yaSSL Architecture and Design

10.11.2010

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

yaSSL
1627 West Main St., Suite 237
Bozeman, MT 59715 USA
+1 206 369 4800
support@yassl.com
www.yassl.com

I. General Architecture

The yaSSL embedded SSL library implements the SSL 3.0, TLS 1.0, TLS 1.1 and TLS 1.2 protocols. TLS 1.2 is currently the most up to date and secure version of the standard. The TLS protocol is implemented as defined in RFC 5246. Two record layer protocols exist within SSL, the message layer and the handshake layer. Handshake messages are used to negotiate a common cipher suite, create secrets, and enable a secure connection. The message layer encapsulates the handshake layer while also supporting alert processing and application data transfer.

II. Cryptography

By default, yaSSL uses the cryptographic services provided by TaoCrypt, which is based in part on CryptoPP (see below). TaoCrypt aims to be more portable while only providing the functionality necessary for SSL-type needs. TaoCrypt Provides MD2, MD4, MD5, SHA-1, RIPEMD, HMAC, DES, 3DES, AES, ARC4, RSA, DSS, DH, and PKCS#5 PBKDF2, SHA-2, Random Number Generation, Large Integer support, and base 16/64 encoding/decoding. We've also added the Rabbit and HC-128 ciphers, public domain stream ciphers from the EU's eSTREAM project. Rabbit is potentially useful to those encrypting streaming media in high performance, high demand environments. RABBIT is nearly twice as fast as RC4 and HC-128 is about 5 times as fast! TaoCrypt is currently in the process of getting FIPS 140-2 level one validation. Should you need immediate support for a FIPS validated crypto module, then see CryptoPP below.

Descriptions and explanations of the algorithms and protocols that are used by yaSSL products can be found on our Algorithm and Protocol Reference page.

III. CryptoPP

CryptoPP can also be used to handle cryptography and crypto related details. Just define USE_CYPTOPP_LIB while compiling yaSSL. RSA, DES, 3DES, ARC4, MD5, SHA-1, and DSS are currently used by yaSSL, as well as CryptoPP's cryptographically secure random number generator, large Integer support, and base64 encoding/decoding. yaSSL declares all of its cryptographic needs in the header file crypto_wrapper.hpp. Nothing in the header file relies on the details of CryptoPP, i.e., any other library may be substituted in place of CryptoPP by switching out the desired functionality in crypto_wrapper.cpp.

More information about CryptoPP, including source download, may be obtained at cryptopp.com. Should you need a FIPS 140-2 level one crypto module for your use with yaSSL then Cryptopp is the crypto library you should use.

IV. TCP/IP

TCP/IP is responsible for all data transmission. Unreliable protocols are not currently supported by SSL. The system's TCP/IP is used by default and yaSSL uses it in a standard way. yaSSL does attempt to encapsulate the difference between BSD sockets and WinSock in socket_wrapper.hpp so that yaSSL does not have to concern itself with the discrepancies.

Please note that the user of yaSSL is responsible for setting socket options before passing off the socket descriptor. *For example, a user wishing to use a non-blocking socket must set this before handing off responsibility to yaSSL.*

For more information about TCP and socket options see your system documentation or a classic tome like Stevens's Unix Network Programming Volume I.

V. Certificates

TaoCrypt currently handles all x509 processing and verification. Extracting public keys, valid dates, issuer, and contact information are a few examples of certificate processing. TaoCrypt also verifies x509 validity through Certificate Authorities that the user controls. cert_wrapper.hpp declares the certificate functionality that yaSSL requires and like crypto_wrapper.hpp none of the details of TaoCrypt are used in the header so that another library can easily be substituted into cert_wrapper.cpp. yaSSL can accept certificates and keys in both PEM (Privacy Enhanced Mail) and DER (Distinguished Encoding Rules) formats.

VI. Input/Output Strategy

The yaSSL embedded SSL library uses a simulated streaming input/output system defined in buffer.hpp. The buffers behave like a smart C style array for use with overloaded << and >> and provide a checking option, input is designed to be read from and output is for writing. std::vector is not used because of a desire to have checked [] access, offset, and the ability to read/write to the buffer bulk wise while maintaining the correct size and offset. The buffers themselves use a checking policy.

The checking policy is Check by default but may be turned off at compile time to use NoCheck. Check merely ensures that range errors are caught and is especially useful for debugging and testing, though it is a simple in-lined test, some users may prefer to avoid runtime buffer flow checks.

One other feature worth noting about the buffers is that since they know their current offset, an index is not required for operator[]. By passing the constant AUTO, the user is freed from making silly index tracking errors, easing the burden of simple, but still error-prone, input/output programming. For example, compare the following two implementations:

A.
```
// input operator for ServerHello
input_buffer& operator>>(input_buffer& input, ServerHello& hello)
{
        // Protocol
        hello.server_version_.major_ = input[AUTO];
        hello.server_version_.minor_ = input[AUTO];

        // Random
        input.read(hello.random_, RAN_LEN);

        // Session
        hello.id_len_ = input[AUTO];
        input.read(hello.session_id_, ID_LEN);

        // Suites
        hello.cipher_suite_[0] = input[AUTO];
        hello.cipher_suite_[1] = input[AUTO];

        // Compression
        hello.compression_method_ = CompressionMethod(input[AUTO]);
        return input;
}
```

B.
```
// input operator for ServerHello
input_buffer& operator>>(input_buffer& input, ServerHello& hello)
{
        size_t i = input.get_current();

        // Protocol
        hello.server_version_.major_ = input[i++];
        hello.server_version_.minor_ = input[i++];

        // Random
        input.read(hello.random_, RAN_LEN);
        i += RAN_LEN;

        // Session
        hello.id_len_ = input[i++];
        input.read(hello.session_id_, ID_LEN);
        i += ID_LEN;

        // Suites
        hello.cipher_suite_[0] = input[i++];
        hello.cipher_suite_[1] = input[i++];
```

```
        // Compression
        hello.compression_method_ =
        CompressionMethod(input[i++]);

        input.set_current(i);

        return input;
}
```

While B is not much more difficult to implement, the chances for simple errors to occur are increased. Not to mention having to remember to get/set the current offset before passing the buffer to handlers and in the event of exceptions, there is no guarantee that the index is correctly positioned, making recovery nearly impossible.

VII. Factory Usage

Factories are used in several areas by yaSSL. Many of the structures defined in the SSL standard obey the Liskov Substitution Principle (Functions that use pointers/references to base classes must be able to use objects of derived classes transparently). That is, a ServerHello message IS A handshake message and a Finished message IS also A handshake message. Moreover, objects of the derived classes need to be created at runtime based on the negotiated parameters of the connection and message types. For example, when a message header is read, a type field identifies the message that follows. Instead of using a switch statement that becomes increasingly harder to maintain, yaSSL uses a message factory to create the desired object.

```
Message* msg = MessageFactory.CreateObject(hdr.type_);
```

factory.hpp defines the generic implementation like this:

```
template
<
        class AbstractProduct,
        typename IdentifierType = int,
        typename ProductCreator = AbstractProduct* (*)()
>
class Factory {
        typedef std::map CallBackMap;
        CallBackMap callbacks_;
...}
```

The Message Factory instance is created like this:

```
typedef Factory MessageFactory;
```

For more information on factories please see the design pattern discussion in (GoF) and Alexandrescu's chapter in Modern C++ Design.

VIII. Cryptographic Policies

Cryptographic Policies are employed to simplify yaSSL's use of digests, ciphers, and signature systems. Each is briefly described and defined in crypto_wrapper.hpp.

Digests, or MACs (Message Authentication Codes) use a basic policy that hides the underlying implementation:

```
struct MAC {
        virtual void get_digest(byte*) = 0;
        virtual void get_digest(byte*, const byte*, unsigned int) = 0;
        virtual void update(const byte*, unsigned int) = 0;
...}
```

Really only the first get_digest() and update() are needed but the extended version of get_digest() allows a user to both update and retrieve the digest in the same call, simplifying some operations. MD5 and SHA use the policy in their definitions, here is SHA as an example:

```
class SHA : public MAC {
public:
        void get_digest(byte*);
        void get_digest(byte*, const byte*, unsigned int);
        void update(const byte*, unsigned int);
...}
```

So when yaSSL has a MAC pointer, it uses it without knowledge of the derived object actually being used, conforming to the Liskov Substitution Principle.

Ciphers also employ a basic policy:

```
struct BulkCipher {
        virtual void encrypt(byte*, const byte*, unsigned int) = 0;
        virtual void decrypt(byte*, const byte*, unsigned int) = 0;
        virtual void set_encryptKey(const byte*, const byte* = 0) = 0;
        virtual void set_decryptKey(const byte*, const byte* = 0) = 0;
...}
```

These functions are all necessary and yaSSL uses BulkCipher pointers transparently and without knowledge of whether the actual object is DES, 3DES, or RC4. Authentication policies define the signature verification interface used by yaSSL.

```
struct Auth {
        virtual void sign(byte*, const byte*, unsigned int, const RandomPool&) =0;
        virtual bool verify(const byte*, unsigned int, const byte*, unsigned int) = 0;
...}
```

The authentication policy is straight forward and support for DSS and RSA is built into yaSSL.

IX. Thread Safe

yaSSL is thread safe by design. Multiple threads can enter the library simultaneously without creating conflicts because yaSSL avoids global data, static data, and the sharing of objects. The user must still take care to avoid potential problems in two areas.

A client may share an SSL object across multiple threads but access must be synchronized, i.e., trying to read/write at the same time from two different threads with the same SSL pointer is not supported.

yaSSL could take a more aggressive (constrictive) stance and lock out other users when a function is entered that cannot be shared but this level of granularity seems counter-intuitive. All users (even single threaded ones) will pay for the locking and multi-thread ones won't be able to re-enter the library even if they aren't sharing objects across threads. This penalty seems much too high and yaSSL leaves the responsibility of synchronizing shared objects in the hands of the user.

Besides sharing SSL pointers, users must also take care to completely initialize an SSL_CTX before passing the structure to SSL_new(). The same SSL_CTX can create multiple SSLs but the SSL_CTX is only read during SSL_new() creation and any future (or simultaneous changes) to the SSL_CTX will not be reflected once the SSL object is created.

Again, multiple threads should synchronize writing access to a SSL_CTX and it is advised that a single thread initialize the SSL_CTX to avoid the synchronization and update problem described above.

X. Thread Caching

yaSSL supports session caching, which can greatly decrease the connection processing time when clients re-issue connects within a relatively short time (the default is to cache for 500 seconds). Access to the cache is internally stored in yaSSL with a Singleton and is the only part of the library which isn't thread safe. For users that aren't multi-threaded or don't plan on using session caching and don't want to pay for the locking of this cache, please define SINGLE_THREADED when compiling yaSSL.

XI. Memory Usage

yaSSL doesn't pre-allocate any memory up front for a connection. This allows users to create a large pool of connection objects without paying for a large memory hit. yaSSL only requests memory when it needs it, and returns it to the system when it no longer needs it. yaSSL requires about 32K of memory for a connection (this includes 6K for 3 certificates, and the total will vary depending on certificate size and number as well as the type of connection). After that, yaSSL only requires about 500 bytes of memory to

send or receive an SSL message. Users wishing to pre-allocate memory, or take over the memory handling in any way, can implement their own ::operator new and delete.

XII. Supported Cipher Suites

The following Cipher Suites are supported by yaSSL:

SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_RMD160
TLS_DHE_DSS_WITH_AES_128_CBC_RMD160
TLS_DHE_DSS_WITH_AES_256_CBC_RMD160
TLS_DHE_RSA_WITH_3DES_EDE_CBC_RMD160
TLS_DHE_RSA_WITH_AES_128_CBC_RMD160
TLS_DHE_RSA_WITH_AES_256_CBC_RMD160
TLS_RSA_WITH_3DES_EDE_CBC_RMD160
TLS_RSA_WITH_AES_128_CBC_RMD160
TLS_RSA_WITH_AES_256_CBC_RMD160

XIII. Removing C++ Symbols

yaSSL is implemented in C++. For those users wishing to use a version of yaSSL without any global C++ symbols, there is a way to achieve this using GCC. Just define YASSL_PURE_C when building yaSSL. No exceptions, RTTI, std library, or global allocators will be defined.

XIV. Conclusion

The combination of yaSSL's buffer strategy and factory use provides a simple paradigm used throughout the handshake and message layer implementations. Reading and processing an input message could not be simpler:

```
while(!buffer.eof()) {
        // each record
        RecordLayerHeader hdr;
        buffer >> hdr;
        while (buffer.get_current() < hdr.length_ + offset) {
                // each message in record
                std::auto_ptr msg(mf.CreateObject(hdr.type_));
                buffer >> *msg;
                msg->Process(buffer, ssl);
        }
        offset += hdr.length_
}
```

This same loop is used by both clients and servers and efficiently handles all message types. Handshake processing uses the exact same paradigm. Sending an SSL output messages in yaSSL is primarily the responsibilityof the output buffer and some simple helper functions like buildHeader() and buildOutput(). Their implementations are also simple.

```
void buildHeader(SSL& ssl, RecordLayerHeader& rlHeader, const Message& msg)
{
        ProtocolVersion pv = ssl.get_connection().version_;
        rlHeader.type_ = msg.get_type();
        rlHeader.version_.major_ = pv.major_;
        rlHeader.version_.minor_ = pv.minor_;
        rlHeader.length_ = msg.get_length();
}

void buildOutput(output_buffer& buffer, const RecordLayerHeader& rlHdr,
                                                const HandShakeHeader& hsHdr, const
HandShakeBase& shake)
{
        buffer.allocate(RECORD_HEADER + rlHdr.length_);
        buffer << rlHdr << hsHdr << shake;
}
```

In fact, using the above functions and a couple of other helpers reveal the ease with which yaSSL sends a client hello message:

```
void sendClientHello(SSL& ssl)
{
        ClientHello ch;
        RecordLayerHeader rlHeader;
        HandShakeHeader hsHeader;
        output_buffer out;
        buildClientHello(ssl, ch);
        ssl.set_random(ch.get_random(), client_end);
        buildHeaders(ssl, hsHeader, rlHeader, ch);
        buildOutput(out, rlHeader, hsHeader, ch);
        hashHandShake(ssl, out);
        ssl.get_socket().send(out.get_buffer(), out.get_size());
}
```

Please see handshake.cpp and yassl_imp.cpp for a better understanding of how yaSSL sends and retrieves the other messages and handshake types.