12.1 Overview of TLS

Transport Layer Security (TLS) is a protocol that provides a secure channel between two communicating applications so that the data transmission in this channel is private and its integrity is preserved. TLS evolved from its predecessor SSL (Secure Sockets Layer), and is gradually replacing SSL. SSL was developed by Netscape to secure web communication. When the SSL protocol was standardized by the IETF, it was renamed to Transport Layer Security. SSL version 3.0, which is the most recent version of SSL defined in RFC 6101 [Freier et al., 2011], was deprecated in June 2015 and replaced by TLS. For this historic reason, the terms SSL, TLS, or TLS/SSL are used interchangeably. Technically, TLS and SSL are different protocols. In this chapter, we focus on TLS. At the time of writing, TLS version 1.2, defined in RFC 5246 [Dierks and Rescorla, 2008], is the most widely used version; TLS version 1.3 was defined in RFC 8446 [Rescorla, 2018] in August 2018.

The secure channel provided by TLS has the following three properties.

- *Confidentiality:* Nobody other than the two ends of the channel can see the actual content of the data transmitted via the channel.
- *Integrity:* If data are tampered by others during the transmission, the channel should be able to detect it.
- *Authentication:* In a typical scenario, at least one end of the channel (usually the server end) needs to be authenticated, so the other end (usually the client end) can be sure that it is communicating to the intended host. Without a proper authentication, the client might be unknowingly establishing a protected channel with an attacker.

TLS sits between the Application Layer and the Transport Layer, as Figure 12.1 shows. Unprotected data from an application are given to the TLS layer, which handles the encryption, decryption, and integrity checking tasks. TLS then gives the protected data to the underlying Transport layer for transmission. TLS is designed to run on top of the TCP protocol; however, it has also been implemented with datagram-oriented transport protocols, such as UDP. TLS over UDP has been standardized independently using the term Datagram Transport Layer Security (DTLS) [Rescorla and Modadugu, 2012].

TLS is a layered protocol, consisting of two layers. The bottom layer of TLS is called Record Layer, and the protocol at this layer is called TLS Record Protocol, which defines the format of the records used by TLS. When a host sends out a TLS message, whether the message is a control message or a data message, TLS puts the message in records. Each record contains a header, a payload, an optional MAC, and a padding (if needed). The payload part carries the actual message, the format of which depends on the protocols running on top of the Record Layer. There are five message protocols in TLS, including the Handshake, Alert, Change Cipher Spec, Heartbeat, and Application Protocols (see Figure 12.1).

The Alert protocol is used for peers to send signal messages to each other; its primary purpose is to report the cause of a failure. The Change Cipher Spec protocol is used to change the encryption method being used by the client and server. It is normally used as part of the handshake process to switch to symmetric key encryption. The Heartbeat protocol is used to keep TLS sessions alive. The most important protocols in TLS are the Handshake protocol and the Application protocol. The Handshake protocol is responsible for establishing the secure channel, while the Application protocol is used for actual data transmission using the channel. We will only focus on these two protocols in this chapter.

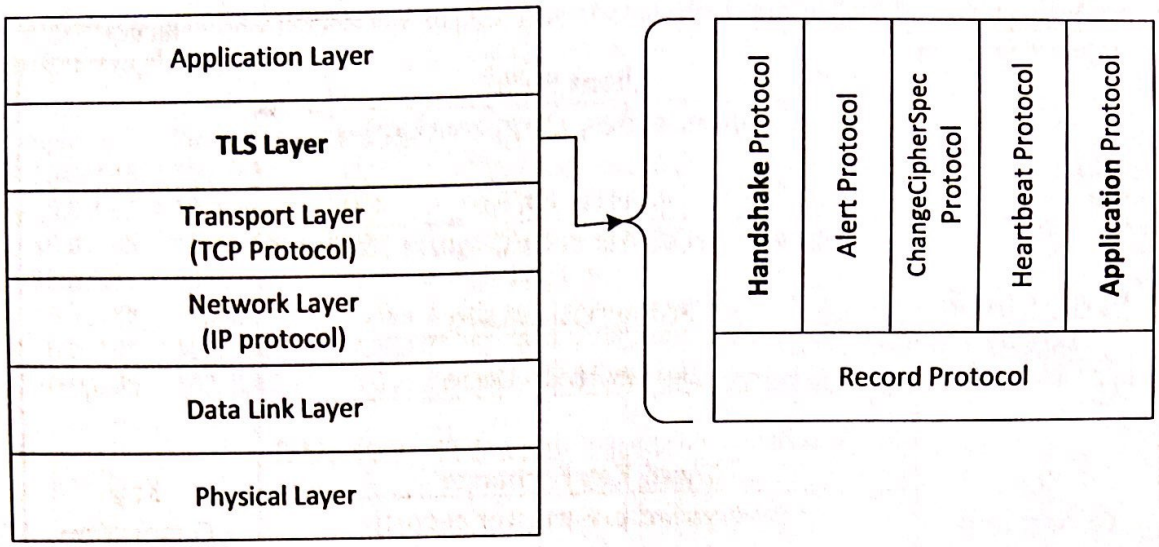

Figure 12.1: TCP/IP network stack with the TLS layer

12.2 TLS Handshake

Before a client and a server can communicate securely, several things need to be set up first, including what encryption algorithm and key will be used, what MAC algorithm will be used, what algorithm should be used for the key exchange, etc. These cryptographic parameters need to be agreed upon by the client and the server. That is the primary purpose of the TLS Handshake Protocol. In this section, we give an overview of the protocol, while emphasizing on two of its essential steps: certificate verification and key generation. Our discussion is based on TLS Version 1.2 [Dierks and Rescorla, 2008].

12.2.1 Overview of the TLS Handshake Protocol

The purpose of the TLS Handshake protocol is for the client and the server to agree upon cryptographic parameters, including cryptographic algorithms, session keys, and various other parameters. Figure 12.2 illustrates the steps of the TLS Handshake protocol. Details are further explained in the following.

- Client: Send a `Client Hello` message. When a client tries to establish a TLS connection with a server, it first sends a TLS hello message to the server. In this message, it tells the server which cipher suites are supported by the client. Moreover, it provides a random string called `client random`, which serves as a nonce for key generation.

- Server: Send a `Server Hello` message. Once the server receives the hello message from a client, it selects a cipher suite that is supported by both client and server, and sends back a message to inform the client about the decision. Moreover, it provides a random string called `server random`, which also serves as a nonce for key generation.

- Server: Send its certificate: The server sends its public-key certificate to the client. The certificate needs to be verified by the client, and the verification is crucial for security; we provide the details of the verification in §12.2.2.

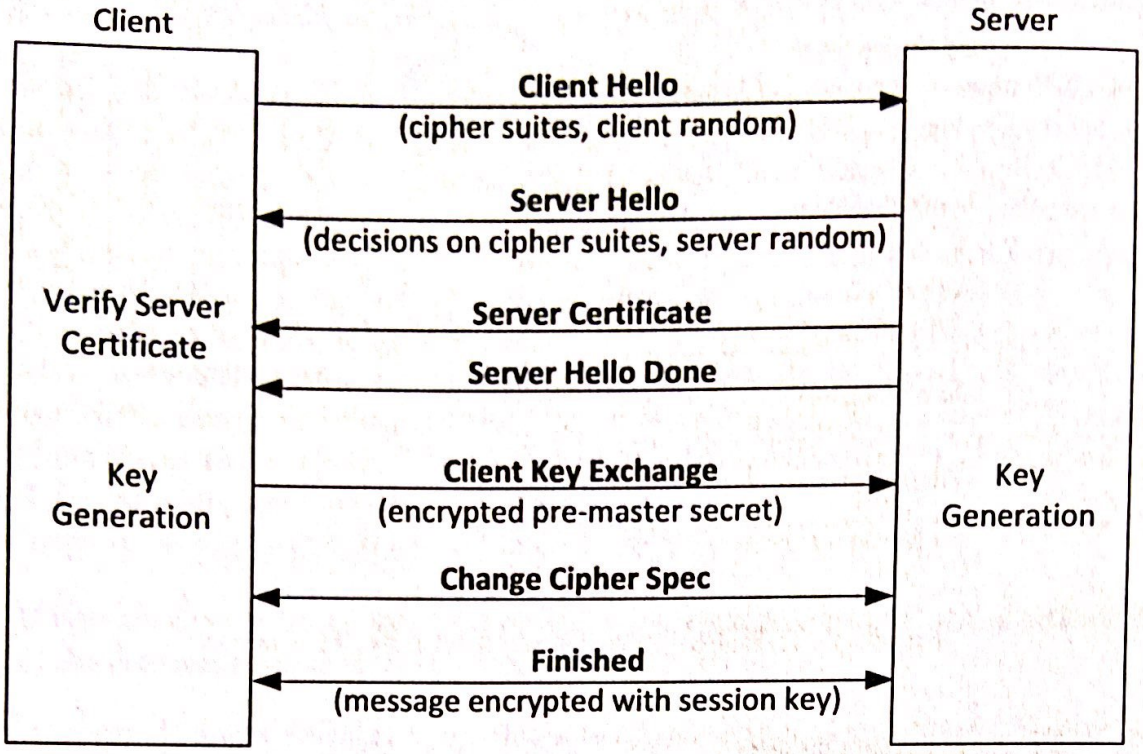

Figure 12.2: TLS handshake protocol

- Server: Send a `Server Hello Done` message, indicating that it is done with the handshake negotiation.
- Client: Send a `Client Key Exchange` message. The client generates a random pre-master secret, encrypts it using the server's public key obtained, and sends the encrypted secret to the server. Both client and server first use the pre-master secret to generate a master secret, and then further use the master secret to generate session keys, which will be used to secure the communication between the client and the server. Details of the key generation will be provided in §12.2.3.
- Client and Server: The client sends a `Change Cipher Spec` message to the server, telling the server that further communication from client to server will be authenticated and encrypted. The server does the same by sending a `Change Cipher Spec` message to the client.
- Client and Server: The client sends an encrypted `Finished` message, containing a hash and MAC over the previous handshake messages. The server will decrypt the message and verify the hash and MAC. If the verification fails, the handshake protocol fails, and the TLS connection will not be established. The server does the same by sending a Finished message to the client, which conducts the same verification.

Using Wireshark, we have captured the packets exchanged between a client and a server during the TLS handshake protocol. Table 12.1 shows the result. TLS runs on top of TCP, so before the TLS protocol runs, a TCP connection needs to be established first. Packets No.1 to No.3 are for the TCP three-way handshake protocol, which establishes a connection between the client and the server. After the connection is established, the client and the server run the

TLS handshake protocol (Pakcets 4 to 9). Note that some of the steps in the protocol are carried out using a single packet.

No.	Source	Destination	Protocol	Info
1	10.0.2.45	10.0.2.35	TCP	59930 –> 11110 [SYN] Seq=0 Win=14600 Len=0 MSS=1460...
2	10.0.2.35	10.0.2.45	TCP	11110 –> 59930 [SYN, ACK] Seq=0 Ack=1 Win=14480...
3	10.0.2.45	10.0.2.35	TCP	59930 –> 11110 [ACK] Seq=1 Ack=1 Win=14720 Len=0...
4	10.0.2.45	10.0.2.35	TLSv1.2	Client Hello
6	10.0.2.35	10.0.2.45	TLSv1.2	Server Hello, Certificate, Server Hello Done
8	10.0.2.45	10.0.2.35	TLSv1.2	Client Key Exchange, Change Cipher Spec, Finished
9	10.0.2.35	10.0.2.45	TLSv1.2	New Session Ticket, Change Cipher Spec, Finished

Table 12.1: TLS traffic captured by Wireshark

12.2.2 Certificate Verification

In the TLS Handshake Protocol, the client generates a secret (called pre-master secret) and sends it to the server. Both sides will use this secret to generate session keys, which are used for encryption and MAC. The pre-master secret has to be protected when it is sent to the server, so adversaries cannot see the secret. This protection is achieved using public-key encryption. Namely, the server sends its public key to the client, who uses this public key to encrypt the pre-master secret. As we have seen from Chapter 11 (Public Key Infrastructure), directly sending a public key over the network is subject to man-in-the-middle attacks. Instead, the server should send a valid public key certificate to the client. The certificate contains the server's public key and identity information, an expiration date, a CA's signature, and other relevant information.

When the client receives the server's certificate, it needs to ensure that the certificate is valid. The validation involves several checks, including checking the expiration date and most importantly, checking that the signature is valid. The signature checking requires the client to have the signing CA's public-key certificate. Client programs, such as browsers, need to load a list of trusted CA certificates beforehand, or they will not be able to verify any certificate. If the signing CA is on this list, the certificate can be directly verified; if not, the server needs to provide the certificates of all the intermediate CAs, so the client can verify them one by one, and eventually verify the server's certificate.

It should be noted that the above TLS validation only checks whether a certificate is valid or not, it does not check whether the identity information contained in the certificate matches with the identity of the intended server. The latter check is also essential for security, but it is the responsibility of applications. Without this check, we may be talking to `attacker32.com`, which impersonates the intended server `facebook.com`. The impersonator can provide a valid certificate of its own, and the certificate can pass TLS's validation, even though it has nothing to do with `facebook.com`. We will explain this in more details in §12.5.

12.2.3 Key Generation and Exchange

Although public-key algorithms can be used to encrypt data, it is much more expensive than secret-key encryption algorithms. For this reason, TLS only uses public-key cryptography for key exchange, i.e., enabling a client and a server to agree upon some common secret for key generation. Once the keys are generated, the client and server will switch to a more efficient secret-key encryption algorithm. The entire key generation consists of three steps: generating

pre-master secret, generating master secret, and finally generating session keys. Figure 12.3 illustrates these three steps.

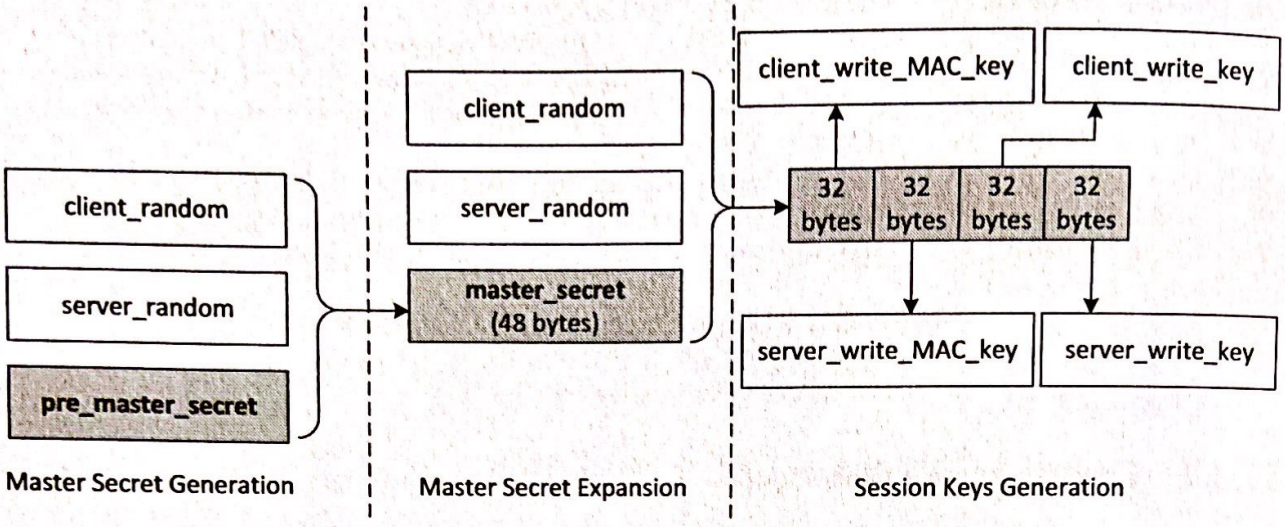

Figure 12.3: TLS key generation (master Secret and session keys)

Pre-master secret. After the server certificate has been verified successfully, the client program generates a random number, which is called pre-master secret. The length of the number depends on the key exchange algorithm. The secret, encrypted with the server's public key, is sent to the server. Since only the server has the corresponding private key, the encrypted pre-master secret can only be decrypted by the server. Therefore, the pre-master secret is only known to the client and the server.

Master secret. During the initial steps, the client and the server have exchanged two random numbers (nonces), `client_random` and `server_random`. Using these two numbers and the pre-master secret, both client and server generate another secret, called master secret, which is 48 bytes long.

Session keys. The master secret is then used to generate a sequence of bytes according to the cipher algorithm. This sequence is further split into four separate keys: two MAC keys and two encryption keys. The `client_write` keys (for both MAC and encryption) are used to secure the data from the client to the server, while the `server_write` keys are used to secure the data from the server to the client. The communication between the client and the server is bidirectional, and each direction uses its own keys for MAC and encryption. Figure 12.3 assumes that the cipher suite `AES_256_CBC_SHA` is used, so the key size is 32 bytes (256 bits).

Key generation in TLS renegotiation. TLS allows either the client or the server to initiate renegotiation to establish new cryptographic parameters, such as changing session keys. Instead of requiring the client and the server to conduct another round of handshake protocol, TLS provides an abbreviated handshake protocol for the renegotiation purpose. In the abbreviated protocol, the client and the server generate a new `client_random` and `server_random`, respectively, and send their numbers to each other. They then repeat the Master Secret Expansion

and Session Keys Generation steps as shown in Figure 12.3. The mater secret used in the process is the same as the one generated from the full handshake protocol, so there is no need for resending the server certificate or the pre-master secret. The abbreviated handshake simplifies the handshake process and improves the efficiency.

12.3 TLS Data Transmission

Once a client and a server have finished their TLC Handshake protocol, they can start exchanging application data. Data are transferred using records, the format of which is defined by the TLS Record protocol. Records are not only used for transferring application data; messages in the Handshake protocol are also transferred using records. Each record contains a header and a payload. There are three fields in the header.

- Content Type: TLS contains several protocols, including Alert, Handshake, Application, Hearbeat, and ChangeCipherSpec protocols. They all use the TLS Record Protocol to transfer data. The Content Type field indicates what type of protocol data is carried by the current record.
- Version: This field identifies the major and minor version of TLS for the contained message. As of July 2017, TLS supports the following versions: SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, and TLS 1.3.
- Length: The length of the payload field, not to exceed 2^{14} bytes.

In this section, we focus on the Application record type, which is used to transfer application data between a client and a server. The format of the record is depicted in Figure 12.4. The Content Type field for application records is `0x17`, while the Length field contains the length of the contained application data, excluding the protocol header but including the MAC and padding trailers.

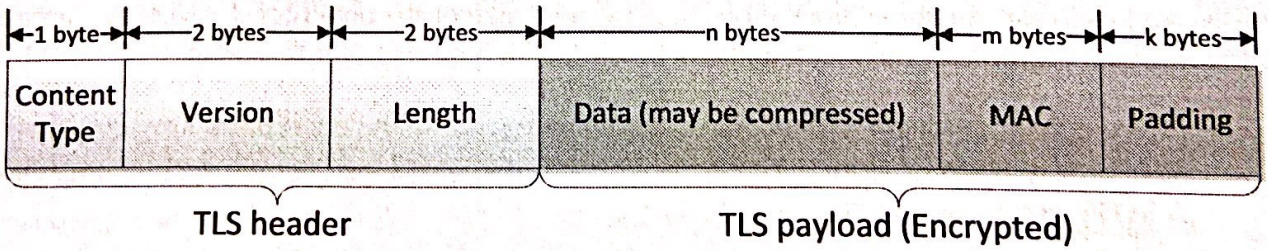

Figure 12.4: Record format of the TLS Record Protocol

12.3.1 Sending Data with TLS Record Protocol

To send data over TLS, applications invoke the `SSL_write()` API, which breaks data into blocks, and generates a MAC for each block before encrypting the block. TLS then puts each encrypted block into the payload field of a TLS record, and then gives the record to the underlying transport layer for transmission. Figure 12.5 depicts the entire flow. We further explain each step in the following.

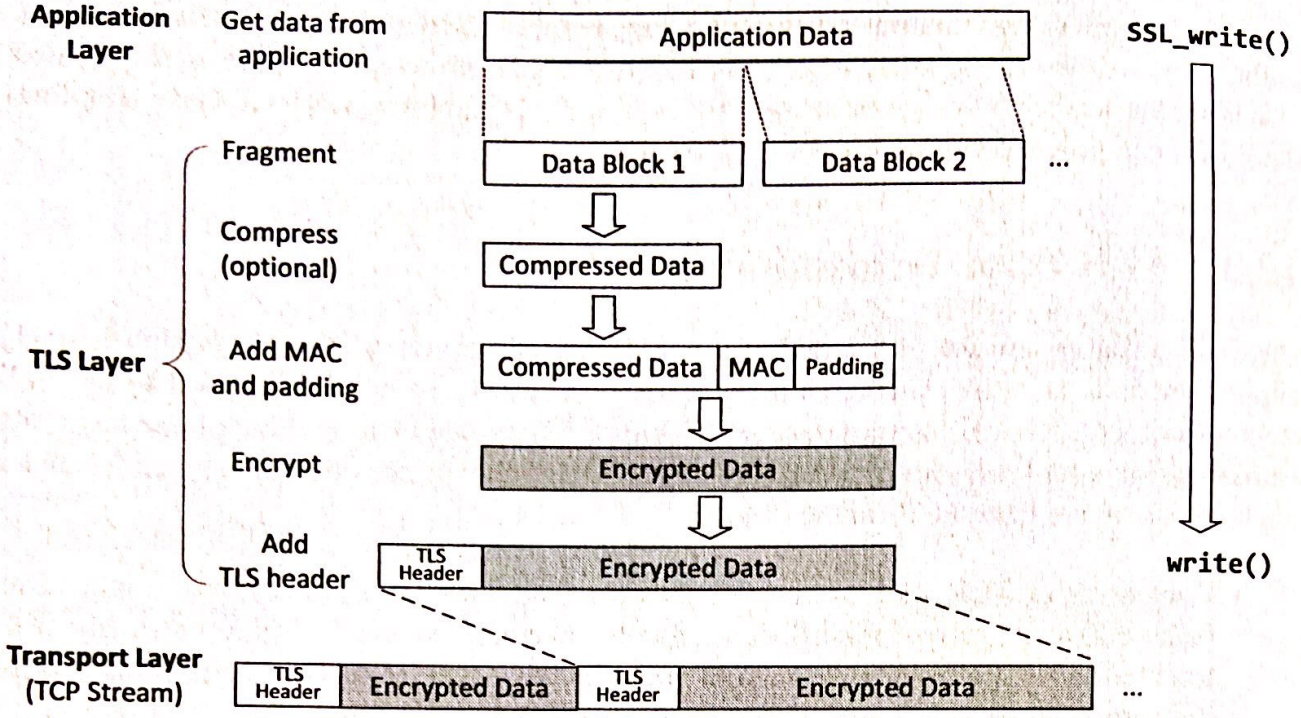

Figure 12.5: Sending data with TLS record protocol

- Fragment data: fragment the data to be sent into blocks of 2^{14} bytes or less.
- Compress data (optionally): compress each block if required. This step is optional; by default, TLS uses the standard method `CompressionMethod.null`, which does not compress data.
- Add MAC: use the MAC key to calculate the MAC of the data. Each record has a sequence number, which is included in the MAC calculation.
- Add padding: for some block ciphers, if the total size of the data plus the MAC is not an integral multiple of the block cipher's block length, padding will be added.
- Encrypt data: encrypt the data, MAC, and padding using the encryption key. For block ciphers, a random IV (Initial Vector) is used and it is put at the beginning of the payload field (IV is not encrypted).
- Add TLS Header: add the TLS header to the payload.

After a TLS record is constructed, TLS writes the record to the TCP stream using APIs such as `write()`. TCP will be responsible for sending out the data. TCP does not observe the boundary of the records; it simply treats the records as part of its data stream.

12.3.2 Receiving Data with TLS Record Protocol

When an application needs to read data from the TLS channel, it calls the TLS API `SSL_read()`, which calls the system call `read()` to read one or multiple records from the TCP stream, decrypt them, verifies their MAC, decompress the data (if needed), before giving the data to the application. Figure 12.6 depicts the entire process.

Remember that as the term "record protocol" implies, TLS Record protocol processes data based on records. Only when a TLS record is completely received, it can be processed. Once a record is taken out of the TCP stream, even if the application's `SSL_read()` request does not consume all the data in the record, the leftover cannot be saved back to the TCP stream; it must be buffered in a different place for the next `SSL_read()` request. TLS provides a buffer to handle this situation, which buffers unused application data.

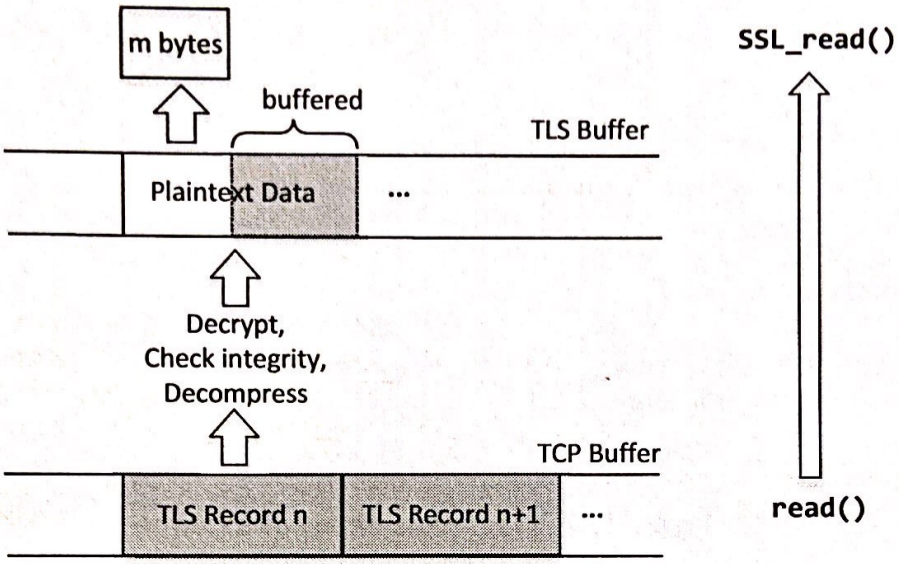

Figure 12.6: Reading data with TLS record protocol

When `SSL_read()` is called and the TLS buffer is empty, one TLS record is retrieved from the TCP buffer and processed. If the number of bytes is not enough to satisfy the request, one more TLS record will be retrieved from the TCP buffer and processed. This will repeat until the request is satisfied or no more data is available in the TCP buffer. If the total number of bytes processed by TLS ends up exceeding what is requested by the application, the leftover will be stored in the TLS buffer for later read requests.

When `SSL_read()` is called and the TLS buffer is not empty, TLS tries to get the requested amount of data from the TLS buffer. If the buffer contains enough data, TLS just delivers the requested amount of data to the application; if the TLS buffer does not have enough data for the request, TLS returns all the data in the buffer to the application. The next `SSL_read()` request will start with an empty TLS buffer, and it follows the same procedure described earlier.

12.4 TLS Programming: A Client Program

Now we have understood how the TLS protocol works. We would like to use the protocol to secure the communication between a client and a server. In this section, we focus on the client side. We implement a client program to communicate with real-world web servers using the TLS protocol.

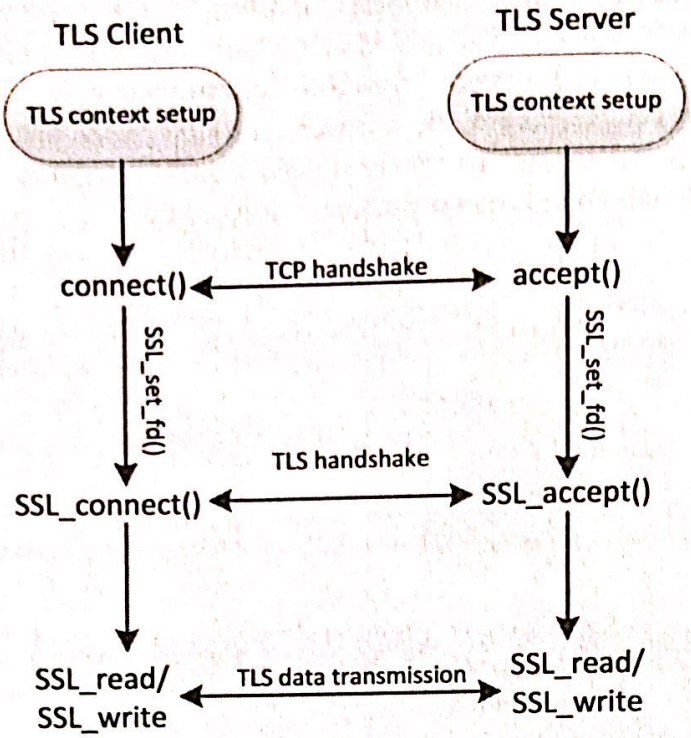

Figure 12.7: TLS programming overview

12.4.1 The Overall Picture

Figure 12.7 gives an overview of the four major steps in TLS programming, including both client and server sides.

1. TLS context setup: This step prepares everything needed in a TLS connection, including loading the cryptographic algorithms, loading private key, decision on TLS version, decision on whether to verify peer's certificate, etc.
2. TCP connection setup: TLS is mostly built on top of TCP, so the client and the server need to establish a TCP connection first.
3. TLS handshake: Once the TCP connection is established, the client and server run the TLS Handshake protocol to establish a TLS session.
4. Data transmission: At this step, the client and the server can send data to each other using the established TLS session.

12.4.2 TLS Initialization

The TLS protocol is a stateful protocol, i.e., after a TLS connection is established, its state will be maintained and used during the lifetime of the connection. We need to initialize data structures for holding the state information. There are three components that need to be initialized: library, SSL context, and SSL session. The code for initialization is listed in the following.

Listing 12.1: TLS initialization (part of `tls_client.c`)

```
SSL* setupTLSClient(const char* hostname)
```

```
{
   // Step 0: OpenSSL library initialization
   // This step is no longer needed since version 1.1.0.
   // The OpenSSL version in our Ubuntu16.04 VM is 1.0.2g,
   //     so this step is still needed.
   SSL_library_init();
   SSL_load_error_strings();

   // Step 1: SSL context initialization
   SSL_METHOD *meth = (SSL_METHOD *)TLSv1_2_method();
   SSL_CTX* ctx = SSL_CTX_new(meth);
   SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
   SSL_CTX_load_verify_locations(ctx, NULL, "./cert");

   // Step 2: Create a new SSL structure for a connection
   SSL* ssl = SSL_new (ctx);

   // Step 3: Enable the hostname check
   X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
   X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);

   return ssl;
}
```

- Step 0: Before calling any other `OpenSSL` APIs, a program must perform an initialization using `SSL_library_init()` and `SSL_load_error_strings()`. The first API registers the available ciphers and digests, while the second API loads error strings, so when an error happens inside the TLS library, more meaningful text messages can be printed out. If memory usage is an issue, the second API does not need to be called [Openssl Wiki, 2017].

  Note: Since version 1.1.0, `OpenSSL` will automatically allocate all resources needed, so there is no need to call these two APIs any more. However, the OpenSSL version in our Ubuntu16.04 VM is 1.0.2g, so this step is still needed.

- Step 1: Create a SSL context data structure, and initialize it. This context will be used as the basis for the SSL data structure. During the TLS handshake, the server needs to send its certificate to the client, which verifies whether the certificate is valid. We use `SSL_CTX_set_verify()` with the `SSL_VERIFY_PEER` option to tell TLS that the certificate verification is needed and that if the verification fails, the handshake protocol should be terminated immediately.

  To verify server certificates, the client needs to have a set of trusted CA certificates. We can store these trusted certificates in a folder and tell TLS about the location using `SSL_CTX_load_verify_locations()`.

- Step 2: Create an SSL data structure, which will be used for making a TLS connection. This data structure inherits the settings from the context `ctx`. Whatever the setting we make to the SSL data structure will not affect the context data structure, and vice versa. In a sense, we can say that the SSL data structure contains an instance of the context.

- Step 3: Enable the hostname check. These two lines are very important, missing the hostname check is one of the common mistakes made by developers. We will provide detailed explanation of this step in §12.5.

12.4.3 TCP Connection Setup

Although TLS can be used with datagram-oriented transport protocols, such as the User Datagram Protocol (UDP), it is primarily used with reliable transport protocols such as the Transmission Control Protocol (TCP). To use TCP, our client code should establish a TCP connection with the server first. This part of code is listed below.

Listing 12.2: Establish a TCP connection with the server (part of `tls_client.c`)

```
int setupTCPClient(const char* hostname, int port)
{
    struct sockaddr_in server_addr;

    // Get the IP address from hostname
    struct hostent* hp = gethostbyname(hostname);

    // Create a TCP socket
    int sockfd= socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Fill in the destination information (IP, port #, and family)
    memset (&server_addr, '\0', sizeof(server_addr));
    memcpy(&(server_addr.sin_addr.s_addr), hp->h_addr, hp->h_length);
    server_addr.sin_port   = htons (port);
    server_addr.sin_family = AF_INET;

    // Connect to the destination
    connect(sockfd, (struct sockaddr*) &server_addr,
            sizeof(server_addr));

    return sockfd;
}
```

The above program is quite standard for socket programming. It first creates a TCP socket, fills in the `server_addr` data structure with the destination information, and the finally invokes `connect()` to perform the TCP three-way handshake protocol with the server.

12.4.4 TLS Handshake

After a TCP connection is established with a server, we can now use this connection to perform the TLS Handshake protocol. First, we use `SSL_set_fd()` to bind the SSL object with the TCP connection, which is represented by the connection's socket file descriptor `sockfd`. We then call `SSL_connect()` to initiate the TLS handshake with the server. If the handshake fails, we can use `ERR_print_errors_fp()` to print out the error message.

Listing 12.3: Start TLS Handshake (part of `tls_client.c`)

```
#define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr);
                                      exit(2); }

SSL* ssl = setupTLSClient(hostname);          // See Listing 12.1
int  sockfd = setupTCPClient(hostname, port); // See Listing 12.2

SSL_set_fd(ssl, sockfd);
int err = SSL_connect(ssl);
CHK_SSL(err);

printf("SSL connection is successful\n");
printf ("SSL connection using %s\n", SSL_get_cipher(ssl));
```

12.4.5 Application Data Transmission

Once the TLS Handshake protocol has succeeded, a TLS session will be established between the client and the server. We can consider this session as consisting of two uni-directional pipes, one from the client to the server, and the other from the server to the client. For each pipe, the sender side can use `SSL_write()` to write its data to the pipe, while the receiver side can use `SSL_read()` to read data from the pipe. Data going through the pipe are protected by the underlying TLS protocol.

Since we will test our client program with a real-world web server, we will send an HTTP request to the web server. In the following code, we construct a simple HTTP GET request using `sprintf()` and a hostname, and then send the request to the server using `SSL_write()`. We then use `SSL_read()` to keep reading the data returned from the server. By default, `SSL_read()` will block if no data is currently available, until data become available or the session is closed.

Listing 12.4: Receive and send data (part of `tls_client.c`)

```
char buf[9000];
char sendBuf[200];

sprintf(sendBuf, "GET / HTTP/1.1\nHost: %s\n\n", hostname);
SSL_write(ssl, sendBuf, strlen(sendBuf));

int len;
do {
    len = SSL_read (ssl, buf, sizeof(buf) - 1);
    buf[len] = '\0';
    printf("%s\n",buf);
} while (len > 0);
```

It should be noted that when we create the TCP connection, i.e., when calling the function `setupTCPClient(hostname, port)`, we use 443 as the port number. Therefore, our HTTP request is sent to the web server's HTTPS service (default port is 443), which basically runs the HTTP protocol over TLS (whereas the HTTP service at the default port 80 runs the HTTP protocol directly over the unprotected TCP).

12.4.6 Set Up the Certificate Folder

Before testing our client program on a real-world web server, we need to gather some trusted CA certificates, and store them in the "./cert" folder specified in the TLS initialization step. For example, if we want to test our client program on https://www.google.com, we need to know what trusted CA certificates can be used to verify the certificates from Google. We can use openssl's built-in HTTPS client to get the information:

```
$ openssl s_client -connect www.google.com:443
...
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google
    LLC/CN=www.google.com
   i:/C=US/O=Google Trust Services/CN=Google Internet Authority G3
 1 s:/C=US/O=Google Trust Services/CN=Google Internet Authority G3
   i:/OU=GlobalSign Root CA - R2/O=GlobalSign/CN=GlobalSign
```

The above result shows that Google's certificate is issued by "Google Trust Services", but this is only an intermediate CA, whose own certificate is issued by a root CA called "GlobalSign Root CA - R2". We need to get this root CA's self-signed certificate in order to verify Google's certificate. This is a well-known root CA, and it has been preloaded by most browsers. We can export the CA's certificate from a browser. The following instructions show how to get a CA certificate from Firefox.

1. Type about:preferences in the URL field, and we will enter the setting page.
2. Select Privacy & Security, and scroll down to the bottom. Click the "View Certificates" button; a pop-up window titled "Certificate Manager" will show up.
3. Select the Authorities Tab, and we can see a list of certificates trusted by Firefox. Select the one that we need, and then click the Export button to save the selected certificate in a file (GlobalSignRootCA-R2.crt) inside the "./cert" folder.

Putting the exported PEM file in the "./cert" folder is not enough. When TLS tries to verify a server certificate, it will generate a hash value from the issuer's identify information, use this hash value as part of the file name, and then use this name to find the issuer's certificate in the "./cert" folder. Therefore, we need to rename each CA's certificate using the hash value generated from its subject field, or we can make a symbolic link out of the hash value. In the following command, we use openssl to generate a hash value, which is then used to create a symbolic link.

```
$ openssl x509 -in GlobalSignRootCA-R2.crt -noout -subject_hash
4a6481c9

$ ln -s GlobalSignRootCA-R2.crt 4a6481c9.0
$ ls -l
total 4
lrwxrwxrwx 1 ... 4a6481c9.0 -> GlobalSignRootCA-R2.crt
-rw-r--r-- 1 ... GlobalSignRootCA-R2.crt
```

Once we set up the CA certificate, we are now ready to test our client program. If we did everything correctly, we should be able to get an HTML page from the target web server.

However, if we have not saved the correct CA certificates in our "./cert" folder, we are likely to see the following error message:

```
67136:error:14090086:SSL routines:ssl3_get_server_certificate:
certificate verify failed:s3_clnt.c:1264:
```

Many reasons can trigger the above error message, such as an expired certificate, an corrupted certificate, etc. However, if we are sure that the certificate is valid, we should investigate whether our program is correct or whether we have obtained the correct CA certificate needed for verifying the server certificate.

12.4.7 The Complete Client Code

For the completeness, we include the entire client code in the following. We omit the code for the `setupTLSClient()` and `setupTCPClient()` functions, because they have already been listed in their entirety in our earlier discussion.

Listing 12.5: The complete TLS client code (`tls_client.c`)

```
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <netdb.h>

#define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr);
                                      exit(2); }

SSL* setupTLSClient(const char* hostname) { ... }
int setupTCPClient(const char* hostname, int port) { ... }

int main(int argc, char *argv[])
{
   char *hostname = "example.com";
   int port = 443;

   if (argc > 1) hostname = argv[1];
   if (argc > 2) port = atoi(argv[2]);

   // TLS initialization and create TCP connection
   SSL *ssl   = setupTLSClient(hostname);
   int sockfd = setupTCPClient(hostname, port);

   // TLS handshake
   SSL_set_fd(ssl, sockfd);
   int err = SSL_connect(ssl); CHK_SSL(err);
   printf("SSL connection is successful\n");
   printf ("SSL connection using %s\n", SSL_get_cipher(ssl));

   // Send and Receive data
   char buf[9000];
   char sendBuf[200];
```

```
    sprintf(sendBuf, "GET / HTTP/1.1\nHost: %s\n\n", hostname);
    SSL_write(ssl, sendBuf, strlen(sendBuf));

    int len;
    do {
      len = SSL_read (ssl, buf, sizeof(buf) - 1);
      buf[len] = '\0';
      printf("%s\n",buf);
    } while (len > 0);
}
```

Compilation. To compile the above code (`tls_client.c`), we use the following `gcc` command. The `-lssl` and `-lcrypto` options specify that `openssl`'s `ssl` and `crypto` libraris are needed.

```
$ gcc -o tls_client tls_client.c -lssl -lcrypto
```

If we do have `SSL_library_init()` and `SSL_load_error_strings()` in our code, when we compile the code against the `OpenSSL` 1.1.0 library, we will see the following error message. This is because the two APIs have been removed from the library since version 1.1.0.

```
$ gcc -o tls_client tls_client.c -lssl -lcrypto
/tmp/ccqgXBF2.o: In function 'setupTLSClient':
tls_client.c:(.text+0x7): undefined reference to 'SSL_library_init'
tls_client.c:(.text+0xc): undefined reference to
    'SSL_load_error_strings'
```

12.5 Verifying Server's Hostname

Not checking server's hostname is a very common security flaw in programs that are based on TLS [Georgiev et al., 2012]. In this section, we use experiments to demonstrate why failing to do so can cause security problems.

12.5.1 Modified Client Code

We slightly modify our client program, so we can print out more information during the runtime. The information provides us with insights on the internal logic of TLS. We have changed the TLS setup code, and the new `setupTLSClient()` function is shown in the following:

Listing 12.6: Modified TLS setup code (part of `tls_client_with_callback.c`)

```
SSL* setupTLSClient(const char* hostname)
{
    // Step 0: OpenSSL library initialization
    // This step is no longer needed as of version 1.1.0.
    SSL_library_init();
    SSL_load_error_strings();

    // Step 1: SSL context initialization
```

```
    SSL_METHOD *meth = (SSL_METHOD *)TLSv1_2_method();
    SSL_CTX* ctx = SSL_CTX_new(meth);
    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback); ❶
    SSL_CTX_load_verify_locations(ctx, NULL, "./cert");

    // Step 2: Create a new SSL structure for a connection
    SSL* ssl = SSL_new (ctx);

    // Step 3: Enable the hostname check
    X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);  ❷
    X509_VERIFY_PARAM_set1_host(vpm, hostname, 0); ❸

    return ssl;
}
```

In Line ❶, we specify a callback function when invoking `SSL_CTX_set_verify()`. The callback function will be invoked every time a certificate is verified, so the application code can get a chance to respond to the verification result, especially when the verification fails. In the original version of our code, we do not pass a callback function. The implementation of our callback function is shown in the following.

Listing 12.7: The callback function (part of `tls_client_with_callback.c`)

```
int verify_callback(int preverify_ok, X509_STORE_CTX *x509_ctx)
{
    char  buf[300];

    X509* cert = X509_STORE_CTX_get_current_cert(x509_ctx); ❹
    X509_NAME_oneline(X509_get_subject_name(cert), buf, 300);
    printf("subject= %s\n", buf);      ❺

    if (preverify_ok == 1) {
       printf("Verification passed.\n");
    } else {
       int err = X509_STORE_CTX_get_error(x509_ctx);
       printf("Verification failed: %s.\n",
          X509_verify_cert_error_string(err)); ❻
    }

    /* For the experiment purpose, we always return 1, regardless of
     * whether the verification is successful or not. This way, the
     * TLS handshake protocol will continue. This is not safe!
     * Readers should not blindly copy the following line. We will
     * discuss the correct return value later.
     */
    return 1;
}
```

to the invocation of the callback function. We print out the subject field of the certificate in question (Line ❺) and if the verification is not a success, we print out the reason (Line ❻).

12.5.2 An Experiment: Man-In-The-Middle Attack

With the modified client program, we can conduct an experiment to see why it is important to check the hostname. Without checking whether the server's hostname matches with the subject information in its certificate, the client program will be susceptible to Man-In-The-Middle (MITM) attacks. In our experiment, we emulate such an attack on a victim who wants to visit `www.facebook.com`. We use `www.example.org` as the malicious server, which tries to steal the victim's Facebook credentials via an MITM attack.

The first step for such an attack is to get a victim to come to `www.example.org` every time they visit Facebook. A typical way to achieve that is to use DNS cache poisoning attack, which is discussed in Chapter 4. Using this attack, attackers can poison the victim's DNS cache, so when the victim's machine tries to find out the IP address of `www.facebook.com`, it gets the IP address of `www.example.org`. Therefore, the victim thinks that he/she is visiting `www.facebook.com`, but in reality, he/she is visiting `www.example.org`.

We will not launch a real DNS cache poisoning attack; instead, we manually add an entry to the `/etc/hosts` file, so the hostname `www.facebook.com` is always resolved to `93.184.216.34`, which is the IP address of `www.example.org` (it should be noted that this IP address is what we obtained during the writing of this book, and it may change; to repeat this experiment, readers should use the `dig` command to get the IP address). The following entry is added to `/etc/hosts`.

```
93.184.216.34    www.facebook.com
```

We basically tell our computer that `93.184.216.34` is the (spoofed) IP address of `www.facebook.com`. Now we visit Facebook using our modified client program. First, we comment out Lines ❷ and ❸ from the `setupTLSClient()` function in Listing 12.6, so we do not do hostname checks. We run our client program and get the following result:

```
$ tls_client www.facebook.com 443
subject= ... /CN=DigiCert High Assurance EV Root CA
Verification passed.
subject= ... /CN=DigiCert SHA2 High Assurance Server CA
Verification passed.
subject= ... /CN=www.example.org
Verification passed.
SSL connection is successful
SSL connection using ECDHE-RSA-AES128-GCM-SHA256
```

The first six lines are printed out from the `verify_callback()` function shown in Listing 12.7. Because of the setting in Line ❶ of the `setupTLSClient()` function, every time a certificate is verified by TLS, the callback function is invoked. In total, three verification has been conducted, one for each certificate on the certificate chain provided by the server, starting from the root certificate (this one is not provided by the server; instead, it is loaded from our "`./cert`" folder).

From the result, we can tell that the verifications carried out by TLS are all successful, and eventually our client was able to connect to `www.example.org`'s HTTPS sever, even though what we really want to connect to is `www.facebook.com`. Now imagine that

Listing 12.8: Firefox's Warning message after failing to verify the server's identity

```
This Connection is Untrusted

The owner of www.facebook.com has configured their website
improperly.  To protect your information from being stolen,
Firefox has not connected to this website.

This site uses HTTP Strict Transport Security (HSTS) to specify
that Firefox may only connect to it securely. As a result, it is
not possible to add an exception for this certificate.

www.facebook.com uses an invalid security certificate.

The certificate is only valid for the following names:
www.example.org, example.com, example.edu, example.net,
example.org, www.example.com, www.example.edu, www.example.net.

Error code: SSL_ERROR_BAD_CERT_DOMAIN
```

`www.example.org` is evil: it returns a login page that looks exactly like Facebook's login page. Since the URL is going to display `www.facebook.com`, there is no easy way for victims to know that the login page is fake. If they type in their credentials, their security will be compromised.

The above experiment shows that the MITM attack is successful. This is counter-intuitive, because the public key Infrastructure and TLS protocol are designed to defeat such type of attack. What is wrong here? Before answering the question, let us use a real client program, such as Firefox browser, to visit Facebook (the fake IP is still in effect). Listing 12.8 shows the warning message from Firefox (other browsers have different but similar warnings).

Clearly, browsers can detect our MITM attack by warning the users that the so-called Facebook site does not provide a certificate valid for `www.facebook.com`. The certificate is valid though, but it is only valid for a list of related names, and Facebook is not on that list.

12.5.3 Hostname Checking

The reason why browsers can detect our MITM attack is that browsers conduct an extra check to ensure that the subject name on the certificate from a server matches with the user's intention, which is the hostname displayed in a browser's URL field. The TLS library does not know what the user's intention is, because it is application dependent. It is the responsibility of the application to conduct the checking. That is exactly what was missing from our code, and it is also what is missing from many TLS-based applications.

Before `OpenSSL` 1.0.2, applications need to conduct the hostname checking manually. Namely, they need to extract the common name entry from the subject field of the server's certificate, compare it with the hostname provided by the user, and check whether these two names match or not. Some certificates may contain a list of alternative names stored in their extension field. For example, from the Firefox warning message shown above, the certificate from `www.example.org` is actually valid for several names, in addition to `www.example.org` that is specified in the common name field, including `example.com`, `www.example.`

com, example.edu, etc. Therefore, conducting the hostname checking is quite complicated and tedious.

Since 1.0.2, OpenSSL automates the aforementioned hostname checking, if an application tells it to do so and also tells it what hostname should be checked against. Keep in mind that TLS does not know what hostname is the user's intention, so it must be told by the application. In our previous MITM experiment, we intentionally commented out two lines of code in Listing 12.6 (Lines ❷ and ❸), which are also shown in the following. Let us uncomment them.

```
// Enable the hostname check
X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);
```

Let us repeat the experiment by visiting www.facebook.com using our client program. The following messages are printed out.

```
$ client www.facebook.com 443
subject= /C=US/ST=California/L=Los Angeles/O=Internet Corporation for
Assigned Names and
Numbers/OU=Technology/CN=www.example.org
Verification failed:  Hostname mismatch.  ❶
subject= ... /CN=DigiCert High Assurance EV Root CA
Verification passed.
subject= ... /CN=DigiCert SHA2 High Assurance Server CA
Verification passed.
subject= ... /CN=www.example.org
Verification passed.                                    ❷
SSL connection is successful                            ❸
SSL connection using ECDHE-RSA-AES128-GCM-SHA256
```

From the execution result, we can tell that TLS conducts two types of verification. The first verification is the hostname verification. Since none of the acceptable names in www.example.org's certificate matches with the intended hostname www.facebook.com, the verification fails (Line ❶). The second verification is the certificate verification, i.e., verifying whether the certificates on the certificate chain is valid or not. From the results, we can see that all the certificates have passed the verification and that verifying hostname and verifying certificates are separate. Many developers mistakenly think that the certificate verification automatically covers the hostname verification. That is not true.

Surprisingly, the connection with the server is still successful (Line ❸). This is because in the callback function, when the verification fails, we did not abort the TLS connection. Therefore, it is important to know that if an application uses a callback function to handle failed verifications, it must make sure to handle the situation appropriately. Most browsers handle it by warning users about the danger, but still let users decide whether to proceed or not. Unless a developer intends to use a similar strategy, the safest way is to abort the TLS connection and/or exit from the program. Actually, if we do not specify a callback function, TLS will use its default callback function, which aborts the TLS connection if the hostname check or other checks fail.

Whether to proceed or not depends on the return value of the callback function. If the return value is 1, the TLS handshake protocol will continue; if the return value is 0, the handshake will immediately be terminated. For the sake of security, we decide to terminate TLS if any verification check fails. The modified callback function is provided in the following:

Listing 12.9: Adding return value to the callback function

```
int verify_callback(int preverify_ok, X509_STORE_CTX *x509_ctx)
{
   char  buf[300];

   X509* cert = X509_STORE_CTX_get_current_cert(x509_ctx);
   X509_NAME_oneline(X509_get_subject_name(cert), buf, 300);
   printf("subject= %s\n", buf);

   if (preverify_ok == 1) {
     printf("Verification passed.\n");
     return 1;  // Continue the TLS handshake protocol
   } else {
     int err = X509_STORE_CTX_get_error(x509_ctx);
     printf("Verification failed: %s.\n",
     X509_verify_cert_error_string(err));
     return 0;  // Stop the TLS handshake protocol
   }
}
```

We repeat the experiment and revisit `www.facebook.com`. From the following result, we can see that our program exits after the failed hostname check.

```
$ tls_client www.facebook.com 443
subject= ... /CN=www.example.org
Verification failed: Hostname mismatch.
3071059648:error:14090086:SSL routines:ssl3_get_server_certificate:
certificate verify failed:s3_clnt.c:1264:
```

12.6 TLS Programming: the Server Side

In this section, we write a simple TLS server. Instead of writing one that only works with our client program, we want to make the server more interesting: we implement a very simple HTTPS server, which can be tested using browsers. The objective of our server is that upon receiving anything from a client, the server returns a web page containing "Hello World". HTTPS itself is not a new protocol; it is the HTTP protocol running on top of the TLS protocol. Therefore, an HTTPS server consists of two steps: (1) establishing a TLS connection with the client, and (2) receiving HTTP requests and sending HTTP responses via the established TLS connection.

12.6.1 TLS Setup

The name of our HTTPS server will be `bank32.com`. We have already created a public-key certificate for this server in Chapter 11, and we will use it here. The TLS setup on the server side is quite different from that on the client. We show our code in the following.

Listing 12.10: TLS initialization on the server side (part of `tls_server.c`)

```
   const SSL_METHOD *meth;
   SSL_CTX* ctx;
```

```
SSL* ssl;

// Step 0: OpenSSL library initialization
SSL_library_init();
SSL_load_error_strings();

// Step 1: SSL context initialization
meth = (SSL_METHOD *)TLSv1_2_method();
ctx = SSL_CTX_new(meth);
SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, NULL);                    ❶

// Step 2: Set up the server certificate and private key
SSL_CTX_use_certificate_file(ctx, "./cert_server/bank32_cert.pem",
                             SSL_FILETYPE_PEM);                    ❷
/* SSL_CTX_use_certificate_chain_file(ctx, ...); */               ❸
SSL_CTX_use_PrivateKey_file(ctx, "./cert_server/bank32_key.pem",
                             SSL_FILETYPE_PEM);                    ❹

// Step 3: Create a new SSL structure for a connection
ssl = SSL_new (ctx);
```

- Step 1 creates an SSL context. The `SS_VERIFY_NONE` in Line ❶ indicates that the server will not ask the client to send a certificate. This is a typical behavior of server, because clients are usually operated by end users, most of which do not have a certificate. For the TLS protocol to work, only one side needs to send its public key; that is the job of the server and the reason why we use `SSL_VERIFY_PEER` in the client program.

- Step 2 consists of two important steps: (1) load the server's certificate, and (2) load the server's private key. If the server's certificate is signed by a trusted root CA, we can use the `SSL_CTX_use_certificate_file()` to load the certificate file (Line ❷). However, in most cases, a server's certificate is signed by an intermediate CA, the certificate of which may be signed by another intermediate CA. It is the server's obligation to provide all the certificates on this certificate chain (not including the last one, which is the root CA certificate). The server needs to save these required certificates in a single file, which must be in the PEM format. The certificates must be sorted following the order in the certificate chain, starting with the server's certificate. We then use `SSL_CTX_use_certificate_chain_file()` to load the certificate chain file (Line ❸, which is commented out).

  The server also needs to know the private key corresponding to its certificate. Recall that in the TLS protocol, the client will send a secret to the server, encrypted using the server's pubic key. To get the secret, the server needs to use the corresponding private key. The `SSL_CTX_use_PrivateKey_file()` API is used to load the private key (Line ❹). If the private key is password protected, users running the server program will be asked to provide the password.

- Step 3 creates an SSL data structure from the SSL context data structure. This step is similar to that in the client code.

12.6.2 TCP Setup

As we have mentioned before, a typical TLS program runs on top of TCP, so before we run the TLS protocol, a TCP connection needs to be established first. This part of the program is listed below.

Listing 12.11: Initialize TCP server (part of `tls_server.c`)

```
int setupTCPServer()
{
    struct sockaddr_in sa_server;
    int listen_sock, err, tr = 1;

    // Create a listening socket
    listen_sock= socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    err = setsockopt(listen_sock, SOL_SOCKET, SO_REUSEADDR,
                     &tr, sizeof(int));
    CHK_ERR(err, "setsockopt");

    // Prepare for address structure
    memset (&sa_server, '\0', sizeof(sa_server));
    sa_server.sin_family      = AF_INET;
    sa_server.sin_addr.s_addr = INADDR_ANY;
    sa_server.sin_port        = htons (4433);

    // Bind the socket to a port
    err = bind(listen_sock, (struct sockaddr*)&sa_server,
                            sizeof(sa_server));
    CHK_ERR(err, "bind");

    // Listen to connections
    err = listen(listen_sock, 5);
    CHK_ERR(err, "listen");
    return listen_sock;
}
```

The above program is quite standard for socket programming. It creates a TCP socket, binds it to a TCP port (4433), and marks the socket as a passive socket (via the `listen()` call), which means that the socket will be used to accept incoming connection requests.

12.6.3 TLS Handshake

Once the TCP is set up, the server program enters the waiting state via the `accept()` system call, waiting for incoming connection requests. By default, `accept()` will block. When a TCP connection request comes from a client, TCP will finish the three-way handshake protocol with the client. Once the connection is established, TCP unblocks `accept()`, which returns a new socket to the server program. This new socket will be given to the SSL layer via `SSL_set_fd()`. At this point, the server calls `SSL_accept()` to wait for the client to initiate the TLS Handshake protocol.

Listing 12.12: The main function of the TLS server code (part of `tls_server.c`)

```
#include <unistd.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <netdb.h>

#define CHK_SSL(err) if ((err) < 1) { ERR_print_errors_fp(stderr);
                                      exit(2); }
#define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }

int  setupTCPServer();                    // Defined in Listing 12.11
void processRequest(SSL* ssl, int sock); // Defined in Listing 12.13

int main()
{
  const SSL_METHOD *meth;
  SSL_CTX* ctx;
  SSL* ssl;

  // Step 0: OpenSSL library initialization
  SSL_library_init();
  SSL_load_error_strings();

  // Step 1: SSL context initialization
  meth = (SSL_METHOD *)TLSv1_2_method();
  ctx = SSL_CTX_new(meth);
  SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, NULL);

  // Step 2: Set up the server certificate and private key
  SSL_CTX_use_certificate_file(ctx, "./cert_server/bank32_cert.pem",
                               SSL_FILETYPE_PEM);
  /* SSL_CTX_use_certificate_chain_file(ctx, ...); */
  SSL_CTX_use_PrivateKey_file(ctx, "./cert_server/bank32_key.pem",
                              SSL_FILETYPE_PEM);

  // Step 3: Create a new SSL structure for a connection
  ssl = SSL_new (ctx);

  struct sockaddr_in sa_client;
  size_t client_len;

  int listen_sock = setupTCPServer();
  while (1) {
    int sock = accept(listen_sock, (struct sockaddr*)&sa_client,
                      &client_len);
    printf ("TCP connection established!\n");
    if (fork() == 0) { // Child process
      close (listen_sock);

      SSL_set_fd (ssl, sock);
      int err = SSL_accept (ssl);
```

```
        CHK_SSL(err);
        printf ("SSL connection established!\n");

        processRequest(ssl, sock);
        close(sock);
        return 0;
    } else { // Parent process
        close(sock);
    }
  }
}
```

12.6.4 TLS Data Transmission

Once a TLS connection is established, there is no difference between the client and the server, as both ends can send data to and receive data from the other end. The logic for sending and receiving data are the same as that in the client program. In our server program, we simply send an HTTP reply message back to the client. To do that, we construct the reply message, and then use `SSL_write()` to send the message to the client, over the TLS connection.

Listing 12.13: Send HTTP response (part of `tls_server.c`)

```
void processRequest(SSL* ssl, int sock)
{
    char buf[1024];
    int len = SSL_read (ssl, buf, sizeof(buf) - 1);
    buf[len] = '\0';
    printf("Received: %s\n",buf);

    // Construct and send the HTML page
    char *html =
      "HTTP/1.1 200 OK\r\n"                                    ①
      "Content-Type: text/html\r\n\r\n"
      "<!DOCTYPE html><html>"
      "<head><title>Hello World</title>"
      "<style>body {background-color: black}"
      "h1 {font-size:3cm; text-align: center; color: white;"
      "text-shadow: 0 0 3mm yellow}</style></head>"
      "<body><h1>Hello, world!</h1></body></html>";   ②
    SSL_write(ssl, html, strlen(html));
    SSL_shutdown(ssl);  SSL_free(ssl);
}
```

12.6.5 Testing

Before running the TLS server program, we need to place `bank32.com`'s public-key certificate (`bank32_cert.pem`) and private key (`bank32_key.pem`) inside the `./cert_server` folder. We also need to map the hostname `bank32.com` to the server's IP address. Once everything is done, we can run the following programs.

```
// On the server side:
$ gcc -o tls_server tls_server.c -lssl -lcrypto
$ sudo ./tls_server

// On the client side: use "openssl s_client"
$ openssl s_client -CAfile modelCA_cert.pem -connect bank32.com:4433
CONNECTED(00000003)
depth=1 ..., O = Model CA, CN = modelCA.com, ...
verify return:1
depth=0 ..., O = Bank32 Inc., CN = bank32.com, ...
verify return:1
...

// On the client side: use our own tls_client program
$ tls_client bank32.com 4433
subject= .../O=Model CA/CN=modelCA.com/...
Verification passed.
subject= .../O=Bank32 Inc./CN=bank32.com/...
Verification passed.
SSL connection is successful
SSL connection using AES256-GCM-SHA384
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html><html><head><title>Hello World</title>
<style>body {background-color: black}h1 {font-size:3cm;
text-align: center; color: white;text-shadow: 0 0 3mm
yellow}</style>
</head><body><h1>Hello, world!</h1></body></html>
```

It should be noted that normally we do not need to use the root privilege to run `tls_server`. However, in our provided `Ubuntu16.04` VM, without using the root privilege, the server will not be able to establish a TLS session with clients. We have not figured out the reason yet; it may be caused by some configuration issue. We have tried to run the same program on a physical machine running Ubuntu 16.04, and no root privilege is needed.

We can also use a browser to access our TLS server. Just do not forget to load our root CA's certificate `modelCA_cert.pem` into the browser. If everything works fine, we should be able to get a web page that displays `"Hello World"`.

12.7 Summary

Transport Layer Security provides a secure channel for applications to transmit data. To use TLS, an application first invokes the TLS Handshake protocol to establish a secure channel with its peer, and then sends data using this channel. The encryption, decryption, and detecting tampering are handled by TLS, transparent to applications, making it quite simple to develop applications that can communicate securely. Applications using TLS can communicate with one another, because they "speak" the same protocol. To demonstrate that, we wrote a simple TLS client that can talk to real-world HTTPS web servers; we also wrote a simple TLS server that can communicate with HTTPS-speaking browsers.