OpenSSL programming

Piotr Pacyna

Department of Telecommunications
AGH Unviersity of Science and Technology, Kraków.

Plan

- Overview
- SSL programing - example 1
- SSL programing - step-by-step
- SSL programing - example 2

SSL online documentation

- OpenSSL:
  - SSL commands
  - SSL library (API) *)
  - Crypto library

https://www.openssl.org/docs/manmaster/
https://www.openssl.org/docs/man1.0.2/

see also:

http://fm4dd.com/openssl/
http://fm4dd.com/openssl/manual-ssl/ (2012 ?)

*) Implements SSL v2/v3 and TLS v1 protocols. Provides API. Has 214 API functions.

API functions

- Funtions dealing with ciphers
- Funtions dealing with protocol methods
- Funtions dealing with protocol contexts
- Funtions dealing with sessions
- Funtions dealing with connections

OpenSSL data structures (1/8)

SSL_CTX (SSL Context)

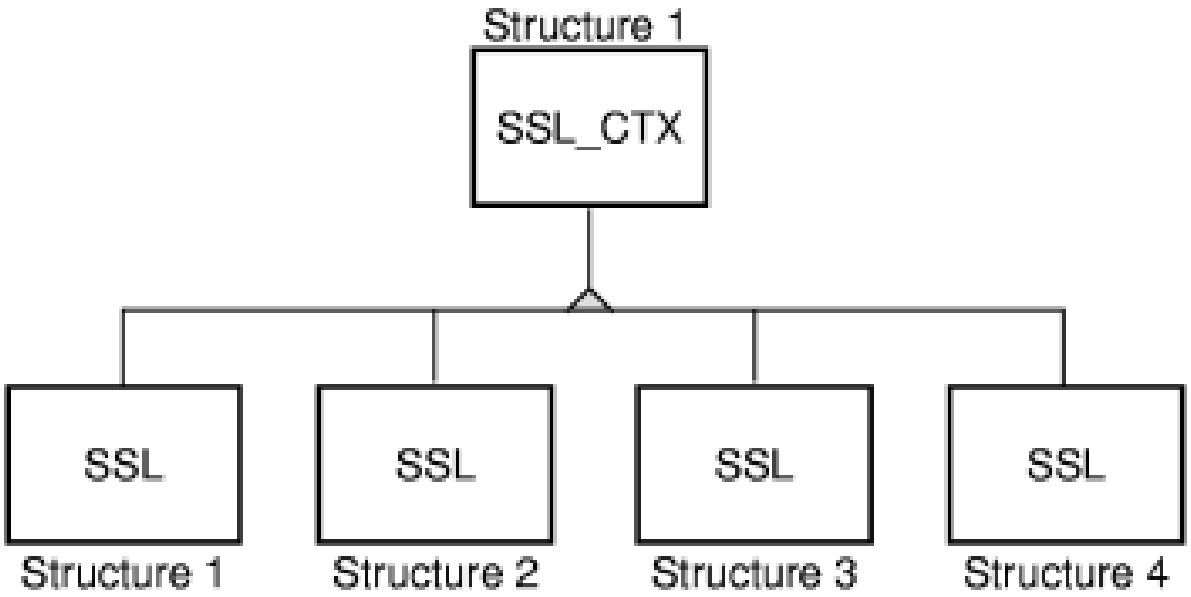
- Global context structure, created by a server or client once in program life-time.
- One SSL_CTX is sufficient per SSL application.
- Stores default values for the SSL structures, which are later created for the connections.
- Holds *some* information about connections and sessions (the numbers of new SSL connections, renegotiations, session resumptions)
- Defined in **`ssl.h`**.

OpenSSL data structures (2/8)

SSL structure

- Main SSL structure in the SSL API, required by a server or client for every SSL connection.
- Created each time a new SSL connection is created.
- Keeps pointers to other data structures that store information about SSL connections and sessions:
  - **SSL_CTX** - from which SSL struct. was created (derived)
  - **SSL_METHOD** - SSL protocol version
  - **SSL_SESSION**
  - **SSL_CIPHER**
  - **CERT** (cert. info. extracted from X.509 structure)
  - **BIO** (an SSL connection is performed via BIO)
- Created after creating and configuring SSL_CTX structure.
- Inherits default values from SSL_CTX structure.

OpenSSL data structures (3/8)



source: HP web page

OpenSSL data structures (4/8)
SSL_CTX vs. SSL

- CTX stores unchanging configuration: keys and certificates, CAs, cipherlist, etc.) and shared state for SSL connections (e.g. a cache of reusable sessions)

- Each SSL object handles and contains the state for one (active) connection: handshake temporary values, nonces, derived keys, partial record buffers, sequence counters, HMAC state, and BIO(s) for the TCP connection.

OpenSSL data structures (5/8)

SSL_SESSION (SSL Session)

Contains the current TLS/SSL session details for a connection: SSL_CIPHERs, client and server certificates, keys, etc.

OpenSSL data structures (6/8)

CERT/X509 Structure

- X.509 certificate is stored as an X509 structure.
- After loading an X509 structure into an SSL_CTX or SSL structure, the X.509 certificate information is extracted from the X509 structure and stored in a CERT structure associated with the SSL_CTX or SSL structure.
- Definitions are in: **`x509.h`** and **`ssl_locl.h`**.

OpenSSL data structures (7/8)

BIO Structure

- BIO is an I/O stream abstraction in SSL application.
- Encapsulates details for communcations.
- Handles TCP connection underlying a TLS/SSL connection.
- Communications between client and server is conducted through this structure.
- Defined in **`bio.h`**.

OpenSSL data structures (7/8)

- OpenSSL comes with a number of useful predefined BIO types.
- BIOs come in two flavors: source/sink, or filter.
- BIOs can be chained together. Each chain has exactly one source/sink, but can have any number (zero or more) of filters.
- You can use BIO functions to create the (connected) socket in a BIO, or you can create the socket yourself (with connect on the client side or accept on the server side) and then put it in a BIO.
- BIOs also support quasi-I/O, such as filtering and loopback within a process.

OpenSSL data structures (8/8)

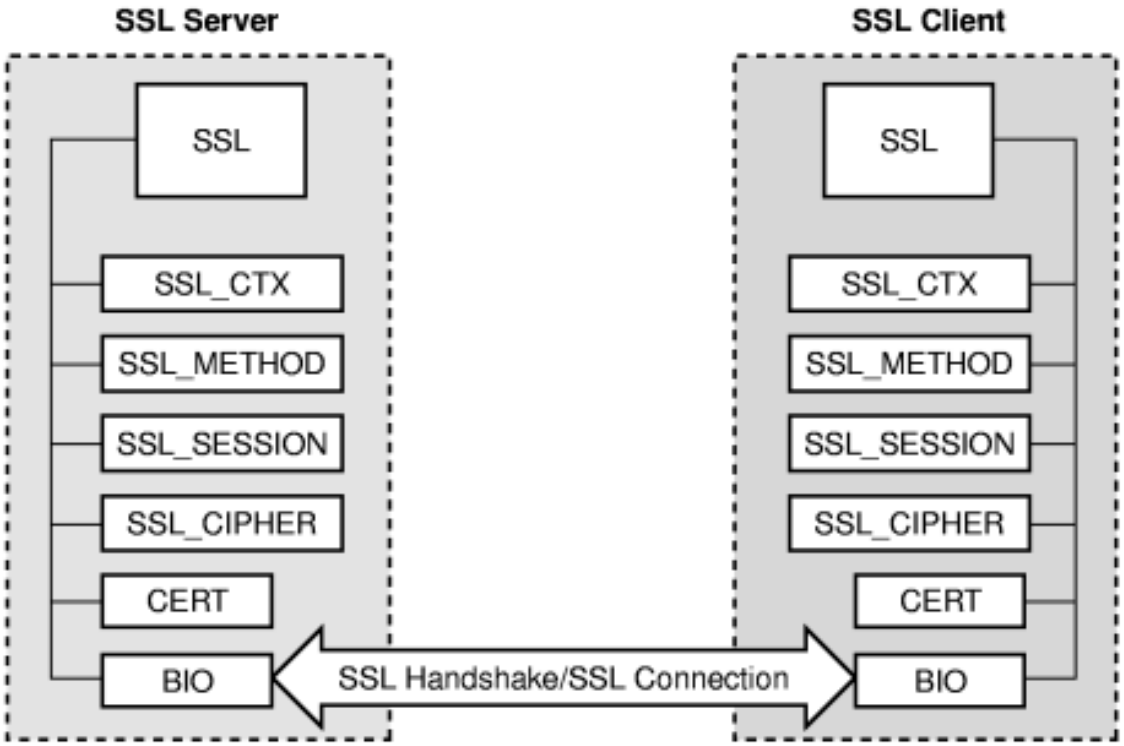
SSL_METHOD (SSL Method)

- Contains pointers to SSL library functions which implement various protocol versions (SSLv2, TLSv1).
- Needed to create an SSL_CTX.
- Defined in *ssl.h*

SSL_CIPHER (SSL Cipher)

- Holds the algorithm information for a particular cipher.
- The available ciphers are configured on a SSL_CTX basis and the actually used ones are then part of the SSL_SESSION.
- Defined in *ssl.h*

OpenSSL data structures
SSL structure on client and server



source: HP web page

OpenSSL data structures
Creation and termination

Data Structure	Creation	Termination
SSL_CTX	SSL_CTX_new()	SSL_CTX_free()
SSL	SSL_new()	SSL_free()
SSL_SESSIONS	SSL_SESSION_new()	SSL_SESSION_free()
BIO	BIO_new()	BIO_free()
X509	X509_new()	X509_free()
RSA	RSA_new()	RSA_free()
DH	DH_new()	DH_free()

OpenSSL – examples of API functions used in various applications

- add_ev_oids.c
- certfprint.c
- certserial.c
- certverify.c
- keytest.c - load and display a SSL private key
- sslconnect.c - make SSL/TLS conn., get server cert.
- certcreate.c - create a X509 digital certificate from a CSR request
- certpubkey.c - extract pub. key data from X509 cert.
- certsignature.c
- crldisplay.c
- pkcs12test.c - create a PKCS12 cert bundle
- certextensions.c - extract cert. extensions from X509 cert.
- certrenewal.c
- certstack.c
- eckeycreate.c
- set_asn1_time.c

http://fm4dd.com/openssl/

AGH

Example 1

This example is based on Secure Programming with OpenSSL from IBM
http://www.ibm.com/developerworks/library/l-openssl/

TCP handshake

- TCP offers three-way handshake
  - Client sends request (SYN)
  - Server accepts (SYN)
  - Client reciprocates (ACK)
- Client accepts and begins communications

SSL handshake

- Client sends ciphers list and random value
- Server selects the cipher
- Server sends certificate with the public key, and random value
- Client verifies the server certificate and sends private key encrypted with clients private key
- Server accepts private key and sends own private key

OpenSSL initialization

- Load algorithm tables
- Load error messages
- Select interface method
- Create new context.

```
SSL_METHOD *method;
SSL_CTX *ctx;

OpenSSL_add_all_algorithms();
SSL_load_error_strings();
method = SSLv2_server_method();
ctx = SSL_CTX_new(method);
```

Certificate management

- Load certificate files
- Load private key files
- Verify private key files

```
SSL_CTX_use_certificate_file(ctx,
CertFile, SSL_FILETYPE_PEM);

SSL_CTX_use_PrivateKey_file(ctx, KeyFile,
SSL_FILETYPE_PEM);

if ( !SSL_CTX_check_private_key(ctx) )
fprintf(stderr, "Files don't match!");
```

Attach Client to SSL

- Create SSL instance
- Attach client to instance
- Establish SSL handshake
- Commence transactions

```
SSL *ssl;
ssl= SSL_new(ctx);
SSL_set_fd(ssl, client);
SSL_accept(ssl);
SSL_read(ssl, cmd, cmdlen);
SSL_write(ssl, reply, replylen);
```

Programming steps (1/2)

1. Initialize the library - see **`SSL_library_init()`**.

2. Create SSL_CTX object. It is a framework to establish TLS/SSL enabled connections – see **`SSL_CTX_new()`**. Various options regarding certificates, algorithms etc. can be set in this object.

3. When a network connection has been created, it can be assigned to an SSL object.

Programming steps (2/2)

4. After the SSL object has been created using **`SSL_new()`**, **`SSL_set_fd()`** or **`SSL_set_bio()`** can be used to associate the network connection with the object.

5. The TLS/SSL handshake is performed using **`SSL_accept()`** or **`SSL_connect()`** respectively. **`SSL_read()`** and **`SSL_write()`** are used to read and write data on the TLS/SSL connection. **`SSL_shutdown()`** can be used to shut down the TLS/SSL connection.

Goal: *create basic unsecure and secure connections*

- source files:
  - **`nossl.c`** - demo on OpenSSL for basic communication without using SSL
  - **`withssl.c`** - demo on OpenSSL for an SSL connection
  - **`TrustStore.pem`** - Certificate file needed by **`withssl.c`**

noSSL – basic setup – openSSL 0.9.8

```
#include "openssl/ssl.h"
#include "openssl/bio.h"
#include "openssl/err.h"
#include "stdio.h"
#include "string.h"

int main()
{
    BIO * bio;
    int p;

    char * request = "GET / HTTP/1.1\x0D\x0AHost:
   www.verisign.com\x0D\x0A\x43onnection:
   Close\x0D\x0A\x0D\x0A";
    char r[1024];

   /* Set up the library */

   ERR_load_BIO_strings();
    SSL_load_error_strings();
```

noSSL – create connection, send request

```
/* Create and setup the connection */

  bio = BIO_new_connect("www.verisign.com:80");
  if(bio == NULL) { printf("BIO is null\n"); return; }

  if(BIO_do_connect(bio) <= 0)
  {         ERR_print_errors_fp(stderr);
      BIO_free_all(bio);
      return;
  }

/* Send the request */

BIO_write(bio, request, strlen(request));
```

noSSL – get response, close connection

```
  /* Read in the response */

  for(;;)
    {
        p = BIO_read(bio, r, 1023);
        if(p <= 0) break;
        r[p] = 0;
        printf("%s", r);
    }

  /* Close the connection and free the context */

  BIO_free_all(bio);    return 0;
}
```

withSSL – basic setup

```
#include "openssl/ssl.h"
#include "openssl/bio.h"
#include "openssl/err.h"

#include "stdio.h"
#include "string.h"

int main()
{
    BIO * bio;
    SSL * ssl;
    SSL_CTX * ctx;

    int p;

    char * request = "GET / HTTP/1.1\x0D\x0AHost:
  www.verisign.com\x0D\x0A\x43onnection:
  Close\x0D\x0A\x0D\x0A";
    char r[1024];
```

withSSL – set up: library, context, conn.

```
/* Set up the library */

ERR_load_BIO_strings();
  SSL_load_error_strings();
  OpenSSL_add_all_algorithms();

/* Set up the SSL context */
ctx = SSL_CTX_new(SSLv23_client_method());    /* Load the
 trust store */
 if(! SSL_CTX_load_verify_locations(ctx, "TrustStore.pem",
 NULL))    {
      fprintf(stderr, "Error loading trust store\n");
     ERR_print_errors_fp(stderr);
     SSL_CTX_free(ctx);
       return 0;    }
```

withSSL – set up: library, context, conn.

```
/* Setup the connection */

bio = BIO_new_ssl_connect(ctx);

/* Set the SSL_MODE_AUTO_RETRY flag */
BIO_get_ssl(bio, & ssl);
  SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);
```

withSSL – set up conn., check cert.

```
/* Create and setup the connection */

BIO_set_conn_hostname(bio, "www.verisign.com:https");
  if(BIO_do_connect(bio) <= 0)
  {         fprintf(stderr, "Error attempting to connect\n");
      ERR_print_errors_fp(stderr);
      BIO_free_all(bio);
      SSL_CTX_free(ctx);
      return 0;
  }

/* Check the certificate */

if(SSL_get_verify_result(ssl) != X509_V_OK)
  {         fprintf(stderr, "Certificate verification error:
 %i\n", SSL_get_verify_result(ssl));
      BIO_free_all(bio);
      SSL_CTX_free(ctx);
return 0;     }
```

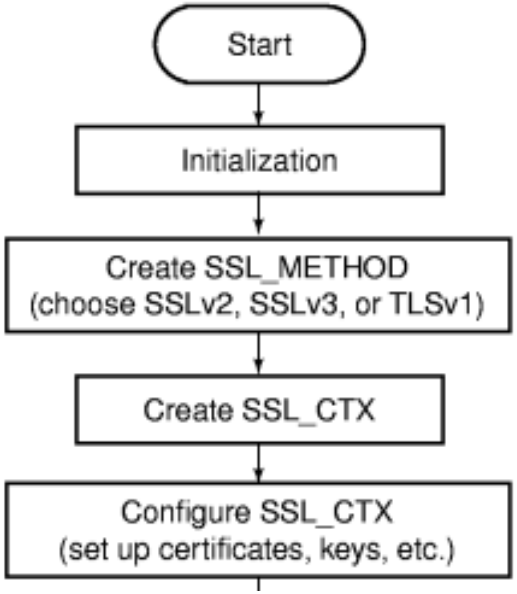
withSSL – send request, get response, close connection

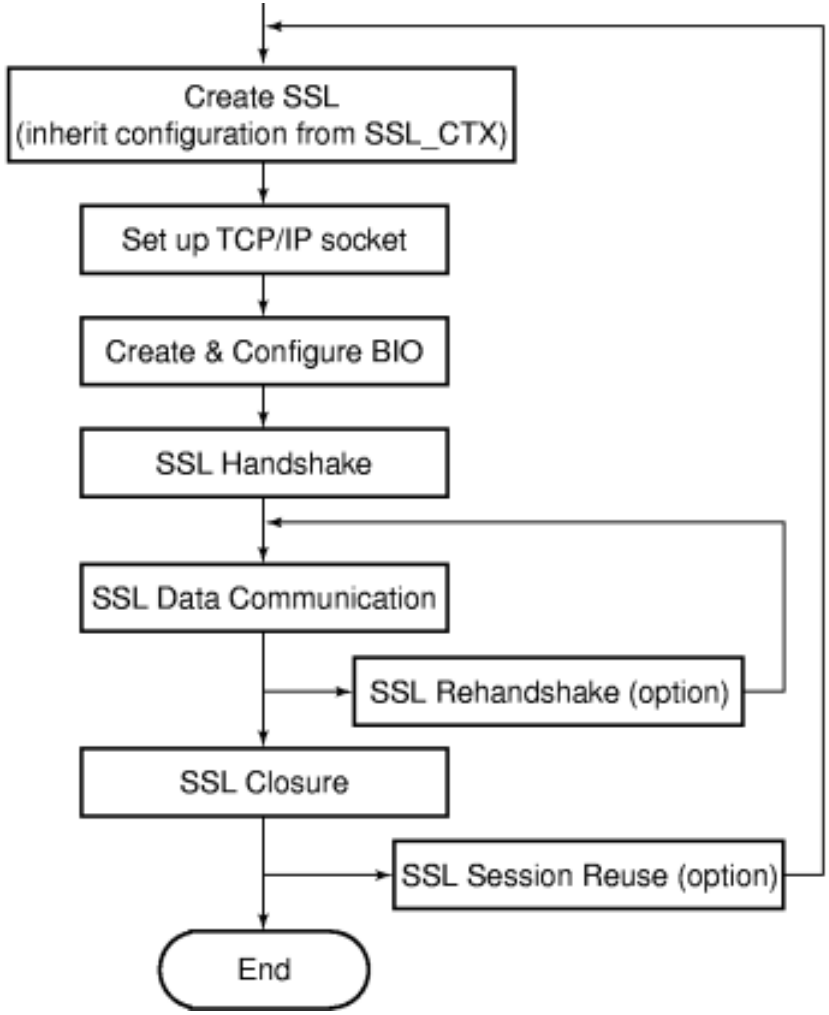
```
/* Send the request */

  BIO_write(bio, request, strlen(request));

/* Read in the response */

for(;;)
  {
      p = BIO_read(bio, r, 1023);
      if(p <= 0) break;
      r[p] = 0;
      printf("%s", r);
  }

/* Close the connection and free the context */

BIO_free_all(bio);
  SSL_CTX_free(ctx);
  return 0;
}
```

SSL programing step-by-step

SSL programing is based on SSL Programming Tutorial from HP
http://h71000.www7.hp.com/doc/83final/ba554_90007/ch04s03.html

OpenSSL application workflow





source: HP web page

Initializing the SSL library

- Register all ciphers and hash algorithms in SSL.

```
SSL_library_init();
/* load encryption & hash algorithms */
```

- Include error strings for SSL APIs and Crypto APIs.

```
SSL_load_error_strings();
/* load error strings for error reporting */
```

Creating and setting up SSL Method and Context structures

- Choose an SSL/TLS protocol version and create SSL_METHOD structure.

Proto.	client & server	server	client
SSLv2	`SSLv2_method()`	`SSLv2_server_ method()`	`SSLv2_client_ method`
SSLv3	`SSLv3_method()`	`SSLv3_server_ method()`	`SSLv3_client_ method`
TLSv1	`TLSv1_method()`	`TLSv1_server_ method()`	`TLSv1_client_ method`
SSLv23	`SSLv23_method()`	`SSLv23_server_ method()`	`SSLv23_client_ metho`

- Create SSL_CTX structure with the **`SSL_CTX_new()`**.

```
meth = SSLv3_method();
ctx = SSL_CTX_new(meth);
```

Example functions to deal with SSL_CTX structure

```
int     SSL_CTX_add_client_CA(SSL_CTX *ctx, X509 *x);
long    SSL_CTX_add_extra_chain_cert(SSL_CTX *ctx, X509 *x509);
int     SSL_CTX_add_session(SSL_CTX *ctx, SSL_SESSION *c);
int     SSL_CTX_check_private_key(const SSL_CTX *ctx);
long    SSL_CTX_ctrl(SSL_CTX *ctx, int cmd, long larg, char *parg);
void    SSL_CTX_flush_sessions(SSL_CTX *s, long t);
void    SSL_CTX_free(SSL_CTX *a);
char    *SSL_CTX_get_app_data(SSL_CTX *ctx);
X509_STORE *SSL_CTX_get_cert_store(SSL_CTX *ctx);
STACK *SSL_CTX_get_client_CA_list(const SSL_CTX *ctx);
int (*SSL_CTX_get_client_cert_cb(SSL_CTX *ctx))(SSL *ssl, X509 **x509,
   EVP_PKEY **pkey);
void    SSL_CTX_get_default_read_ahead(SSL_CTX *ctx);
char    *SSL_CTX_get_ex_data(const SSL_CTX *s, int idx);
int     SSL_CTX_get_ex_new_index(long argl, char *argp, int
   (*new_func);(void), int (*dup_func)(void), void (*free_func)(void))
```

see https://www.openssl.org/docs/manmaster/ssl/ssl.html

Example 2: Setting up certificates for the SSL server

- In order to conduct client authentication by the server, the server must load CA certificate so that it can verify the client certificate.

```
/* Load server certificate into the SSL context */
          if (SSL_CTX_use_certificate_file(ctx, SERVER_CERT,
        SSL_FILETYPE_PEM) <= 0) {
                ERR_print_errors(bio_err);  /* ==
                ERR_print_errors_fp(stderr); */
                exit(1);
          }
/* Load the server private-key into the SSL context */
           if (SSL_CTX_use_PrivateKey_file(ctx, SERVER_KEY,
         SSL_FILETYPE_PEM) <= 0) {
                ERR_print_errors(bio_err);  /* ==
                ERR_print_errors_fp(stderr); */
                   exit(1);
          }
```

Example 2: Setting up certificates for the SSL server (cont.)

```
/* Load trusted CA. */
          if (!SSL_CTX_load_verify_locations(ctx,CA_CERT,NULL)) {
                ERR_print_errors(bio_err);  /* ==
              ERR_print_errors_fp(stderr); */
                 exit(1);
          }

/* Set to require peer (client) certificate verification */
        SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);

/* Set the verification depth to 1 */
             SSL_CTX_set_verify_depth(ctx,1);
```

Example 2: Setting up certificates for SSL Client

- TLS client verifies the server certificate during SSL handshake.
- Verification requires SSL client to set up its trusting CA certificate.
- The server certificate must be signed with the CA certificate loaded in the SSL client in order for the server certificate verification to succeed.

The following example shows how to set up certificates for the SSL client:

Example 2: Setting up certificates for SSL Client (cont.)

```
/*-- Load a client certificate into the SSL_CTX structure -*/
      if(SSL_CTX_use_certificate_file(ctx,CLIENT_CERT,
      SSL_FILETYPE_PEM) <= 0){
                  ERR_print_errors_fp(stderr);
              exit(1);
           }
 /*-- Load a private-key into the SSL_CTX structure -----*/
       if(SSL_CTX_use_PrivateKey_file(ctx,CLIENT_KEY,
       SSL_FILETYPE_PEM) <= 0){
                  ERR_print_errors_fp(stderr);
              exit(1);
           }
 /* -- Load trusted CA. -- */
           if (!SSL_CTX_load_verify_locations(ctx,CA_CERT,NULL)) {
                   ERR_print_errors_fp(stderr);
                  exit(1);
           }
```

Creating and setting up SSL structure

- SSL structure stores information for an SSL connection:

```
ssl = SSL_new(ctx);
```

- A newly created SSL structure inherits information from the SSL_CTX structure, including types of connection methods, options, verification settings, and timeout settings. No additional settings are required for the SSL structure if the appropriate initialization and configuration have been done for the SSL_CTX structure.

Creating and setting up SSL structure

- *Default* values in the SSL structure can be modified by setting attributes of the SSL_CTX structure.
  - use SSL_CTX_use_certificate() to load a certificate into an SSL_CTX structure,
  - use SSL_use_certificate() to load a certificate into an SSL structure.

Setting up TCP connection
Creating and setting up listening socket on the SSL server

- Configuration is like in many other TCP/IP client/server applications (not specific to SSL)
- TCP/IP connection is set up with ordinary socket.
- The SSL server needs two sockets —one for the SSL connection, the other for detecting an incoming connection request from the SSL client.
  - **`socket()`** creates a listening socket.
  - **`bind()`** assigns address / port to the listening socket,
  - **`listen()`** allows the listening socket to handle incoming TCP/IP connection request from the client.

Setting up TCP connection
Creating and setting up listening socket on the SSL server

```
listen_sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
CHK_ERR(listen_sock, "socket");

memset(&sa_serv, 0, sizeof(sa_serv));
sa_serv.sin_family      = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port        = htons(s_port);/* Srvr port */

err = bind(listen_sock, (struct
  sockaddr*)&sa_serv,sizeof(sa_serv));
CHK_ERR(err, "bind");

/* Receive a TCP connection. */
err = listen(listen_sock, 5);
CHK_ERR(err, "listen");
```

Setting up TCP connection
Creating and setting up socket on the SSL client

- Create a TCP socket
- Connect to the server with the socket using **`connect()`**.
- If **`connect()`** succeeds, the socket passed to the function as the first argument can be used for data communication over the connection.

Setting up TCP connection
Creating and setting up socket on the SSL client (cont.)

```
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
CHK_ERR(sock, "socket");

memset (&server_addr, '\0', sizeof(server_addr));
server_addr.sin_family      = AF_INET;
server_addr.sin_port        = htons(s_port);       /*
 Server Port number */
server_addr.sin_addr.s_addr = inet_addr(s_ipaddr); /*
 Server IP */

err = connect(sock, (struct sockaddr*) &server_addr,
 sizeof(server_addr));
CHK_ERR(err, "connect");
```

Setting up TCP connection
Establishing TCP connection on the SSL server

- To accept an incoming connection request and to establish a TCP/IP connection, SSL server needs to call **`accept()`**.
- Socket created with this function is used for the data communication between the SSL client and server.

```
sock = accept(listen_sock, (struct sockaddr*)&sa_cli,
  &client_len);
BIO_printf(bio_c_out, "Connection from %lx, port %x\n",
sa_cli.sin_addr.s_addr, sa_cli.sin_port);
```

Setting up TCP connection
Setting up the socket/socket BIO in the SSL structure

- After you create the SSL structure and the TCP socket, you must configure them so that SSL data communication with the SSL structure can be performed automatically through that socket.
- Three ways are possible to assign **`sock`** to SSL :
  - set the socket directly into the SSL structure:

```
SSL_set_fd(ssl, sock);
```

Setting up TCP connection
Setting up the socket/socket BIO in the SSL structure (cont.)

- **You can also use a BIO structure.** It is the I/O abstraction provided by OpenSSL. BIO hides details of an underlying I/O. Create a socket BIO and set it into the SSL structure.

```
sbio=BIO_new(BIO_s_socket());
BIO_set_fd(sbio, sock, BIO_NOCLOSE);
SSL_set_bio(ssl, sbio, sbio);
```

- Create BIO socket with `BIO_new_socket()` in which the TCP socket is assigned:

```
sbio = BIO_new_socket(socket, BIO_NOCLOSE);
SSL_set_bio(ssl, sbio, sbio);
```

`SSL_set_bio()` assigns the socket BIO into the SSL structure.

SSL handshake on the SSL server

- SSL handshake is performed by calling **`SSL_accept()`** on the SSL server. The **`SSL_accept()`** waits for SSL handshake initiation from SSL client.

```
err = SSL_accept(ssl);
```

- Successful completion means that the SSL handshake has been completed.

SSL handshake on the SSL client

- SSL handshake is performed by calling **`SSL_connect()`** on the SSL client. It initiates the handshake.

```
err = SSL_connect(ssl);
```

☞ Return value of 1 indicates successful handshake.
Data can now be transmitted securely over this connection.

SSL Handshake with SSL_read and SSL_write

- Optionally, you can call **`SSL_write()`** and **`SSL_read()`** to complete the SSL handshake as well as to perform SSL data exchange.
  - When **`SSL_accept()`** is not called, **`SSL_set_accept_state()`** must be called prior to **`SSL_read()`** on the SSL server.
  - When **`SSL_connect()`** is not called,**`SSL_set_connect_state()`** must be called prior to **`SSL_write`**() on the client.

```
SSL_set_accept_state(ssl);
SSL_read()

SSL_set_connect_state(ssl);
SSL_write()
```

Obtaining peer certificate

- After the SSL handshake, you can obtain a peer certificate, if needed:

```
peer_cert = SSL_get_peer_certificate(ssl);
```

☞ This function can be used for straight certificate verification, such as checking certificate information (for example, the common name and expiration date).

Transmitting data over SSL

- After SSL handshake is completed, data can be transmitted securely over the established SSL connection. **`SSL_write()`** and **`SSL_read()`** are used for secure data transmission, just as **`write()`** and **`read()`** or **`send()`** and **`recv()`** are used for an ordinary TCP/IP connection.

```
err = SSL_write(ssl, wbuf, strlen(wbuf));
err = SSL_read(ssl, rbuf, sizeof(rbuf)-1);
```

Using BIOs for SSL data transmission

- Instead of using **`SSL_write()`** and **`SSL_read()`**, you can transmit data by calling **`BIO_puts()`** and **`BIO_gets()`**, and **`BIO_write()`** and **`BIO_read()`**, provided that a BIO buffer is created and set up:

Using BIOs for SSL data transmission

```
BIO         *buf_io, *ssl_bio;
char      rbuf[READBUF_SIZE];
char     wbuf[WRITEBUF_SIZE]

buf_io = BIO_new(BIO_f_buffer());  /* create a buffer BIO */
ssl_bio = BIO_new(BIO_f_ssl());    /* create an ssl BIO */
BIO_set_ssl(ssl_bio, ssl, BIO_CLOSE);/* assign ssl BIO to SSL*/
BIO_push(buf_io, ssl_bio);          /* add ssl_bio to buf_io */

ret = BIO_puts(buf_io, wbuf);
      /* Write contents of wbuf[] into buf_io */
ret = BIO_write(buf_io, wbuf, wlen);
      /* Write wlen-byte contents of wbuf[] into buf_io */
 ret = BIO_gets(buf_io, rbuf, READBUF_SIZE);
      /* Read data from buf_io and store in rbuf[] */
ret = BIO_read(buf_io, rbuf, rlen);
      /* Read rlen-byte data from buf_io and store rbuf[] */
```

Closing SSL connection with SSL shutdown

- Each party is required to send a **`close_notify`** alert before closing the write side of the connection.
- Either party can initiate close by sending a **`close_notify`** alert to notify of the SSL closure.
- Any data received after sending a closure alert is ignored.
- The other party is required both to respond with a **`close_notify`** alert of its own and to close down the connection immediately, discarding any pending writes.
- The initiator of the close is not required to wait for the responding **`close_notify`** alert before closing the read side of the connection.
- The SSL client or server that initiates the SSL closure with **`SSL_shutdown()`** sent once or twice. If it calls the function twice, one call sends the **`close_notify`** alert and one call receives the response from the peer. If the initator makes the call only once, the initiator does not receive the **`close_notify`** alert from the peer. (The initiator is not required to wait for the responding alert.)
- The peer that receives the alert calls **`SSL_shutdown()`** once to send the alert to the initiating party.

Resuming SSL connection

- You can reuse the information from an already established SSL session to create a new SSL connection.
- SSL handshake can be performed faster because the new SSL connection is reusing the same master secret.
- SSL session resumption reduces the load of a server that is accepting many SSL connections.

Resuming SSL connection
SSL session resumption on the SSL client

- Start the first SSL connection (and create SSL sess.)

```
ret = SSL_connect(ssl)
```

- Use **`SSL_read()`** and **`SSL_write()`** for data communication over SSL connection.

- Save the SSL session information.

```
sess = SSL_get1_session(ssl);
/* sess is an SSL_SESSION, and ssl is an SSL */
```

- Shut down the first SSL connection.

```
SSL_shutdown(ssl);
```

Resuming SSL connection
SSL session resumption on the SSL client

- Create a new SSL structure.

```
ssl = SSL_new(ctx);
```

- Set the SSL session to a new SSL session before calling **`SSL_connect()`**.

```
SSL_set_session(ssl, sess);
err = SSL_connect(ssl);
```

- Start the second SSL conn. with sess. resumption.

```
ret = SSL_connect(ssl)
```

- Use **`SSL_read()`** / **`SSL_write()`** for data exchange

Finishing the SSL application

- When closing SSL application, deallocate data structures. Deallocation functions usually contain the _free suffix.
- You must free up data structures that you explicitly created in the SSL application.
- Data structures that were created inside another structure with an xxx_new() API are automatically deallocated when the structure is deallocated with the corresponding xxx_free().
  - a BIO structure created with SSL_new() is freed when you call SSL_free(); you do not need to call BIO_free() to free the BIO inside the SSL structure.
  - However, if you called BIO_new() to allocate a BIO structure, you must free that structure with BIO_free().

AGH

Example 3
SSL client and server using OpenSSL APIs

Example 3: Setting up unsecured connection

```
BIO * bio;
int x;

if ((bio = BIO_new_connect("hostname:port")) == NULL ||
BIO_do_connect(bio) <= 0) {
/* Handle failed connection */
}

if ((x = BIO_read(bio, buf, len)) <= 0) {
/* Handle error/closed connection */
}

BIO_reset(bio); /* reuse the connection */
BIO_free_all(bio); /* cleanup */
```

Example 3: Setting up a secured connection

```
SSL_CTX * ctx;
SSL * ssl;
if ((ssl = SSL_CTX_new(SSLv23_client_method())) == NULL)
err(1, "SSL_CTX_new());
if
   (SSL_CTX_load_verify_locations(ctx,”/path/to/TrustStore.pem",
   NULL) != 0) {
/* Handle failed load here */
SSL_CTX_free(ctx);
}
if ((bio = BIO_new_ssl_connect(ctx)) == NULL) {
SSL_CTX_free(ctx);
err(1, "BIO_new_ssl_connect());
}
BIO_get_ssl(bio, & ssl);
SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);
```

Example 3: Setting up a secured connection

```
/* Attempt to connect */
BIO_set_conn_hostname(bio, "hostname:port");

/* Verify the connection opened and perform the handshake */
if (BIO_do_connect(bio) <= 0 || SSL_get_verify_result(ssl) !=
   X509_V_OK) {
BIO_free_all(bio);
SSL_CTX_free(ctx);
err(1, "BIO_do_connect()/SSL_get_verify_result()");
}

BIO_free_all(bio);
SSL_CTX_free(ctx);
```

Example 3: Error detection and reporting

```
printf("Error: %s\n",
   ERR_reason_error_string(ERR_get_error()));
ERR_print_errors_fp(FILE *);
ERR_print_errors(BIO *);
CRYPTO_mem_ctrl(CRYPTO_MEM_CHECK_ON); /* really needed? */
(void)SSL_library_init();
SSL_load_error_strings();
printf("Error: %s\n",
   ERR_error_string(SSL_get_error((ssl),(err)), NULL);
```

Example 3: Server example (1/2)

```
SSL_load_error_strings();
OpenSSL_add_ssl_algorithms();
if ((ctx = SSL_CTX_new(SSLv23_server_method())) == NULL)
fatalx("ctx");
if (!SSL_CTX_load_verify_locations(ctx, SSL_CA_CRT, NULL))
fatalx("verify");
SSL_CTX_set_client_CA_list(ctx,
   SSL_load_client_CA_file(SSL_CA_CRT));
if (!SSL_CTX_use_certificate_file(ctx, SSL_SERVER_CRT,
   SSL_FILETYPE_PEM))
fatalx("cert");
if (!SSL_CTX_use_PrivateKey_file(ctx, SSL_SERVER_KEY,
   SSL_FILETYPE_PEM))
fatalx("key");
if (!SSL_CTX_check_private_key(ctx))
fatalx("cert/key");
```

Example 3: Server example (2/2)

```
SSL_CTX_set_mode(ctx, SSL_MODE_AUTO_RETRY);
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER |
   SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);
SSL_CTX_set_verify_depth(ctx, 1);

/* setup socket - socket()/bind()/listen() */
for (; work != 0;) {
if ((s = accept(sock, 0, 0)) == -1)
err(EX_OSERR, "accept");
sbio = BIO_new_socket(s, BIO_NOCLOSE);
ssl = SSL_new(ctx);
SSL_set_bio(ssl, sbio, sbio);
if ((r = SSL_accept(ssl)) == -1)
warn("SSL_accept");
}
```

Example 3: Client example (1/2)

```
SSL_load_error_strings();
OpenSSL_add_ssl_algorithms();
if ((ctx = SSL_CTX_new(SSLv23_client_method())) == NULL)
fatalx("ctx");
if (!SSL_CTX_load_verify_locations(ctx, SSL_CA_CRT, NULL))
fatalx("verify");
if (!SSL_CTX_use_certificate_file(ctx, SSL_CLIENT_CRT,
   SSL_FILETYPE_PEM))
fatalx("cert");
if (!SSL_CTX_use_PrivateKey_file(ctx, SSL_CLIENT_KEY,
   SSL_FILETYPE_PEM))
fatalx("key");
if (!SSL_CTX_check_private_key(ctx))
fatalx("cert/key");
SSL_CTX_set_mode(ctx, SSL_MODE_AUTO_RETRY);
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
SSL_CTX_set_verify_depth(ctx, 1);
```

Example 3: Client example (2/2)

```
/* setup connection */
if ((hp = gethostbyname("localhost")) == NULL)
err(EX_OSERR, "gethostbyname");
/* init socket - socket()/connect() */
/* go do ssl magic */
ssl = SSL_new(ctx);
sbio = BIO_new_socket(sock, BIO_NOCLOSE);
SSL_set_bio(ssl, sbio, sbio);
if (SSL_connect(ssl) <= 0)
fatalx("SSL_connect");
if (SSL_get_verify_result(ssl) != X509_V_OK)
fatalx("cert");
printf("connected to server!\n");
SSL_free(ssl);
BIO_free_all(sbio);
SSL_CTX_free(ctx);
```

Sources

https://www.openssl.org/docs/manmaster/

https://www.openssl.org/docs/man1.0.2/

http://fm4dd.com/openssl/

http://fm4dd.com/openssl/manual-ssl/

http://www.cs.utah.edu/~swalton/

http://h71000.www7.hp.com/doc/83final/ba554_90007/ch04s03.html (SSL programming step-by-step)

https://www.ibm.com/developerworks/linux/library/l-openssl/index.html#download (Example 1)

References

https://www.openssl.org/
http://www.libressl.org/ (OpenSSL fork)
http://www.ietf.org/rfc/rfc2246.txt
http://www.ietf.org/rfc/rfc3546.txt
http://tools.ietf.org/html/rfc6347.txt
http://tools.ietf.org/html/rfc6083.txt
https://tools.ietf.org/html/rfc6520.txt

End of OpenSSL programming

Piotr Pacyna

Department of Telecommunications
AGH Unviersity of Science and Technology, Kraków.

AGH

thank you