A short introduction to OpenSSL

This is not a complete picture but should be enough for your IoT project.

At the highest level, OpenSSL provides data *authentication* (able to uniquely identify sender/receiver), data *integrity* (guarantee that messages weren't tampered with) and *encryption* (message scrambling rendering contents opaque to anyone but the intended recipient) at the **transport level** of the network stack. There are other ways to provide these features, such as IPsec at the link level (done in switches/routers instead of an end host/server). You can picture OpenSSL (Secure *Socket* Layer) sitting between your application and the socket - the socket being a door to the network.

I'm going to take a detour and try to give some high level ideas of cryptography concepts (I'm in no way an expert but it's some useful background).

Encryption methods are typically split in 2 families:

- **symmetric key ciphers** : There is a (large) set of algorithms that take as input plaintext and a key and output ciphertext. Given the ciphertext alone, recovering the key and message is impossible. Given the key and the ciphertext, the plaintext can be recovered.

Encryption is easy : both recipients have a common key, use it to encrypt and decrypt. (See it as a safe box: put the message in the box, anyone with the key can open the box and reveal the message).

This poses a problem: how does one safely share the key? Before one does, one can't perform encryption, so one can't transmit the key…

In practice the second family of method is used:

- **asymmetric (public/private) key ciphers**: There are some simple explanations for this online. Essentially, imagine that you have a safe box , this time with a 3-position lock:

- Left: locked

- Middle: unlocked

- Right: locked (this is the initial position)

You have 2 keys for this box. One allows you to go right only, and the other left only.

You give the 'left' key to absolutely everyone, even potential attackers (public key). They can go to the box, and use the public key to open the lock (right -> middle), place the message, and lock it (middle -> left). The public key doesn't allow for (middle->right) or (left->middle).

Once the box is on left position, only the 'right' key can open it back up.

Only you have that key (private key).

Bottom line: anyone can send you an encrypted message, only you can decrypt it. This method is computationally expensive (involves taking large exponents), therefore it is only used to share a symmetric key. Once the symmetric key has been safely shared, it can be used for the rest of the communication.

-Now, onto *integrity*. One can use MACs (Message Authentication Codes), that are Hashes of the plaintext message. The hash function chosen has an interesting property : two different messages will have two different MACs. This is interesting: any modification in the message will incur a modification in the MAC.

How does integrity work?

Send encrypted content combining (message, MAC). An attacker can temper with the encrypted message. If this encrypted message is modified in any way, there is no chance that the message will still match its MAC.

The recipient can thus detect any tampering -> extract message and MAC, and recompute the MAC from the message, using the same algorithm used by the sender. If the computed MAC and the sent MAC are the same, the message hasn't been modified.

- *Authentication*: Read up on digital signatures. The basic concept is that authentication is performed by certifying the owner of a public key. This is done by an entity known as a Certificate Authority (a private company, for example). The owner of the public key is the only entity that will be able to read messages encrypted with that public key.

All of this depends on the capacity of a user to **generate pseudo-random numbers**.

I'll return the the topic of OpenSSL. The path of your application level data from source to destination is:

(Application) -> (OpenSSL) -> (Network) -> (OpenSSL) -> (Application).

How do you set-up the OpenSSL Block?

0) **Initialization:**

For anything to happen, you need to initialize the library (so it can, among other things, initialize its random number generation engine). This also creates a table filled with cipher and digest lookup functions. These functions are used by a lot of the functions that are defined in the library. If the table of 'lookups' isn't defined, the OpenSSL calls you'll use will complain that algorithms can't be found. To do so, call SSL_library_init()/add_all_algorithms

1) **BIOs**

SSL needs to be able to perform I/O (from Application level data to network level data). For that, it uses BIOs, which are I/O abstraction objects (they abstract sockets, terminals, memory buffers ... all the normal objects you'd use to perform I/O and that you've been using for this class). Think of it as a file handle. Why do we use that abstraction? It allows to specify the API of the SSL library functions as taking BIO objects as input, regardless of what that object actually is.

For example, when you're calling the load_error_strings() functions, you're just linking a BIO to stderr for error reporting. You should do that after initializing to help debugging.

2) **OpenSSL methods**

OpenSSL needs to be able to perform actual encryption/authentication/guarantee integrity. You can pick several protocols that do just that, and you have specify which you want to use. This choice of protocol is referred to as a *method*, such as SSLv1, TLSv1.2, ...

TLS stands for Transport Layer Security, see it as an upgraded SSL protocol.

3) **OpenSSL context**

A context is a collection of settings. Given a method, a *context* will sum up all the information required to establish a connection. Establishing a connection requires a handshake, negotiating with the server to use a particular set of ciphers, hash functions, protocol versions … It also describes behavior on shutdown.

Therefore, when you want to initlialize a new SSL connection, all you have to refer to is the *context*. This saves both time and memory. The *context* depends on the *method* you're using.

4) **SSL Session**

Now that we gathered all the information we need to start a connection, we need to establish it. Information about an established connection will be stored in an SSL *session*. It holds less information than a context: just the exchanged key and the agreed upon protocol version and ciphers. Calling SSL_new() will initialize such a *session*.

5) **Linking a session and a socket**

Remember : your application will give data to the OpenSSL ‘block’. That ‘block' will output encrypted messages to the socket, that is linked with the recipient of the messages. You have to link the ‘block’ and the socket to complete the path. SSL_set_fd creates a BIO, see it as a link between that SSL box and the socket file descriptor. You now have a path from your application to the destination.

6) **Handshake, negotiate protocols, connect**

SSL_connect() fills out the SSL session object that you previously created. Source and destination will agree on what protocols to use and operations to perform to protect your data. You can now communicate!

7) **SSL read/write**

See it as a normal call to read/write, but the SSL ‘block’ performs encryption and decryption in the background, using the information in the session to do so.

8) **Shutdown**

You initialized a lot of tables and buffers to use the library, you must now free the sessions/ contexts and lookup tables, through SSL_free and SSL_shutdown.