you are done with your encryption, you need to call the *EVP_EncryptFinal()* function. This will add the appropriate padding to your message and output the final block:

```
char ciphertext[8];
int ciphertext_length = 8;

success = EVP_EncryptFinal(ciphertext, &ciphertext_length);
```

Note that although we did not provide any new input, we still needed to provide enough output space for one ciphertext block.

After calling *EVP_EncryptFinal()*, you should not make any more update calls. Once you are done with the encryption or decryption, you should call *EVP_CIPHER_CTX_cleanup()* to clean up the cipher context and remove any sensitive information from memory:

```
success = EVP_CIPHER_CTX_cleanup(ctx)
```

3.1.3 Encryption vs. Decryption

The above examples show how to use a cipher for encryption. Decryption is accomplished in exactly the same way, except you will use *EVP_DecryptInit()*, *EVP_DecryptUpdate()*, *EVP_DecryptFinal()*. These work almost identically to the encryption versions, with one caveat: The output buffer passed to *EVP_DecryptUpdate()* should have room for the input, plus a complete cipher block [1].

3.2 Message Digest (Hash) Functions

Digest functions are, like symmetric ciphers, provided by the EVP high-level library. OpenSSL provides several digest algorithms, including MD5 and SHA-1. Using EVP for hashing works very similarly to encryption. The main difference is that there is no ciphertext generated, only a digest.

[1] Exception to the exception: When using a cipher with block size of 1 byte, it is sufficient to have output and input buffers of equal size.

Another difference is that none of the EVP functions have return values. This is because, unlike a cipher, it is not possible for a hash function to fail. This means you do not have to check return values from the EVP digest functions.

3.2.1 Initializing a Digest

When using a digest function, EVP requires a context structure, just as with a cipher. Here, that function is of type EVP_MD_CTX. We initialize this context using the *EVP_MD_CTX()* function. For example:

```
EVP_MD_CTX ctx;

EVP_DigestInit(&ctx, EVP_sha1());
```

This initializes the context to use the SHA-1 hashing algorithm.

3.3 Using a Digest

As with a cipher, you update the digest with each block of data you want to add to the digest. You can pass in as much data as you want, but it may be advantageous to split it up into smaller chunks, depending on the needs of your program.

To add data to the digest, use the *EVP_DigestUpdate()* function:

```
char *data = ...;
int bytes_in_data;

EVP_DigestUpdate(&ctx, data, bytes_in_data);
```

Once you have called the update function with the data to be hashed, you can call *EVP_DigestFinal()* to retrieve the digest:

```
char digest[20];
int digest_length = 20;

EVP_DigestFinal(&ctx, digest, &digest_length);
```

Similar to encryption and decryption, we pass in a pointer to the length of our buffer, which will be filled in with the actual length of the digest. Here, we used a 20-byte buffer because SHA-1 has a digest size of 160 bits. To find the length of a given digest algorithm, you can use the *EVP_MD_size()* function.

Once you have finalized a digest context, you cannot use it again without first calling *EVP_DigestInit()*. However, since hashes are not generally considered sensitive data, EVP does not provide a function to clear the context, and it is not necessary to do so.

Note that OpenSSL provides no functions to compare two hashes to see if they are the same. However, since a message digest is simply an array of bytes, we can compare it with standard C functions such as *memcmp()*.

3.4 Message Authentication Codes (MACs)

Unlike symmetric ciphers and message digests, OpenSSL does not provide a high-level interface for MACs. Instead, we will use the direct interface for OpenSSL's implementation of the HMAC algorithm.

Note that HMAC functions, like the EVP message digest functions, do not have return values.

3.4.1 HMAC Initialization

As with ciphers and digests, we will use a context pointer for keeping track of the HMAC algorithm's state. The type for that pointer is `HMAC_CTX` and we initialize it using the *HMAC_Init()* function:

```
HMAC_CTX ctx;
char *key = ... ;
int keylength = ...;

HMAC_Init(&ctx, key, keylength, EVP_sha1());
```

To initialize the HMAC context, we pass in a key, which may be of any length, and a type of hash algorithm to use in the HMAC construction. This may be any EVP hash algorithm, as described above.

3.4.2 Using HMAC

Once the HMAC context has been initialized, it works exactly like a digest does:

```
char mac[20];
int mac_length = 20;

/* repeat as necessary... */each time.
HMAC_Update(&ctx, some_data, data_length);

HMAC_Final(&ctx, &mac, &mac_length);
```

This will add all the data and finally compute the MAC. Since the HMAC construction is based on the use of a hash function, the output buffer must be as large as the size of the hash function used to initialize the HMAC context. In our example, since we used the 160-bit hash function SHA-1, we need at least 20 bytes of output buffer.

3.4.3 Cleaning up an HMAC

Since the HMAC context contains sensitive information (the key), you should be sure to clean it up after you are finished:

```
HMAC_cleanup(&ctx);
```

This erases the key and any other data from the `HMAC_CTX`.

3.5 Password-Based Encryption (PBE)

It is often useful to be able to generate keys for symmetric algorithms from a user-entered password. OpenSSL provides several functions to help achieve this goal.

3.5.1 Reading a Password From the User

OpenSSL provides the function *EVP_read_pw_string()* to read a password from standard input. This is useful for a command-line Unix program to read a password from the user. Here is an example:

```
char password[32];
int error;

error = EVP_read_pw_string(password, sizeof(password),
    "Enter the new password: ", 1);
```

This will prompt the user "Enter the new password:" and wait for them to enter it. Setting the fourth argument to 1 makes the user type the password twice, to verify it. Set the argument to 0 if they need only type the password once. Generally, you will want the user to verify a password when they are setting a new one, but not when they are entering an already-chosen password.

If successful, the password will be placed in the given buffer as a null-terminated C string. Note that unlike most other OpenSSL functions, *EVP_read_pw_string()* returns 0 for success, not for failure.

3.5.2 Converting a Password to a Key

If you want to use a human-entered password as the key to a symmetric cipher, you should not simply use it directly. OpenSSL provides a function that uses the PKCS#5 algorithm for converting a password to a key using a hash function and salt. Here is an example:

```
char *password = "my password";
char salt[8] = { 0xc7, 0x73, 0x21, 0x8c, 0x7e, 0xc8, 0xee, 0x99 };
int iteration_count = 20;
char key[8];
```

```
char iv[8];
int success

success = EVP_BytesToKey(EVP_des_cbc(), EVP_sha1(), salt, password,
                         strlen(password), iteration_count, key, iv);
```

Some notes on using this function:

- This function generates a random key and IV for the specific symmetric cipher given as the first argument. Be sure the key and IV arguments are of the appropriate length. If you do not need to generate a random IV, you may pass in NULL.
- The PKCS#5 algorithm uses an optional 8-byte salt to add strength to the password. If you do not wish to store a salt with your password, you may use a fixed salt, or simply pass NULL for this argument.
- The iteration count specifies how many times to run the hash algorithm. A higher number may slow down an adversary. However, for most purposes the number chosen is not relevant, and 1 may be perfectly acceptable (RSA Laboratories recommends a minimum of 1000 iterations). This number should be fixed in your program, or you will not generate the same key from a password each time.