

Securing Sockets with OpenSSL (part 2)

Securing Sockets with OpenSSL (part 2)

As you write programs that interface with the network, you quickly find that nothing is really private or confidential. Effortlessly, you can write network snoopers that grab every message that passes along your network segment. This also means that others, who may be not as trustworthy, can do the same to your messages. The Internet is moving more towards privacy. Of course, because of the public nature of the Internet, this is impossible. But, how close to data security and privacy can the industry get?

While privacy is unreachable in a public setting like the Internet, you can get pretty close with encryption. The Secure Socket Layer (SSL) provides a standard and reliable mechanism to interface two networked computers. This is the second part of an article on OpenSSL (from www.openssl.org), a production-ready SSL API. The first part discusses the terminology and issues of securing the channel over TCP/IP. This part completes the discussion with program examples for a secure client and server.

Writing an HTTPS Server

Writing a program to serve up secure messages requires only a few changes to the TCP server. Once you have completed those steps you are free and clear to send and receive private information. (You can get the complete code listings to all the code fragments in this article from <http://www.cs.utah.edu/~swalton>.)

Before creating a socket, you must initialize the OpenSSL library. Initialization includes loading the ciphers, error messages, creating server instances, and creating an SSL context. The only time that you change the following code is when and if you need to set up separate contexts (like supporting TLS and SSLv3 in the same program).

```
SSL_METHOD *method;
SSL_CTX *ctx;
OpenSSL_add_all_algorithms();      /* load & register cryptos */
SSL_load_error_strings();         /* load all error messages */
method = SSLv2_server_method();   /* create server instance */
ctx = SSL_CTX_new(method);        /* create context */
```

The next step is to load the server's certificates. The certificates are stored in a file along with your private key. The certificates must be ready before the client connects; this is why you have to take these steps prior to setting up a socket. The code fragment, below, lists the code to load the certificate and private key from CertFile and KeyFile (remember: you can store both in the same file). This source listing leaves out the error checking for presentation clarity. Be sure you include checks for any errors in production code.

```
/* set the local certificate from CertFile */
SSL_CTX_use_certificate_file(ctx, CertFile, SSL_FILETYPE_PEM);
/* set the private key from KeyFile */
SSL_CTX_use_PrivateKey_file(ctx, KeyFile, SSL_FILETYPE_PEM);
/* verify private key */
if ( !SSL_CTX_check_private_key(ctx) )
    abort();
```

Securing Sockets with OpenSSL (part 2)

The first two calls, `SSL_CTX_use_certificate_file()` and `SSL_CTX_use_PrivateKey_file()`, get the certificate and the private key, respectively. The last call, `SSL_CTX_check_private_key()`, verifies the private key against the known certificate. This makes sure that nothing got corrupted (or even cracked).

The next step is to create the server socket. This is identical to a typical TCP listening socket: you create the socket, bind it to a particular port and convert it to a listening socket.

```
/*--- Standard TCP server setup and connection ---*/
int sd, client;
struct sockaddr_in addr;
sd = socket(PF_INET, SOCK_STREAM, 0); /* create stream socket */
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(port); /* select 'port' */
addr.sin_addr.s_addr = INADDR_ANY; /* any available addr */
bind(sd, (struct sockaddr*)&addr, sizeof(addr)); /* bind it */
listen(sd, 10); /* make into listening socket */
client = accept(sd, 0, 0); /* await and accept connections */
```

OpenSSL sits on top of the TCP stack. So, all you have to do is hand off the `client` socket descriptor. When the library gets the newly connected client, it begins the SSL handshake. And, when it is finally done, your program has a fully qualified and secure connection.

```
int client, bytes;
SSL *ssl = SSL_new(ctx); /* get new SSL state with context */
SSL_set_fd(ssl, client); /* set connection to SSL state */
SSL_accept(ssl); /* start the handshaking */
/* now you can read/write */
bytes = SSL_read(ssl, buf, sizeof(buf)); /* get HTTP request */
/*...process request */
SSL_write(ssl, reply, strlen(reply)); /* send reply */
/*...*/
/* close connection & clean up */
client = SSL_get_fd(ssl); /* get the raw connection */
SSL_free(ssl); /* release SSL state */
close(sd); /* close connection */
```

Each connection gets its own SSL connection state with the `SSL_new()` library call. The program sets the raw client connection to this connection state. From that point on, the OpenSSL library uses the connection state for all I/O and control. The library replaces `recv()` and `send()` with `SSL_read()` and `SSL_write()`. The last step that calls `SSL_accept()` completes the SSL handshaking.

Writing a Secure Client

Writing an SSL client that connects to a secure server is very similar, changing only a few lines of the client code. Some parts appear to be identical to the server's equivalent, but note the single bolded difference.

```
SSL_METHOD *method;
SSL_CTX *ctx;
OpenSSL_add_all_algorithms(); /* load & register cryptos */
SSL_load_error_strings(); /* load all error messages */
method = SSLv2_client_method(); /* create client instance */
```

Securing Sockets with OpenSSL (part 2)

```
ctx = SSL_CTX_new(method); /* create context */
```

The only difference is the call to make a client instance using `SSLv2_client_method()`. After setting up the SSL library, you need to create the socket. Once again the client socket code is essentially a standard TCP socket which finds and connects to a server.

```
/*---Standard TCP Client---*/
int sd;
struct hostent *host;
struct sockaddr_in addr;

host = gethostbyname(hostname); /* convert hostname IP addr */
sd = socket(PF_INET, SOCK_STREAM, 0); /* create TCP socket */
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(port); /* set the desired port */
addr.sin_addr.s_addr = *(long*)(host->h_addr); /* and address */
connect(sd, (struct sockaddr*)&addr, sizeof(addr)); /* connect */
```

The client has, at this point, succeeded in connecting to the TCP-side of the server. Like the server, it must perform the SSL handshaking to complete the transition to a secure channel. The client's handshaking code changes only the last line of the server's code. The client uses the counterpart to `SSL_accept()` called `SSL_connect()`.

```
SSL *ssl;
ssl = SSL_new(ctx); /* create new SSL connection state */
SSL_set_fd(ssl, server); /* attach the socket descriptor */
SSL_connect(ssl); /* perform the connection */
/*...*/
SSL_free(ssl); /* release SSL state */
```

The client uses during the session the `SSL_read()/SSL_write()`, and when done it releases the session resources with `SSL_free()`.

Dealing with the Limitations

The OpenSSL library makes writing standard and secure network programs fairly easy. As you have seen in this article, the interface is straightforward and not really an obstacle. However, for all its ease of use it does present some limitations. These limitations are not showstoppers, but you may need to prepare yourself for them.

One limitation found in the library is *session recovery*. Session recovery is a feature in highly reliable network programs that are able to restart a session where it left off. OpenSSL does not appear to include this feature in the API. The simplicity of the API (you accept the raw connection) contributes to the flaw. The program creates a streaming socket and then hands it off to the OpenSSL library. In a sense, OpenSSL never really *owns* the socket. So, if the peer connection is lost, the library cannot restart the connection and resume the session. Instead, if you are careful, you can use checkpointing during the session, and when a client reconnects after a communication break, the program moves to the last known checkpoint.

Securing Sockets with OpenSSL (part 2)

Another surprise you may encounter is the protocol itself. For some reason Netscape 4.7 only connects to SSLv2 (it then claims that the connection is SSLv3). Later versions may not exhibit this problem, but you may have to support several protocols in you released code. Fortunately, supporting multiple protocols is fairly simple.

Lastly, OpenSSL is a moving target. The version as of this writing is 0.9.6a. Change happens. Also, the API is good and solid but not 100% documented, making development a little challenging. So, as you develop programs, you may want to subscribe to the developers' message forum to gain insight and get help if needed.

Looking at the Security Landscape

Despite the limitations OpenSSL is a production-ready API you can use right now in your secure network programming. It is easy to use and provides state-of-the-art ciphers (even some of the once-patented ones) that the industry is now charging \$1,000s for.

Beginning with an understanding of basic network programming, you can use tools like this to get the most out of your servers and clients. If you would like to know more about sockets programming and SSL, pick up a copy of Linux Socket Programming from SAMS. You can also get the complete source listings for the book and these articles at <http://www.cs.utah.edu/~swalton>.