OpenSSL Mini-HOWTO for NetSec

Thomas Schneider
thomaschneider@gmail.com

March 18, 2008

Contents

Chapter 1

Introduction

This `OpenSSL Mini-HOWTO for NetSec` gives a very short summary of the features of OpenSSL that are needed for the exercise course in network security [NetSec] at University of Erlangen-Nuremberg [4]. Please read the following OpenSSL literature for an introduction into OpenSSL and more detailed information.

Chapter 2

Literature Overview

The manual pages give detailed information on the parameters of the OpenSSL command line functions and libraries:

- openssl(1) [2]: OpenSSL command line tool
- ssl(3) [3]: OpenSSL SSL/TLS library
- crypto(3) [1]: OpenSSL Crypto library

The SSL Certificates HOWTO [6] contains a good introduction into certificates: contents of a certificate, certification authorities (CA), Root CAs, certificate management, etc.

The OpenSSL Command-Line HOWTO [5] gives an overview of the possibilities of the OpenSSL command-line tools.

Working examples for SSL programming including documentation can be found at [7]: `wclient` and `wserver` are basic implementations of a HTTPS client and server.

A good start into basic SSL programming is [8]: SSL Context Initialization, Certificate Verification, SSL Client/Server.

Advanced SSL programming techniques that go beyond the scope of a basic course are in [9]: Advanced SSL programming: SSL sessions, client authentication, SSL rehandshake, cipher selection, multiplexed I/O.

If you still have open questions about OpenSSL or prefer reading a book, Network Security with OpenSSL [10] is a good choice (available in library or via Amazon for approximately 20 EUR).

Chapter 3

OpenSSL command-line

3.1 Certificates

- Generate symmetrically encrypted 1024 bit RSA key pair:
  `openssl genrsa -aes128 -out CAkey.pem 1024`
- Show RSA key:
  `openssl rsa -text < CAkey.pem`
- Generate self-signed root certificate for RSA key:
  `openssl req -new -x509 -key CAkey.pem -out CAcert.pem`
- Show certificate:
  `openssl x509 -text < CAcert.pem`
- Generate certificate request (CR) for key:
  `openssl req -new -key key.pem -out req.pem`
- Show certificate request:
  `openssl req -text < req.pem`
- Issue certificate by signing CR with CA certificate:
  `openssl x509 -req -in req.pem -CA CAcert.pem -CAkey CAkey.pem -CAcreateserial -out cert.pem`

3.2 S/MIME

- Sign message:
  ```
  openssl smime -sign -in message.txt -inkey signkey.pem -signer
  signcert.pem > signed_message.txt
  ```
- Encrypt message:
  ```
  openssl smime -encrypt -aes128 -in message.txt -to hostname_of_receiver
  receivercert.pem > encrypted_message.txt
  ```
- Sign+Encrypt message (First sign, then encrypt):
  ```
  openssl smime -sign ...
  | openssl smime -encrypt ...  (no -in parameter)
  > sign_enc_message.txt
  ```

Chapter 4

OpenSSL API

4.1 Basics

The following commands must be invoked to initialize the OpenSSL API correctly.

`int SSL_library_init();`
registers the available ciphers and digests. Always returns 1.

`void SSL_load_error_strings();`
registers the human readable error strings for all **libcrypto** and **libssl** functions.

4.1.1 PEM

The PEM functions read or write structures in PEM format. In this sense PEM format is simply base64 encoded data surrounded by header lines.

`EVP_PKEY *PEM_read_PrivateKey(FILE *fp, EVP_PKEY **pem_password_cb *cb, void *u);`
reads a private key from file **fp**. The **cb** argument is the callback to use when querying for the pass phrase used for encrypted PEM private key. If the **cb** parameters is set to NULL and the **u** parameter is not NULL then the **u** parameter is interpreted as a null terminated string to use as the passphrase. If both **cb** and **u** are NULL then the default callback routine is used which will typically prompt for the passphrase on the current terminal with echoing

turned off. Returns either a pointer to the structure read or NULL if an error occurred.

`X509 *PEM_read_X509(FILE *fp, X509 **x, pem_password_cb *cb, void *u);`
reads an X509 certificate from file **fp**. The parameters **cb** and **u** are similiar to PEM_read_PrivateKey. If **x** is NULL then the parameter is ignored. If **x** is not NULL but ***x** is NULL then the structure returned will be written to ***x**. If neither **x** nor ***x** is NULL then an attempt is made to reuse the structure at ***x**. Irrespective of the value of **x** a pointer to the structure is always returned (or NULL if an error occurred).

4.2 Certificates

A certificate can be validated only against a collection of other certificate material, i.e., CA certificates (and CRLs). OpenSSL uses the object type X509_STORE to represent a collection of certificates to serve this purpose. The type X509_STORE_CTX is used to hold the data used during an actual verification. For certificate verification, first create an X509_STORE and populate it with all the available certificate (and revocation list) information. This store is used to create an X509_STORE_CTX for actual certificate verification. An X509_LOOKUP_METHOD object represents a general method of finding certificates (or CRLs), e.g. X509_LOOKUP_FILE returns a method to find certificate-related objects within a single file. X509_LOOKUP objects aggregate the collection of certificates accessible through the underlying method.

To review: an X509_STORE holds X509_LOOKUP objects built on X509_LOOKUP_METHODS. This is how the store gains access to certificate (and CRL) data. The store can then be used to create an X509_STORE_CTX to perform a verification operation.

`int X509_print_fp(FILE *fp, X509 *x);`
translates the X509 structure **x** into human-readable format and writes the result to the file pointer **fp**. It returns 1 on success or 0 on error.

4.2.1 X509_STORE

`X509_STORE *X509_STORE_new();`
creates a new X509_STORE structure and returns a pointer to it; NULL is returned on error.

`X509_LOOKUP *X509_STORE_add_lookup(X509_STORE *s, X509_LOOKUP_METHOD *m);`
adds the X509_LOOKUP_METHOD **m** to the stack `s->get_cert_methods` after creating an X509_LOOKUP that contains it as a subfield. It returns a pointer to the new X509_LOOKUP structure or NULL on error.
This can be used to load a certificate from PEM file **f** as follows:

```
X509_LOOKUP *lookup;
if(!(lookup = X509_STORE_add_lookup(s, X509_LOOKUP_file()))
|| (X509_LOOKUP_load_file(lookup, f, X509_FILETYPE_PEM)!=1))
// ERROR
```

4.2.2 Certificate verification

A X509 certificate **x509** can be verified against an X509_STORE of trusted root certificates using the following commands.

`X509_STORE_CTX *X509_STORE_CTX_new();`
create new X509_STORE_CTX.

`int X509_STORE_CTX_init(X509_STORE_CTX *ctx, X509_STORE *store, X509 *x509, STACK_OF(X509) *chain);` sets all fields of **ctx** to 0 or NULL or makes them empty, and then adds in **x509** as the certificate to be verified, **chain** as the certificate chain to be verified (this can be NULL), and **store** as the X509_STORE of trusted certificates and lookup methods for retrieving them. Returns 1 on success.

`int X509_verify_cert(X509_STORE_CTX *ctx);`
checks certificate in initialized X509_STORE_CTX **ctx**. Returns 1 on success.

4.3 Asymmetric Cryptography

4.3.1 PKCS#7 and S/MIME

PKCS#7 defines a standard format for data that has had cryptography applied to it. The Secure Multipurpose Internet Mail Extensions (S/MIME) is based on PKCS#7 and is a specification for sending secure (signed and/or encrypted) email. OpenSSL can sign/verify and encrypt/decrypt PKCS#7 messages and allows conversion between PKCS#7 and S/MIME:

`PKCS7 *SMIME_read_PKCS7(BIO *bio, BIO **bcont);`
parses a message in S/MIME format. **in** is a BIO to read the message from. If cleartext signing is used then the content is saved in a memory bio which is written to ***bcont**, otherwise ***bcont** is set to NULL. The parsed PKCS#7 structure is returned or **NULL** if an error occurred.

4.3.2 Sign/Verify

`PKCS7 *PKCS7_sign(X509 *signcert, EVP_PKEY *pkey, STACK_OF(X509) *certs, BIO *data, int flags);`
creates and returns a PKCS#7 signedData structure. **signcert** is the certificate to sign with, **pkey** is the corresponsding private key. **certs** is an optional additional set of certificates to include in the PKCS#7 structure (for example any intermediate CAs in the chain). The data to be signed is read from BIO **data**. **flags** is an optional set of flags. Returns either a valid PKCS7 structure or NULL if an error occurred. The error can be obtained from ERR_get_error(3).

`int PKCS7_verify(PKCS7 *p7, STACK_OF(X509) *certs, X509_STORE *store, BIO *indata, BIO *out, int flags);`
verifies a PKCS#7 signedData structure. **p7** is the PKCS7 structure to verify. **certs** is a set of certificates in which to search for the signer's certificate. **store** is a trusted certficate store (used for chain verification). **indata** is the signed data if the content is not present in **p7** (that is it is detached). The content is written to **out** if it is not NULL. Returns 1 for a successful verification and zero or a negative value if an error occurs.

`STACK_OF(X509) *PKCS7_get0_signers(PKCS7 *p7, STACK_OF(X509) *certs,`

`int flags);`
retrieves the signer's certificates from **p7**, it does not check their validity or whether any signatures are valid. The **certs** and **flags** parameters have the same meanings as in PKCS7_verify(). Returns all signers or NULL if an error occurred.

4.3.3 Encrypt/Decrypt

`PKCS7 *PKCS7_encrypt(STACK_OF(X509) *certs, BIO *in, const EVP_CIPHER *cipher, int flags);`
creates and returns a PKCS#7 envelopedData structure. **certs** is a list of recipient certificates. **in** is the content to be encrypted. **cipher** is the symmetric cipher to use (recommended: EVP_des_ede3_cbc() for Triple DES or EVP_aes_256_cbc() for 256 bit AES). **flags** is an optional set of flags. Returns either a PKCS7 structure or NULL if an error occurred. The error can be obtained from ERR_get_error(3).

`int PKCS7_decrypt(PKCS7 *p7, EVP_PKEY *pkey, X509 *cert, BIO *data, int flags);`
extracts and decrypts the content from a PKCS#7 envelopedData structure. **pkey** is the private key of the recipient, **cert** is the recipients certificate, **data** is a BIO to write the content to and **flags** is an optional set of flags. Returns either 1 for success or 0 for failure. The error can be obtained from ERR_get_error(3).

4.4 SSL Client/Server

The main feature of the OpenSSL library is its implementation of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols.

4.4.1 SSL_CTX

An SSL_CTX object is a factory for producing SSL connection objects. This context allows to set connection configuration parameters before the connection is made, such as protocol version, certificate information, and verification requirements.

`SSL_CTX *SSL_CTX_new(const SSL_METHOD *method);`
creates a new SSL_CTX object as framework to establish TLS/SSL enabled connections. The SSL_CTX object uses **method** as connection method. For maximum compatibility, `SSLv23_method()` should be used as **method**. Returns a pointer to the created SSL_CTX object or NULL on failure.

The SSL protocol usually requires the server to present a certificate. The certificate contains credentials that the client may look at to determine if the server is authentic and can be trusted. The SSL protocol also allows the client to optionally present certificate information so that the server may authenticate it.

`int SSL_CTX_use_certificate_chain_file(SSL_CTX *ctx, const char *file);`
loads a certificate chain from **file** into **ctx**. The certificates must be in PEM format and must be sorted starting with the subject's certificate (actual client or server certificate), followed by intermediate CA certificates if applicable, and ending at the highest level (root) CA. There is no corresponding function working on a single SSL object. Returns 1 on success.

In addition to loading the certificate chain, the SSL_CTX object must have the corresponding private key. It bears mentioning that this private key must be kept secret. Therefore, using an encrypted PEM format for on-disk storage is recommended; using triple DES in CBC mode or AES-256 is a good choice. OpenSSL collects passphrases through a callback function. The default callback prompts the user on the terminal. Otherwise, SSL_CTX_set_default_passwd_cb allows to set the callback to a user defined callback function which is invoked during the call to SSL_CTX_use_PrivateKey_file if the indicated file contains an encrypted key. Therefore, the callback should be set before making that call.

`void SSL_CTX_set_default_passwd_cb(SSL_CTX *ctx, pem_password_cb *cb);`
sets the default password callback called when loading/storing a PEM certificate with encryption.

`int pem_passwd_cb(char *buf, int size, int rwflag, void *userdata);`
hands back the password to be used during decryption. On invocation a pointer to **userdata** is provided. The pem_passwd_cb must write the password into the provided buffer buf which is of size **size**. The actual length of the password must be returned to the calling function. **rwflag** indicates whether the callback is used for reading/decryption (rwflag=0) or writing/encryption (rwflag=1).

`int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, const char *file, int type);`
adds the first private key found in **file** to **ctx**. The formatting **type** of the certificate must be specified from the known types SSL_FILETYPE_PEM, SSL_FILETYPE_ASN1. Returns 1 on success.

In order to verify the certificates, trusted CA certificates must be loaded into the SSL_CTX.

`int SSL_CTX_load_verify_locations(SSL_CTX *ctx, const char *CAfile, const char *CApath);`
specifies the locations for **ctx**, at which CA certificates for verification purposes are located. The certificates available via **CAfile** and **CApath** are trusted. Returns 1 on success.

`void SSL_CTX_free(SSL_CTX *ctx);`
frees an allocated SSL_CTX object.

Ephemeral keys

When using a cipher with RSA authentication, an ephemeral DH key exchange can take place. In this case, the session data are negotiated using the ephemeral/temporary DH key and the key supplied and certified by the certificate chain is only used for signing.

Using ephemeral DH key exchange yields forward secrecy, as the connection can only be decrypted, when the DH key is known. By generating a temporary DH key inside the server application that is lost when the application is left, it becomes impossible for an attacker to decrypt past sessions, even if he gets hold of the normal (certified) key, as this key was only used for signing.

In order to perform a DH key exchange the server must use a DH group (DH parameters) and generate a DH key. The server will always generate a new DH key during the negotiation, when the DH parameters are supplied via callback and/or when the SSL_OP_SINGLE_DH_USE option of SSL_CTX_set_options(3) is set. It will immediately create a DH key, when DH parameters are supplied via SSL_CTX_set_tmp_dh() and SSL_OP_SINGLE_DH_USE is not set. In this case, it may happen that a key is generated on initialization without later being needed, while on the other hand the computer time during the negotiation is being saved.

As generating DH parameters is extremely time consuming, an application should not generate the parameters on the fly but supply the parameters. An application may either directly specify the DH parameters or can supply the DH parameters via a callback function. The callback approach has the advantage, that the callback may supply DH parameters for different key lengths.

Please contact the manpage of SSL_CTX_set_tmp_dh_callback for more details and an example for DH setup.

4.4.2 Server

Server Socket

Accept BIOs are a wrapper round the platform's TCP/IP socket accept routines.

`BIO *BIO_new_accept(char *host_port);`
creates a new accept BIO with port **host_port**.

`int BIO_do_accept(BIO *b);`
serves two functions. When it is first called, after the accept BIO has been setup, it will attempt to create the accept socket and bind an address to it. Second and subsequent calls to BIO_do_accept() will await an incoming connection, or request a retry in non blocking mode.

If a server wishes to process multiple connections (as is normally the case) then the accept BIO must be made available for further incoming connections. This can be done by waiting for a connection and then calling:

```
connection = BIO_pop(accept);
```

After this call **connection** will contain a BIO for the recently established connection and accept will now be a single BIO again which can be used to await further incoming connections. If no further connections will be accepted the accept can be freed using BIO_free().

If only a single connection will be processed it is possible to perform I/O using the accept BIO itself. This is often undesirable however because the accept BIO will still accept additional incoming connections. This can be resolved by using BIO_pop() (see above) and freeing up the accept BIO after the initial connection.

SSL Server Socket

The SSL socket is put into the established TCP/IP socket.

```
SSL *SSL_new(SSL_CTX *ctx);
```
creates a new **SSL** structure which is needed to hold the data for a TLS/SSL connection. The new structure inherits the settings of the underlying context **ctx**: connection method (SSLv2/v3/TLSv1), options, verification settings, timeout settings.

```
void SSL_set_bio(SSL *ssl, BIO *rbio, BIO *wbio);
```
connects the BIOs **rbio** and **wbio** for the read and write operations of the TLS/SSL (encrypted) side of **ssl**. In practice, both **rbio** and **wbio** are the accepted BIO.

```
int SSL_accept(SSL *ssl);
```
waits for a TLS/SSL client to initiate the TLS/SSL handshake. The communication channel must already have been set and assigned to the ssl by setting an underlying BIO. Returns 1 on success.

After the SSL socket was sucessfully accepted, it can be used for I/O:

```
int SSL_read(SSL *ssl, void *buf, int num);
```
tries to read **num** bytes from the specified **ssl** into the buffer **buf**. Returns the number of bytes actually read from the TLS/SSL connection, 0 on shutdown or <0 on error.

```
int SSL_write(SSL *ssl, const void *buf, int num);
```
writes **num** bytes from the buffer **buf** into the specified **ssl** connection. Returns the number of bytes actually written to the TLS/SSL connection, 0 on shutdown or <0 on error.

In the end, the SSL connection is shut down and cleared for further connections or finally destroyed.

```
int SSL_shutdown(SSL *ssl);
```
shuts down an active TLS/SSL connection. Returns 1 on success, 0 on retry and -1 on error.

```
int SSL_clear(SSL *ssl);
```
reset SSL object to allow another connection. All settings (method, ciphers, BIOs) are kept. Returns 1 on success.

```
void SSL_free(SSL *ssl);
```
frees an allocated SSL structure.

4.4.3 Client

Client Socket

Connect BIOs are a wrapper round the platform's TCP/IP socket connect routines.

```
BIO *BIO_new_connect(char *name);
```
creates a new connect BIO with host **name**. The hostname can be an IP address. The hostname can also include the port in the form hostname:port . It is also acceptable to use the form "hostname/any/other/path" or "hostname:port/any/other/path".

`int BIO_do_connect(BIO *b);`
attempts to connect the supplied BIO. It returns 1 if the connection was established successfully. A zero or negative value is returned if the connection could not be established.

Client SSL Socket

The SSL socket is put into the established TCP/IP socket.

`SSL_new` and `SSL_set_bio` are used as described before.

`int SSL_connect(SSL *ssl);`
initiates the TLS/SSL handshake with a server. The communication channel must already have been set and assigned to the **ssl** by setting an underlying **BIO**. Returns 1 on success, 0 on retry and <0 on error.

After the SSL socket was successfully accepted, the server certificate must be verified.

`X509 *SSL_get_peer_certificate(const SSL *ssl);`
returns a pointer to the X509 certificate the peer presented. If the peer did not present a certificate, NULL is returned.

The subject of the certificate should be equal to the hostname:
```
X509_NAME *subj;
if((subj = X509_get_subject_name(cert))
&& X509_NAME_get_text_by_NID(subj, NID_commonName, data, 256) > 0)
{
data[255] = 0;
if(strcasecmp(data,host) != 0) // ERROR
}
```

`long SSL_get_verify_result(const SSL *ssl);`
returns the result of the verification of the X509 certificate presented by the peer, if any. Can only return one error code while the verification of a certificate can fail because of many reasons at the same time. Only the last verification error that occurred during the processing is available. Returns

X509_V_OK on success or any other value on failure. The other return values are documented in verify(1). Some of the most relevant are:
X509_V_ERR_CERT_SIGNATURE_FAILURE - the signature of the certificate is invalid,
X509_V_ERR_CERT_HAS_EXPIRED - the certificate has expired,
X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT - the passed certificate is self signed and the same certificate cannot be found in the list of trusted certificates,
X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN - the certificate chain could be built up using the untrusted certificates but the root could not be found locally,
X509_V_ERR_INVALID_CA - a CA certificate is invalid. Either it is not a CA or its extensions are not consistent with the supplied purpose, ...

After the server's certificate chain was checked successfully, the SSL socket can be used for I/O using `SSL_read` and `SSL_write` as described before.

In the end, the SSL connection is shut down (`SSL_shutdown`) and cleared for further connections (`SSL_clear`) or finally destroyed (`SSL_free`) as described before .

Bibliography

[1] OpenSSL manual pages - crypto(3).
http://www.openssl.org/docs/crypto/crypto.html.

[2] OpenSSL manual pages - openssl(1).
http://www.openssl.org/docs/apps/openssl.html.

[3] OpenSSL manual pages - ssl(3).
http://www.openssl.org/docs/ssl/ssl.html.

[4] Falko Dressler. Netzwerksicherheit [NetSec].
http://www7.informatik.uni-erlangen.de/~dressler/lectures/netzwerksicherheit/.

[5] Paul Heinlein. OpenSSL Command-Line HOWTO, Jun 2004.
http://www.madboa.com/geek/openssl/.

[6] Franck Martin. SSL Certificates HOWTO.
http://www.gtlib.cc.gatech.edu/pub/linux/docs/HOWTO/other-formats/html_single/SSL-Certificates-HOWTO.html.

[7] Eric Rescorla. OpenSSL Examples.
http://www.rtfm.com/openssl-examples/.

[8] Eric Rescorla. An Introduction to OpenSSL Programming (Part I), Oct 2001.
http://www.rtfm.com/openssl-examples/part1.pdf.

[9] Eric Rescorla. An Introduction to OpenSSL Programming (Part II), Jan 2002.
http://www.rtfm.com/openssl-examples/part2.pdf.

[10] Jon Viega, Pravir Chandra, and Matt Messier. *Network Security with Openssl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. http://www.opensslbook.com http://www.oreilly.com/catalog/openssl/.