OpenSSL Compatibility

10.07.2010

http://www.yassl.com
info@yassl.com
phone: +1 206 369 4800

I. Compatibility with OpenSSL

The CyaSSL and yaSSL embedded SSL libraries provide an OpenSSL compatibility header, ssl.h, to ease the transition into using yaSSL or CyaSSL. Our test beds for OpenSSL compatibility are stunnel and Lighttpd, which means that we build both of them with CyaSSL as a way to test our OpenSSL compatibility API. Experimental versions of both packages built with CyaSSL are available on our download page.

II. Differences Between CyaSSL and OpenSSL

There are several differences between CyaSSL and OpenSSL. Listed here are the most prominent:

1. CyaSSL builds are 20-40 times smaller than OpenSSL. Hence it is much more useful in embedded SSL implementations.

2. Standards support: CyaSSL supports TLS 1.1 and 1.2. OpenSSL does not support TLS 1.1 or 1.2.

3. CyaSSL was built with securing streaming media in mind. OpenSSL was built before streaming media was popular on the Internet. As such, CyaSSL supports the latest streaming ciphers like Rabbit and HC-128 where OpenSSL does not.

4. License: CyaSSL is dual licensed under the GPLv2 and commercial license, with a company behind the commercial license. OpenSSL does not have a clear license.

5. We have tried to apply Occam's razor as the guiding philosophy to our implementation of SSL. As such, our API focuses on the most critical and necessary functionality in order to simplify the problem. CyaSSL has 20 or so function calls and an additional 230 for our OpenSSL compatibility layer. OpenSSL has over 3,500.

6. Really old code versus relatively new code: CyaSSL was written starting in 2004. OpenSSL started in 1995. Coding standards and requirements are a lot different today. OpenSSL has a longer legacy to support and maintain.

7. The OpenSSL legacy code comes from supporting usage profiles and operating systems that are no longer mainstream. The legacy code makes OpenSSL easier to break and harder to fix.

8. OpenSSL was written as the SSL/TLS standards were being defined. OpenSSL's code went down a number of blind alleys. We had the luxury of writing our code once the standards were well settled.

III. Supported OpenSSL Structs

SSL_METHOD holds SSL version information and is either a client or server method.
SSL_CTX holds context information including certificates.
SSL holds session information for a secure connection.

IV. Supported OpenSSL Functions

The three structures are usually initialized in the following way:

```
SSL_METHOD* method = SSLv3_client_method();
SSL_CTX* ctx = SSL_CTX_new(method);
SSL* ssl = SSL_new(ctx);
```

This establishes a client side SSL version 3 method, creates a context based on the method, and initializes the SSL session with the context. A server side program is no different except that the SSL_METHOD is created using *SSLv3_server_method()*.

When an SSL connection is no longer needed the following calls free the structures created during initialization.

```
SSL_CTX_free(ctx);
SSL_free(ssl);
```

SSL_CTX_free() has the additional responsibility of freeing the associated SSL_METHOD. Failing to use the XXX_free() functions will result in a resource leak. Using the system's free() instead of the SSL ones results in undefined behavior.

Once an application has a valid SSL pointer from SSL_new(), the SSL handshake process can begin. From the client's view, SSL_connect() will attempt to establish a secure connection.

```
SSL_set_fd(ssl, sockfd);
SSL_connect(ssl);
```

Before the SSL_connect() can be issued, the user must supply yaSSL with a valid socket file descriptor, sockfd in the example above. sockfd is typically the result of the TCP function socket() which is later established using TCP connect(). The following creates a valid client side socket descriptor for use with a local yaSSL server on port 11111, error handling is omitted for simplicity.

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
sockaddr_in servaddr;
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(11111);
```

```
servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
connect(sockfd, (const sockaddr*)&servaddr, sizeof(servaddr));
```

Once a connection is established, the client may read and write to the server. Instead of using the TCP functions send() and receive() CyaSSL and yaSSL use the SSL functions SSL_write() and SSL_read(). Here is a simple example from the client demo:

```
char msg[] = "hello yassl!";
int wrote = SSL_write(ssl, msg, sizeof(msg));
char reply[1024];
int read = SSL_read(ssl, reply, sizeof(reply));
reply[read] = 0;
printf("Server response: %s\n", reply);
```

The server connects in the same way except that is uses SSL_accept() instead of SSL_connect(), analogous to the TCP API. See the server example for a complete server demo program.

V. x509 Certificates

Both the server and client can provide CyaSSL and yaSSL with certificates in either PEM or DER. Typical usage is like this:

```
SSL_CTX_use_certificate_file(ctx, "certs/cert.pem", SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "certs/key.der", SSL_FILETYPE_ASN1);
```

A key file can also be presented to the Context in either format. SSL_FILETYPE_PEM signifies the file is PEM formatted while SSL_FILETYPE_ASN1 declares the file to be in DER format. To verify that the key file is appropriate for use with the certificate the following function can be used:

```
SSL_CTX_check_private_key(ctx);
```