A Brief Guide to OpenSSL: Part 2

4 The OpenSSL Command Line Program

In addition to programming libraries, OpenSSL provides a command-line program for using various functions of the OpenSSL crypto library from the shell. This program, invoked as `openssl` can be used to manipulate most of the constructs and use most of the functionality of the crypto library, but we will only discuss a few of those functions here. For a complete list of possible commands, see the *openssl* man page.

4.1 General `openssl` Usage

Each function is invoked as an argument to the `openssl` command-line tool. For example, the command to manipulate RSA keys would be `openssl rsa`. Each command has different options, but most use the same syntax for dealing with input and output files. In general, you can specify the input to a command with the `-in` option, and the output with `-out`. If you do not specify, the program will try to read from standard input and write to standard output. If the input file is encrypted, the program will automatically prompt for the password.

In many cases, you will not be interested in generating new output, but simply in looking at an existing file. In this case, you can use `-noout` to suppress the output. Another useful option supported by most of the commands is `-text`, which displays the input file's contents in human-readable form, instead of the encoded binary format normally used. For example, to examine the parameters of an RSA public key in the file `key.pem`, you might use the following command:

```
openssl rsa -in key.pem -noout -text
```

4.2 RSA Data Management: `rsa`

The `rsa` command processes RSA keys. It can translate between various forms, but its main use is to view the keys in human-readable form, or to add or remove encryption of the keys.

The filename specified in the `-out` argument will, by default, not be encrypted, so you can use this command to remove the password from an RSA private key file if necessary. You can also add `-des`, `-des3` or `-idea` to prompt for a pass phrase and encrypt the output using DES, triple-DES or IDEA, respectively.

You can also use the `-check` option to verify an RSA private key; it will make sure that all the elements are consistent.

Normally, the `rsa` command works with RSA private keys. If you wish to input or output an RSA public key, give the `-pubin` or `-putout` option, appropriately. For example, to input an RSA private key from `priv.key` and place the public key in `pub.key`, you could use this command:

```
openssl rsa -in priv.key -pubout -out pub.key
```

4.3 X.509 Certificate Signing Request Management: `req`

X.509 is the standard format for storing certificates and certificate requests used by SSL. The `req` program can generate and inspect X.509 certificate requests.

Its main function, other than viewing existing certificate requests, is to generate new ones. For example, if you were setting up a new SSL server, you would generate a certificate request and send it to the Certificate Authority (CA) to be signed.

To generate a new certificate request, you can use a command line the following:

```
openssl req -new -out certreq.pem -keyout certkey.pem
```

This will prompt for all the necessary information to store in the certificate request, and store the new certificate request in `certreq.pem`. The corresponding private key will be stored in `certkey.pem`, encrypted using a pass phrase that the program will prompt for. The certificate request can now be sent to the CA to be signed.

4.4 X.509 Certificate Data Management: `x509`

The `x509` command is used to manage X.509 certificates. With this command, you can view certificates and display their various information. In addition to the `-text` option, you can supply other options to view individual pieces of information about the certificate, e.g., `-serial` to output the serial number. See the *x509* man page for a complete list.

The main other use for the `x509` command is as a CA. It can be used to sign certificates and requests. It will treat the input as an X.509 certificate request instead of a certificate when given the `-req` option, and can output a signed certificate which can then be used for any application that requires a certificate.

4.4.1 Self-Signing Certificates

Self-signing is signing a certificate with itself. This is generally done only for the root CA (who has no one else to sign its certificate), or for test certificates. To do this, use the `-signkey` option to supply a private key. For example, to self-sign the certificate request generated in the previous section:

```
openssl x509 -in certreq.pem -req -out cert.pem -signkey certkey.pem -days 30
```

This will self-sign the certificate, with an expiration date 30 days in the future (the `-days` option specifies the expiration). The new self-signed certificate will be available in `cert.pem`

4.4.2 Signing Using a CA

If you want to use the `x509` command as a mini-CA, you need to give it the CA certificate and key to use to sign the certificate. For example, if the CA certificate is stored in `cacert.pem` and the CA key in `cakey.pem`, we could sign our certificate request using the CA key as follows:

```
openssl x509 -in certreq.pem -req -out cert.pem -CA cacert.pem
    -CAkey cakey.pem -CAserial cacert.srl -CAcreateserial -days 365
```

This example introduces several new options: The `-CA` and `-CAkey` options specify the CA certificate and public key files, respectively. The other options deal with serial numbers; every certificate issued by a CA needs to have a serial number, and OpenSSL uses serial number files to keep track of the current serial number for a given CA certificate. The `-CAserial` option identifies the file used to store the next serial number in sequence, and the `-CAcreateserial` option tells OpenSSL to create the file (with serial number 1) if it does not already exist.

Note that we did not require the private key corresponding to the request to sign it. The only private key we needed was the one for the CA.

Although all our examples have shown signing a certificate request, we could also have signed standard X.509 certificates as well by leaving out the `-req` option. Since certificates already have a signature, signing them replaces the existing signature with the new one.

4.5 X.509 Certificate Verification: `verify`

Once an X.509 certificate has been signed, we can verify that the signature is valid. This is useful when testing a CA, or to see if a certificate received from another party has been signed by a known entity. The `verify` command can be used as follows:

```
openssl verify -CAfile cacert.pem cert.pem
```

This will try to verify that the certificate contained in `cert.pem` was signed by the certificate given in `cacert.pem`. The result will be printed to standard output.

For more information about certificate verification, see the *verify* man page.

5 Public Key Cryptography

5.1 General Functionality

The OpenSSL cryptography library provides a number of functions that, while not directly related to public key cryptography, are nonetheless extremely useful when dealing with them. A few of these functions are discussed here.

5.1.1 PEM Functions

PEM is a format for encoding cryptographic objects. Short for Privacy Enhanced Mail, it was originally designed for PGP, but it has become a standard format for storing binary cryptographic data on disk. It is the standard format used for public and private keys, certificates, etc... Files ending with a `.pem` extension are in PEM format.

OpenSSL provides functions for converting the most common objects from PEM to the OpenSSL internal structures, and back again. For example, to read an RSA private key:

```
FILE *fp = fopen("privatekey.pem", "r");
RSA *rsa;

rsa = PEM_read_RSAPrivateKey(fp, NULL, NULL, NULL);
```

This will read and decode a PEM-formatted RSA private key from any open `FILE` pointer that supports reading. If the file cannot be read or is in the wrong format, the function will return NULL. The error code will be available using the standard OpenSSL error mechanism.

Note that in addition to the file pointer, we passed three NULL pointers. The second argument allows you to specify storage space for the returned object. In our case, we want OpenSSL to allocate its own storage, so we pass in NULL. The third and fourth arguments allow you to specify a callback in case the PEM routines need a password to open an encrypted file. A NULL value for these arguments specifies the default callback, which prompts for the password on the command line. In general, this is the correct behavior for a command-line program, and you will not need to change it. If you want to alter the prompt text, you can do this with the *EVP_set_pw_prompt()* function, which takes a single C string argument.

The PEM library can also be used for writing an object to disk. For example, to write an X.509 certificate:

```
X509 *certificate = ... ;    /* an allocated X509 structure */
FILE *fp = ... ;      /* an open FILE pointer that supports writing */
int success;

success = PEM_write_X509(fp, certificate);
```

This function returns 1 if the object was successfully written, 0 if there was an error.

There are equivalent PEM read and write functions for most of the objects that can be read or written from disk. They all work the same, and vary only in name and the type returned or accepted. In addition to the above examples, PEM includes operations on `X509_REQ`, `RSAPublicKey` and `DSAPublicKey` objects. For a complete list, see the `pem.h` header file.

5.1.2 ASN.1 Functions

ASN.1 (Abstract Syntax Notation 1) is a description format that OpenSSL uses to represent some of the objects it represents. Many of the functions and structures are just wrappers around ASN.1

manipulations, and sometimes it is necessary to operate on the ASN.1 objects directly. The format is fairly complex, but normally it is not necessary to deal with it directly, so only a few of the basic data types are covered here.

ASN.1 Strings Strings are represented by pointers to an `ASN1_STRING` structure, which generally will be returned by some other function. An ASN.1 string can contain arbitrary data, not just text. To retrieve the data in an `ASN1_STRING` object, you can use the following functions:

```
ASN1_STRING *string = ... ;

/* Returns the length of the string */
int length = ASN1_STRING_length(string);

/* Returns a pointer to the string's data */
unsigned char *data = ASN1_STRING_data(string);
```

You can also set new data to be contained in the string:

```
ASN1_STRING *string = ... ;
void *data = ... ;
int data_length = ... ;    /* (in bytes) */
int success;

success = ASN1_STRING_set(string, data, data_length);
```

ASN.1 Integers Another common ASN.1 data type that needs to be manipulated directly is the `ASN1_INTEGER` data structure, which represents an integer. The `ASN1_INTEGER` structure is actually the same as `ASN1_STRING`, and you can use the string operations on an integer, but there are more convenient functions for doing integer manipulation directly.

This integer can be of arbitrary size, but covered here are only those functions for dealing with integers in the scale of a C long integer. For larger numbers, the OpenSSL BN (big number) functions must be used, which are not covered in this guide. See the *bn* man page for more details.

To retrieve the value from an `ASN1_INTEGER`, we can use the *ASN1_INTEGER_get()* function, as follows:

```
ASN1_INTEGER *integer = ... ;
long value;

value = ASN1_INTEGER_get(integer);
```

Likewise, we can use the *ASN1_INTEGER_set()* function to change the value:

```
ASN1_INTEGER *integer = ... ;
long newvalue = ... ;
```

```
int success;

success = ASN1_INTEGER_set(integer, newvalue);
```

ASN.1 Times The third type of ASN.1 object you may encounter is the `ASN1_TIME` type, used to represent a date and time. This type is also actually just `ASN1_STRING`, but OpenSSL provides special functions to set (but not retrieve) the time in this structure. For example, create an `ASN1_TIME` object and set it to the current time:

```
ASN1_TIME *time;
time_t now;

time = ASN1_TIME_set(NULL, now);
```

5.2 Public Key Ciphers

In addition to the symmetric ciphers covered earlier, the OpenSSL crypto library provides comprehensive support for public key encryption using RSA and public key signatures using RSA or DSA.

5.2.1 Keys

OpenSSL uses two types of key structures for public key ciphers. First, it uses specific structures to store keys for a specific algorithm, such as RSA or DSA. Second, it can use the more general `EVP_PKEY` structure to store keys for any algorithm. We will use the algorithm-specific structures when generating new keys or loading them from disk (see the PEM routines above), and the EVP structure when doing actual public key operations.

In both cases, OpenSSL does not use separate structures for private keys and public keys, but uses a single structure to store all the known information about a key pair. If all that is available is the public key, then the private key elements of that structure will be left blank, and functions that rely on the private key will fail.

RSA Keys An RSA public or private key is stored in the `RSA` structure. There is an example above of loading an RSA private key from disk. We can also create a new, empty, RSA key using the *RSA_new()* function. More useful is generating a new random RSA key. For example, we could use the following code to generate a new 1024-bit RSA private key:

```
RSA *private_key;

private_key = RSA_generate_key(1024, 65535, NULL, NULL);
```

The first argument (1024), specifies the number of bits in the modulus, and the second argument the public exponent. 65535 is a generally accepted good value here. The third and fourth parameters, if non-NULL provide a callback function that will be used to display feedback about the progress of key generation, which can be slow. For more information, see the *RSA_generate_key* man page.

DSA Keys A DSA public or private key is stored in the `DSA` structure. Like RSA keys, a DSA key can be read from disk, or generated randomly. Here is some code to generate a DSA private key:

```
DSA *private_key;
int success = 0;

private_key = DSA_generate_parameters(1024, NULL, 0, NULL, NULL, NULL, NULL);
success = (private_key != NULL);

if (success) {
    success = DSA_generate_key(private_key)
}
```

As is evident, DSA key generation is a bit more complex than RSA key generation. Here, we generated a random key with 1024-bit primes. The other arguments to *DSA_generate_parameters()*, if non-NULL, can be used to get more information about the generated parameters, specify a seed for the randomly-generated primes, or specify a callback to provide feedback. For more information, see the *DSA_generate_parameters* man page.

EVP Keys The EVP library has its own structure, `EVP_PKEY` for storing keys for public key algorithms. These structures are generated using *EVP_new()*, but their values are usually set from an algorithm-specific key. For example, to create a new EVP key from an RSA key:

```
RSA *rsa_key = ... ;
EVP_PKEY *pkey;
int success = 0;

pkey = EVP_PKEY_new();
if (pkey != NULL) {
    success = EVP_PKEY_assign_RSA(pkey, rsa_key);
}
```

There is a similar function, *EVP_PKEY_assign_DSA()*, that can be used for DSA keys.

There are a number of functions that can be used to get general information from the `EVP_PKEY` structure. For example, *EVP_PKEY_bits()* will return the size of the private key, in bits. The *EVP_PKEY_size()* returns the number of bytes that a public key operation will output (e.g., for RSA this is equal to the size of the modulus).

Once the EVP key has been used for the last time, the structure can be deallocated using the *EVP_PKEY_free()* function. This will also deallocate the algorithm-specific key stored within.

5.2.2 Encrypting (Sealing)

Public key encryption is done using the EVP library, very similar to how it was done for symmetric key encryption. EVP uses the same data type as before, `EVP_CIPHER_CTX`, to keep track of the

current encryption state between calls.

Initialization Since public key encryption is slow, the EVP libraries actually do encryption using a symmetric cipher, and use the public key algorithm only to encrypt the symmetric cipher key. So the initialization function *EVP_SealInit()* takes a symmetric cipher type as well as the public key structure (which contains information on the public key algorithm). In fact, EVP supports encrypting a message with more than one public key, so that the holder of any of the corresponding private keys can read the message.

Here is an example of how to initialize the `EVP_CIPHER_CTX` for encrypting a single public key:

```
EVP_PKEY *public_key = ... ;        /* RSA public key */
char encrypted_key[128];
int encrypted_key_len;
char iv[8] = ... ;                  /* random IV */
EVP_CIPHER_CTX ctx;
int success;

success = EVP_SealInit(&ctx, EVP_des_cbc(), &encrypted_key,
                       &encrypted_key_length, iv, &public_key, 1);
```

Since EVP will be using symmetric encryption for the message, we need to provide a cipher type (here, DES in CBC mode) and IV, although no secret key. Unlike *EVP_EncryptInit()*, public key initialization requires a buffer. This buffer stores the encrypted symmetric key. It must have room for a value encrypted with the given key; this size can be obtained using *EVP_PKEY_size()*. The actual number of bytes used is stored in the `encrypted_key_length` variable.

Once the cipher has been initialized, the actual encryption of the data works exactly as it does in the symmetric case, except for the use of *EVP_SealUpdate()* and *EVP_SealFinal()* instead of *EVP_EncryptUpdate()* and *EVP_EncryptFinal()*. The functions have the exact same properties and calling conventions.

5.2.3 Decryption (Opening)

Decryption works similarly to encryption, except that only one (private) key is required to decrypt. We might initialize an `EVP_CIPHER_CTX` structure for decrypting the message encoded above as follows:

```
EVP_PKEY *private_key = ... ;     /* RSA private key */
char *encrypted_key = ... ;       /* from encryption */
int encrypted_key_length;
char iv[8] = ... ;
EVP_CIPHER_CTX ctx;
int success;

success = EVP_OpenInit(&ctx, EVP_des_cbc(), encrypted_key,
                       encrypted_key_length, iv, private_key);
```

Once the context has been initialized and the encrypted key decrypted, the message can be decrypted using *EVP_OpenUpdate()* and *EVP_OpenFinal()*. These functions behave exactly like *EVP_DecryptUpdate()* and *EVP_DecryptFinal()*.

5.2.4 Signing

EVP also provides routines for using RSA or DSA private keys to sign a message. These routines are very similar to the EVP routines for generating digest functions, since what is actually signed is a hash of the message, not the message itself. We will use the `EVP_MD_CTX` context, as we did for the hash functions.

Here is a complete example for hashing a message:

```
char *message = ... ;
int message_length = ... ;
EVP_PKEY *private_key = ... ;   /* RSA private key */
char signature[128];
int signature_length;
EVP_MD_CTX ctx;
int success;

EVP_SignInit(&ctx, EVP_sha1());
EVP_SignUpdate(&ctx, message, message_length);
success = EVP_SignFinal(&ctx, signature, &signature_length, private_key);
```

It is evident that initialization and updating work exactly as they do for digests. The arguments to *EVP_SignFinal()* include the private key to be used for signing, as well as a buffer to output the signature. This buffer must be large enough to store a signature generated by the private key; this value is obtained by calling *EVP_PKEY_size()*. The actual length of the returned signature is stored in `signature_length`.

Note that the message digest algorithm provided to *EVP_SignInit()* is associated directly with a signature algorithm. For example, if we are using RSA keys, we can use *EVP_sha1()* as our digest algorithm, but if we are using DSA keys, we must use *EVP_dss1()* instead (this is actually the same algorithm, but a different implementation). See the *EVP_DigestInit* man page for a complete list of hash functions and associated signature algorithms.

5.2.5 Verification

Verifying a signature using EVP is almost identical to generating one; it computes the hash of a message and then verify the signature. Instead of generating a signature, it takes one as an argument, and instead of a private key, it needs only a public key. We could verify the signature generated above as follows:

```
EVP_PKEY *public_key;
```

```
EVP_VerifyInit(&ctx, EVP_sha1());
EVP_VerifyUpdate(&ctx, message, message_length);
success = EVP_VerifyFinal(&ctx, signature, signature_length, public_key);
```

If the signature is correct, *EVP_VerifyFinal()* will return 1. If the signature failed, it will return 0, or -1 if some other error occurred.

5.3 X.509 Certificates

An X.509 certificate is represented in OpenSSL with the `X509` structure. It is most commonly read from disk or retrieved from the SSL library, but can also be generated anew with the *X509_new()* function.

The rest of this section describes how to retrieve and modify information stored in an X.509 certificate.

5.3.1 Public Key

The major function of an X.509 certificate is to store a public key. It can be retrieved using the *X509_get_pubkey()* function, as follows:

```
X509 *certificate = ... ;
EVP_PKEY *pubkey;

pubkey = X509_get_pubkey(certificate);
```

A new public key can be installed in the certificate using the *X509_set_pubkey()* function:

```
int success = X509_set_pubkey(certificate, pubkey);
```

5.3.2 X.509 Names

An X.509 certificate contains two "names". An X.509 name is a set of textual data designed to uniquely represent the holder of a certificate. It is important that these names actually be unique, especially for CA certificates, since most SSL implementations will use names to look up certificates in their certificate store. If two certificates have the same name, it is likely the wrong one will be found and SSL will fail.

An X.509 name is represented in OpenSSL by the `X509_NAME` type. New names can be created with the *X509_NAME_new()* function call, or they can be extracted from existing certificates, described below.

Names contain various kinds of information. Each type is identified by a unique NID (a type of ASN.1 identifier). The NIDs that are commonly used in certificates, and their OpenSSL names, are as follows:

`NID_countryName` The two-letter code for the country the named entity is located in, e.g., "US".

`NID_stateOrProvinceName` e.g., "California"

`NID_localityName` Usually the city, e.g., "Stanford"

`NID_organizationName` e.g., "Stanford University"

`NID_organizationalUnitName` Optional, specifies a group or department within the organization.

`NID_commonName` The common name is the name by which the entity named is known to the world. For a client certificate, this might be the username or real name of the user. For an SSL server certificate, this must be the DNS hostname of the server; SSL clients will check this name against the server hostname to make sure they are connecting to the right host.

`NID_pkcs7_emailAddress` An optional field that specifies an email contact address for the named entity.

Where possible, you should add information fields in the above order, since that is how it will be expected. In general, it is always better to replace the name rather to try and modify it, since OpenSSL cannot remove information from a name, but only add to it.

Getting Data From A Name To retrieve from an X.509 name the string associated with a particular field, use the *X509_NAME_get_text_by_NID()* function. This copies the text into a C string buffer. For example, to get the common name from a certificate:

```
X509_NAME *name = ... ;
char commonName[256];
int success;

success = X509_NAME_get_text_by_NID(name, NID_commonName, commonName, 256);
```

This copies the text corresponding to the NID, if it exists, to the buffer provided.

Setting Data In A Name To set data in an X.509 name from a C string, you can use the *X509_NAME_add_entry_by_NID()* function. For example, to create a new name with only a country:

```
X509_NAME *name;
char *countryName = "US";
int success = 0;

name = X509_NAME_new();
if (name != NULL) {
    success = X509_NAME_add_entry_by_NID(name, NID_countryName,
                  V_ASN1_APP_CHOOSE, (unsigned char *)countryName,
                  strlen(countryName), -1, 0);
}
```

There are a number of parameters to this function that have to do with specifics of the ASN.1 encoding. You will probably want to use the same values (`V_ASN1_APP_CHOOSE`, -1, 0) as this example.

Names In A Certificate An X.509 certificate contains the "subject" name, which identifies the current certificate, and the "issuer" name, which identifies the signer of the current certificate. The signature on an X.509 certificate should always match the public key in the certificate for the issuer name.

The subject name can be retrieved using the *X509_get_subject_name()* function and set using *X509_set_subject_name()*. The issuer name can be retrieved and set using *X509_get_issuer_name()* and *X509_set_issuer_name()*.

For example, before signing a certificate, you might set the issuer name as follows:

```
X509 *certificate = ... ;
X509 *ca_certificate = ... ;
X509_NAME *ca_name;
int success = 0;

ca_name = X509_get_subject_name(ca_certificate);
if (ca_name != NULL) {
    success = X509_set_issuer_name(certificate, ca_name);
}
```

5.3.3 Serial Numbers

Every certificate, except for self-signed certificates, must have a serial number. This number should uniquely identify the certificate for the CA that issued it. That is, the issuer and serial number should be unique world-wide. When signing a certificate, be sure and set the serial number appropriately. [1]

The serial number is stored as an ASN.1 integer type, `ASN1_INTEGER`. You can retrieve the serial number with the *X509_get_serialNumber()* function. There is no equivalent setter function needed, since the returned integer is directly mutable. For example, to set the serial number on a certificate to a particular value:

```
X509 *certificate = ... ;
ASN1_INTEGER *serial_number;
long new_serial_number = 255;
int success = 0;

serial_number = X509_get_serialNumber(certificate);
if (serial_number != NULL) {
    success = ASN1_INTEGER_set(serial_number, new_serial_number);
}
```

[1] Netscape Navigator will actually crash if a client certificate does not have a valid serial number.

Note that since the serial numbers need to be unique, it is easiest to keep a sequence number and increment it each time it is used. If you are implementing a CA, it is especially important not to re-use serial numbers, so you should store the current serial number to disk. If you are also using your CA key to sign certificates with the `openssl` command-line tool, as well as programatically, you should use the same file (which stores the next serial number to be generated), to avoid duplicating serial numbers. Although the format of the `.srl` file is generated using ASN.1, it is actually fairly simple so long as the serial numbers do not climb above 2^{31}. You can convert it to a C `long int` using the *scanf()* format `"%lx"` and generate it with *printf()* format `"%.8lX\n"`.

5.4 Issue and Expiration Times

A certificate contains two times: An issue time, when the certificate first becomes valid, and an expiration time, when the certificate ceases to be valid. These are both stored in the `X509` structure as `ASN1_TIME` elements.

OpenSSL does not provide accessor functions for these times, so you will have to access them directly from the X.509 structure:

```
X509 *certificate = ... ;
ASN1_TIME *issue_time = certificate->cert_info->validity->notBefore;
ASN1_TIME *expire_time = certificate->cert_info->validity->notAfter;
```

There are, however, functions for setting the certificate valid times, *X509_set_notBefore()* and *X509_set_notAfter()*.

OpenSSL does provide functions for generating times appropriate for X.509 certificates. For example:

```
X509 *certificate;
ASN1_TIME *time;
int success = 0;

time = X509_gmtime_adj(NULL, 3600);
if (time != NULL) {
   success = X509_set_notAfter(certificate, time);
}
```

This sets the certificate to expire one hour (3600 seconds) after the current time.

There are also functions to compare an `ASN1_TIME` with other. times. For example, the function *X509_cmp_current_time()* takes an `ASN1_TIME` pointer as an argument, and returns a value less than 0 if the passed-in time is earlier than the current time, or a value greater than 0 if it is later. This function can be used to check a certificate for validity.

5.4.1 Signing Certificates

All X.509 certificates have signatures, however, you may find it necessary to re-sign them. Particularly, if you change the contents of a certificate, the signature will become invalid. Thus you should

always sign an X.509 certificate after modifying it.

You sign a certificate using the *X509_sign()* function. For example:

```
X509 *certificate = ... ;
EVP_PKEY *private_key =  ... ;
int success;

success = X509_sign(certificate, private_key, EVP_md5());
```

Note that this function takes a digest function. This digest must be compatible with the private key algorithm (see the section on EVP signatures). For SSL, you should use RSA with MD5 digests, as that is most supported among SSL clients.

5.4.2 Verifying Certificates

You can verify if an X.509 certificate has a valid signature with the *X509_verify()* function:

```
X509 *certificate = ... ;
EVP_PKEY *public_key = ... ;
int success;

success = X509_verify(certificate, public_key);
```

This function returns 1 only if the signature on the certificate matches the given public key.

5.5 X.509 Certificate Requests

In addition to complete X.509 certificates, OpenSSL supports X.590 certificate requests. A certificate request is not a full certificate, and does not have a signature, or validity dates. OpenSSL represents X.509 certificates using the `X509_REQ` type.

You can perform some of the same operations on an X.509 request as on a certificate itself, such as viewing or modifying the public key or subject name. However, usually what one wants to do with a certificate request is to sign it. To do this in OpenSSL, convert the request into a full-fledges X.509 certificate using the *X509_REQ_to_X509()* function:

```
X509_REQ *request = ... ;
EVP_PKEY *ca_private_key = ... ;
X509 *certificate;

certificate = X509_REQ_to_X509(request, 30, ca_private_key);
```

This converts the request to a certificate, valid from the current time until 30 days in the future, and signed with `ca_private_key`. A signature key is required, since all X.509 certificates must be signed.

However, since you will almost want to modify the resulting `X509` structure to add a serial number, issuer name, and probably change the subject name (most certificate requests have an empty or useless subject), you will want to call *X509_sign()* again to re-sign the certificate after making the changes.

6 Using the SSL Library

In addition to the cryptography library (`libcrypto`), OpenSSL contains an SSL library, `libssl`, which provides an implementation of the SSL and TLS protocols for secure socket communication.

This section of the guide provides a brief overview of some of the features of the SSL library. The description is not complete, and is skewed somewhat towards running an SSL server, since that it what the CS 255 project implements, although OpenSSL supports both well. For more information on the SSL library, start at the *ssl* man page; the SSL library is much better documented than the crypto library.

6.1 SSL Contexts

An SSL context, of type `SSL_CTX` defines the framework in which an SSL client or server operates. It is generally used to set default options and properties, which are inherited by individual SSL connections.

Most functions that set options for `SSL_CTX` structures also have an equivalent function that operates on individual SSL connection structures. However, it is usually desirable to set the options once at application initialization, rather than having to set them for each connection.

6.1.1 Initializing an SSL Context

An SSL context is created by the *SSL_CTX_new()* function, which takes as an argument the type of service to provide. For example:

```
SSL_CTX *ssl_ctx;

ssl_ctx = SSL_CTX_new(SSLv23_server_method());
```

This establishes a context for a server application that will understand the SSLv2, SSLv3 and TLSv1 protocols. (For the CS 255 project, you do not need to create your own SSL context, it has been done for you.)

Once the context has been created, you should assign it a session ID. This is optional if you will not be using peer certificates. A session ID is simply a string used to identify your program in saved SSL session files. Even if you do not plan to use SSL session files, you still need to set the session ID:

```
char *session_id = "My session ID";
```

```
int success;

success  = SSL_CTX_set_session_id_context(ssl_ctx, session_id,
                                          strlen(session_id);
```

6.1.2 Local Certificates

If your SSL connections will be authenticated with certificates (almost always the case for servers, sometimes for clients), you will need to tell OpenSSL which certificate to use, and give it access to the corresponding private key.

The easiest way to do this is to tell OpenSSL which files these are stored in, and let it load them internally:

```
int success;

success =
    SSL_CTX_use_certificate_file(ssl_ctx, "cert.pem", SSL_FILETYPE_PEM)
 && SSL_CTX_use_PrivateKey_file(ssl_ctx, "key.pem", SSL_FILETYPE_PEM);
```

This will load the certificate and RSA private key from the named files (which should be in PEM format). If the key is encrypted, the pass phrase will be prompted for.

Alternately, the certificate and key can be loaded directly from `X509` and `EVP_PKEY` pointers using *SSL_CTX_use_certificate()* and *SSL_CTX_use_PrivateKey()*.

After loading the certificate, it is useful to check its validity against the private key using the function *SSL_CTX_check_private_key()*:

```
int success = SSL_CTX_check_private_key(ssl_ctx);
```

If this returns false, it means that the certificate does not match the private key.

6.1.3 Peer Certificates

SSL allows both parties in a connection to present a certificate. Normally, the server always sends a certificate to the client, but not vice versa. If the SSL context is for a server application, the *SSL_CTX_set_verify()* function can be used to have OpenSSL request that the client also send a certificate. This is needed to use client authentication. For example:

```
SSL_CTX_set_verify(ssl_ctx, SSL_VERIFY_PEER, NULL);
```

This tells OpenSSL to request a peer certificate, and use the default verification function. There are two modes, of which `SSL_VERIFY_PEER` is one. The other, `SSL_VERIFY_NONE`, is the default, and requests no peer certificate.

Note that `SSL_VERIFY_PEER` does not require the peer to send a certificate; it could still elect not to. If you want to enforce this, use `SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT` as the mode. This will cause OpenSSL to reject the connection if the peer does not send a certificate.

The third argument to *SSL_CTX_set_verify()* is a callback to a verification function. If you pass NULL, OpenSSL will use its internal verification function to try and see if the certificate is valid, by ensuring it is signed by a known CA (see below) and is not expired. If you wish to install your own verification function, see the *SSL_CTX_set_verify* man page for an example.

Client CAs If you are using an SSL server, and have set the SSL context to require peer (client) certificates, you also need to tell OpenSSL which certificates you will accept. Do this by calling *SSL_CTX_add_client_CA()* with the CA certificate:

```
X509 *ca_certificate;
int success;

success = SSL_CTX_add_client_CA(ssl_ctx, ca_certificate).
```

OpenSSL will send the list of client CAs that have been added to the client during the SSL handshake, and the client can use this list to choose the appropriate client certificate to use.

Verification Certifications If you are using the default verification callback, OpenSSL will need to know the CA certificates that the client certificates could be signed with, so it can verify them. You can install these verification certificates using *SSL_CTX_load_verify_locations()*, e.g.:

```
int success;

success = SSL_CTX_load_verify_locations(ssl_ctx, "cacert.pem", NULL);
```

Note that you are providing a filename rather than an actual certificate. The third argument (unused and NULL here) can point to a directory with multiple CA certificates, if desired.

Note that installing a CA certificate to send to the client and for verification are distinct operations, and both should be done to properly use client certificates.

6.2 SSL Connections

Once the SSL context has been set up, individual SSL connections can be created. An SSL connection is represented by the `SSL` type, and is associated with a particular network connection and peer.

6.2.1 Creating and Using Connections

SSL connections can be established using *SSL_new()*, and associated with a network socket using *SSL_accept()* or *SSL_connect()*. Data can be written or read with *SSL_write()* and *SSL_read()*, and the connection closed with *SSL_shutdown()*. The SSL connection is deallocated using *SSL_free()*.

For more information about these functions, see the appropriate man page. In the CS 255 project, you do not need to make any of these calls.

6.2.2 Getting Connection Information

Once an SSL connection has been established and the SSL handshake completed, you cannot change very many of the options. You can, however, retrieve information about the connection.

Current Cipher The *SSL_get_current_cipher()* function will return a pointer to an `SSL_CIPHER` type. This contains information about the security of the SSL connection. For example, to retrieve the number of secret bits in the currently-used encryption method:

```
SSL *ssl_connection = ... ;
SSL_CIPHER *cipher;
int bits = 0;

cipher = SSL_get_current_cipher(ssl_connection);
if (cipher != NULL) {
    bits = SSL_CIPHER_get_bits(cipher);
}
```

Peer Certificate If the peer (e.g., client) has sent a valid certificate, you can retrieve it using the *SSL_get_peer_certificate()* function:

```
SSL *ssl_connection = ... ;
X509 *certificate;

certificate = SSL_get_peer_certificate(ssl_connection);
```

This function will return NULL if the peer did not send a certificate. Since client certificates are usually not required or sent, always check for NULL unless the SSL context has been set to fail if they are not present.