**Deep dive on SSL/TLS**

This document serves as an introduction and/or refresher on SSL/TLS security protocol library. A good understanding of SSL/TLS library will be essential in future debugging and/or further optimization of mbedTLS for FT900 and other FTDI products.

MbedTLS has been integrated to FT900 to enable secure IoT connectivity via MQTT protocol over LWIP running on Ethernet. This enables FT900 to communicate with Amazon AWS Greengrass and AWS IoT cloud which requires MQTT over TLS. FT900 also works with TLS-enabled local Mosquitto MQTT message broker.

**SSL / TLS Protocol**

1. The Transport Layer Security (**TLS**) **Handshake** Protocol is responsible for the **authentication** and **key exchange** necessary to establish secure/**encrypted** sessions.

## Cipher Suite Negotiation

## The client and server make contact and choose the cipher suite that will be used throughout their message exchange.

## Authentication

## In TLS, a server proves its identity to the client. The client optionally prove its identity to the server.

## X.509 public key/private key certificates is the basis for authentication.

## The authentication method depends on the negotiated ciphersuite.

## Key Exchange

The client and server exchange random numbers and a special number called the Pre-Master Secret.

These numbers are combined with additional data to create their shared secret, called the Master Secret.

The Master Secret is used by client and server to generate the write MAC secret, which serves as

session key used for [hashing](https://msdn.microsoft.com/en-us/library/ms721586(v=VS.85).aspx), and write key, which is the [session key](https://msdn.microsoft.com/en-us/library/ms721625(v=VS.85).aspx) used for encryption.

**TLS Handshake Protocol**

**// Client Hello and Server Hello**

1. CLIENT: sends "Client hello" (params: client random value, supported cipher suites)
2. SERVER: sends "Server hello" (param: server's random value, chosen cipher suite)
3. SERVER: sends “Certificate” (param: certificate (public key) for authentication)
4. SERVER: may send “CertificateRequest” to request a certificate from the client.
5. SERVER: sends "Server hello done".

**// Authentication and Pre-master secret**

1. CLIENT: authenticates certificate and may send “Certificate” & “CertificateVerify” (if server requested it)
2. CLIENT: sends “ClientKeyExchange” (param: encypted Pre-Master Secret for the session)

(random Pre-Master Secret encrypted with *public key* from the server's certificate)

**// Decryption and Master secret**

Server decrypts the **Pre-Master Secret** using its private key.

Client and server then generate the **Master Secret** with the agreed cipher.

**// Generate Session Keys**

Client and server then generate the **Session Keys** (symmetric) based on the Master Secret.

**// Change to Symmetric Encryption Mode (since asymmetric is slow)**

1. CLIENT: sends "Change cipher spec" to indicate that the client will start using the new *session keys* for *hashing* and encrypting messages.
2. CLIENT: sends "Client finished"
3. SERVER: sends “Chane cipher spec”. Then switches its record layer security state to *symmetric encryption* using the *session keys*.
4. SERVER: sends "Server finished"

Client and server can now exchange application data over the secured channel they have established.

All messages sent from client to server and from server to client are encrypted using session key.

**X.509 digital certificate**

1. Certificates are signed data structures that bind a public key to a person, computer, or organization.

It is issued & signed by **certification authorities** (CAs) like Verisign, Symantec, Comodo, DigiCert.

CAs issues certificates by computing a hash of the contents and signing the hash by CA private key.

When verifying, the parsed hash is decrypted using CA public key and compared with computed hash.

**Self-signed certificates** can be used as CA certificate but only for testing purposes, not for production.

1. The X.509 public key infrastructure (PKI) standard identifies requirements for robust public key certificates.

Public-key cryptography (aka asymmetric-key cryptography) uses a key pair to encrypt/decrypt content.

The public key is used to encrypt while a private key is used to decrypt.

1. The syntax for X.509 certificates is called ASN.1 Abstract Syntax Notation One.

The public key and private key can be in PEM or DER (binary) format.

The private key PEM format can be in PKCS#1 private key format (-----BEGIN RSA PRIVATE KEY-----).

**SHA hash algorithms**

1. SHA stands for Secure Hashing Algorithm.
2. A hashing algorithm is a mathematical function that condenses data to a fixed size which is

**irreversible and unique**.

* 1. Irreversible so that it is impossible to figure out what the original piece of data was.
  2. Unique so that two different pieces of data can’t produce the same hash, no collision.

1. SHA-1 is a 160-bit (20 bytes) hash.

SHA-2 is a family of hashes: SHA-256 (32 bytes) and alternative bit-lengths: SHA-224, SHA384, SHA512)

1. SHA is used in TLS/SSL as authentication of digital signature.

A digitial signature in a digital certificate is a hash of the contents signed by the CA private key.

When verifying the certificate, the hash is verified. The parsed hash is decrypted using CA public key and then compared with the computed hash.

**RSA key exchange**

1. RSA is an asymmetric encryption algorithm used for public key exchange.

It is used to securely agree on a symmetric key for data encryption.

Like DH, it also make use of modular exponentiation.

1. RSA Problem: I want anybody to be able to encrypt a message, but I'm the only one who can decrypt it. I don't want to share decryption keys with anybody.

RSA Weakness: Integer factorization

1. DH Problem: We have a symmetric encryption scheme and want to communicate. We don't want anybody else to have the key, so we can't say it out loud (or over a wire).

DH Weakness: Discrete algorithm

<https://crypto.stackexchange.com/questions/42180/whats-the-difference-between-rsa-and-diffie-hellman>

**AES symmetric encryption algorithm**

1. The Advanced Encryption Standard, or AES, is a symmetric block cipher used to encrypt sensitive data

It is the default encryption algorithm for protecting classified information by U.S. government.

Symmetric (also known as secret-key) ciphers use the same key for encrypting and decrypting, so the sender and the receiver must both know -- and use -- the same secret key.

1. AES comprises three block ciphers: AES-128, AES-192 and AES-256.

Each cipher encrypts and decrypts data in blocks of 128 [bits](https://whatis.techtarget.com/definition/bit-binary-digit) (16 bytes) using cryptographic keys of 128-, 192- and 256-bits

In TLS, this is used in the actual encryption/decryption of data transfer.

**TLS client usage**

1. Setup a TCP/IP connection (TCP/IP interface)
   1. lSocket = lwip\_socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);
   2. lwip\_connect(lSocket, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr));
2. Initialize as an SSL-client TLS (SSL/TLS interface)
   1. mbedtls\_ssl\_init(&ssl\_ctx);
   2. mbedtls\_ssl\_config\_init(&ssl\_conf);
   3. mbedtls\_ctr\_drbg\_init(&drbg\_ctx);
   4. mbedtls\_entropy\_init(&entropy\_ctx);
   5. mbedtls\_ctr\_drbg\_seed(&drbg\_ctx, mbedtls\_entropy\_func, &entropy\_ctx, pers, strlen(pers));
   6. mbedtls\_ssl\_config\_defaults(&ssl\_conf,

MBEDTLS\_SSL\_IS\_CLIENT, MBEDTLS\_SSL\_TRANSPORT\_STREAM, MBEDTLS\_SSL\_PRESET\_DEFAULT);

* 1. mbedtls\_ssl\_conf\_rng(&ssl\_conf, mbedtls\_ctr\_drbg\_random, & drbg\_ctx);
  2. mbedtls\_ssl\_setup(&ssl\_ctx, &ssl\_conf);
  3. mbedtls\_ssl\_set\_bio(&ssl\_ctx, &lSocket, \_TLS\_send, \_TLS\_recv, \_TLS\_recv\_timeout );

1. Load the trusted CA certificates (X.509 interface)
   1. mbedtls\_x509\_crt\_init(&ca);
   2. mbedtls\_ssl\_conf\_authmode(&ssl\_conf, MBEDTLS\_SSL\_VERIFY\_REQUIRED);
   3. mbedtls\_x509\_crt\_parse(&ca, ca\_cert, strlen(ca\_cert) + 1);
   4. mbedtls\_ssl\_conf\_ca\_chain(&ssl\_conf, &ca, NULL);
2. Load your certificate and your private RSA key (X.509 interface)
   1. mbedtls\_x509\_crt\_init(&cert);
   2. mbedtls\_x509\_crt\_parse(&cert, cli\_cert, strlen(cli\_cert) + 1);
   3. mbedtls\_pk\_init(&key);
   4. mbedtls\_pk\_parse\_key(&key, cli\_key, strlen(cli\_key) + 1, NULL, 0);
3. Perform an SSL-handshake (SSL/TLS interface)
   1. mbedtls\_ssl\_handshake(&ssl\_ctx)
4. Verify the server certificate (SSL/TLS interface)
   1. mbedtls\_ssl\_get\_verify\_result(&ssl\_ctx);
5. Write/read data (SSL/TLS interface)
   1. mbedtls\_ssl\_write(&ssl\_ctx, pvBuffer, xDataLength)
   2. mbedtls\_ssl\_read(&ssl\_ctx, pvBuffer, xBufferLength)
6. Close and cleanup (all interfaces)
   1. lwip\_close(lSocket);
   2. mbedtls\_ssl\_free(&ssl\_ctx);
   3. mbedtls\_ssl\_config\_free(&ssl\_conf);
   4. mbedtls\_ctr\_drbg\_free(&drbg\_ctx);
   5. mbedtls\_entropy\_free(&entropy\_ctx);

To view the complete source code, please refer to secure\_sockets.c

**TLS client configuration**

1. The TLS client configuration for FT900 AWS IoT demo only enables the following ciphersuites due to memory constraints.
   1. MBEDTLS\_TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA
   2. MBEDTLS\_TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA

These ciphersuites where chosen because these are the minimum ciphersuites supported by Amazon AWS Greengrass. Refer to <https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-sec.html>

Moreover, these ciphersuites are also supported by Amazon AWS IoT. Refer to <https://docs.aws.amazon.com/iot/latest/developerguide/iot-security-identity.html> (Note that AES128-SHA and AES256-SHA covers the ciphersuites above. This has been proven by the successful test results.)

Additional ciphersuites may be later added to support Microsoft Azure and Google Cloud.

1. To use these 2 ciphersuites, the following macros and its dependencies are enabled:

|  |  |
| --- | --- |
| TLS related configuration |  |
| MBEDTLS\_SSL\_TLS\_C | TLS requirement |
| MBEDTLS\_SSL\_CLI\_C | TLS client requirement |
| MBEDTLS\_CIPHER\_C | required by MBEDTLS\_SSL\_TLS\_C |
| MBEDTLS\_MD\_C | required by MBEDTLS\_SSL\_TLS\_C |
| MBEDTLS\_ENTROPY\_C | required by TLS client usage mbedtls\_entropy\_xxx() |
| MBEDTLS\_CTR\_DRBG\_C | required by TLS client usage mbedtls\_ctr\_drbg\_xxx() |
| MBEDTLS\_SSL\_PROTO\_TLS1\_2 | required by AWS IoT, not by AWS Greengrass |
| RSA related configuration |  |
| MBEDTLS\_KEY\_EXCHANGE\_RSA\_ENABLED | required by our chosen ciphersuites |
| MBEDTLS\_RSA\_C | required by our chosen ciphersuites |
| MBEDTLS\_OID\_C | required by MBEDTLS\_RSA\_C |
| MBEDTLS\_PKCS1\_V15 | required by MBEDTLS\_RSA\_C |
| MBEDTLS\_BIGNUM\_C | required by MBEDTLS\_RSA\_C |
| X509 Certificate related configuration |  |
| MBEDTLS\_X509\_CRT\_PARSE\_C | required by MBEDTLS\_KEY\_EXCHANGE\_RSA\_ENABLED |
| MBEDTLS\_X509\_USE\_C | required by MBEDTLS\_X509\_CRT\_PARSE\_C |
| MBEDTLS\_ASN1\_PARSE\_C | required by MBEDTLS\_X509\_USE\_C |
| MBEDTLS\_PK\_PARSE\_C | required by MBEDTLS\_X509\_USE\_C |
| MBEDTLS\_PK\_C | required by MBEDTLS\_PK\_PARSE\_C |
| MBEDTLS\_PEM\_PARSE\_C | required by format of our given X509 certificates |
| MBEDTLS\_BASE64\_C | required by MBEDTLS\_PEM\_PARSE\_C |
| SHA Hash related configuration |  |
| MBEDTLS\_SHA1\_C | required by our chosen ciphersuites |
| MBEDTLS\_SHA256\_C | required by MBEDTLS\_ENTROPY\_C |
| AES symmetric encryption related configuration |  |
| MBEDTLS\_AES\_C | required by our chosen ciphersuites |
| MBEDTLS\_CIPHER\_MODE\_CBC | required by our chosen ciphersuites |
| Optimization related configuration |  |
| MBEDTLS\_SSL\_MAX\_CONTENT\_LEN | 1536 |
| MBEDTLS\_AES\_ROM\_TABLES | decreases code size by 896 bytes |
| MBEDTLS\_MPI\_WINDOW\_SIZE 1 | decreases code size by 808 bytes |
| MBEDTLS\_MPI\_MAX\_SIZE 256 | decreases code size by 64 bytes |
| MBEDTLS\_SHA256\_SMALLER | decreases code size by 2944 bytes |
| MBEDTLS\_TLS\_DEFAULT\_ALLOW\_SHA1\_IN\_KEY\_EXCHANGE |  |

To view the complete list of configurations enabled, please refer to mbedtls\_config.h

1. Server certificate verification is disabled by default. It is disabled to save some memory footprint. To enable server certificate verification, set the macro USE\_ROOTCA.
   1. Advantage: prevent man-in-the-middle attacks
   2. Disadvantage: enabling this adds 2kb memory footprint (for DEBUG mode).
2. Elliptic Curve Cryptography (ECC) ciphersuite support and AES\_GCM support are now supported using the macro USE\_CIPHERSUITE.
   1. CIPHERSUITE\_OPTION\_1: // AES\_CBC
      1. MBEDTLS\_TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA
      2. MBEDTLS\_TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA
   2. CIPHERSUITE\_OPTION\_2: // AES\_GCM is stronger than CBC
      1. MBEDTLS\_TLS\_RSA\_WITH\_AES\_128\_GCM\_SHA256
      2. MBEDTLS\_TLS\_RSA\_WITH\_AES\_256\_GCM\_SHA384
   3. CIPHERSUITE\_OPTION\_3: // ECDHE\_RSA is strong than RSA
      1. MBEDTLS\_TLS\_ECDHE\_RSA\_WITH\_AES\_128\_CBC\_SHA
      2. MBEDTLS\_TLS\_ECDHE\_RSA\_WITH\_AES\_256\_CBC\_SHA