Chapter 36

Transport Layer Security

Nowadays more and more data transmissions are done through the Internet. However, when data are transmitted over such a public network unprotected, they can be read or even modified by others. Applications worrying about the security of their communication need to encrypt their data and detect tampering. Cryptographic solutions can be used to achieve this goal. There are many cryptographic algorithms, and even for the same algorithm, there are many parameters that can be used. To achieve interoperability, i.e., allowing different applications to communicate with one another, these applications need to follow a common standard. TLS, Transport Layer Security, is such a standard.

In this chapter, we first discuss how Transport Layer Security works. We specifically focus on the two most important aspects of TLS: handshake and data transmission. We show how two communicating peers can establish a secure channel between themselves, including deciding on the cryptographic algorithms to use, verifying certification, finding common session keys, etc. Once the channel is established, we show how data are transmitted via the channel, and what TLS does to the data inside the secure channel.

We would also like readers to know how to use TLS in their programs, so in the second part of this chapter, we implement two programs, a simple HTTPS client that can get web pages from real-world web servers, and an HTTPS server that can provide pages to browsers. We also show some common mistakes made by developers when using TLS. To better understand the content in this chapter, we suggest that readers study the public-key cryptography and public-key infrastructure first, because TLS depends on them.

Contents

36.1 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 36.2 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). See Figure 36.1.

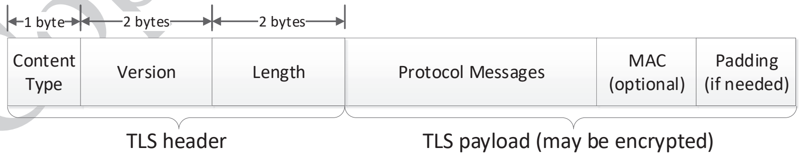

Figure 36.1: TLS record

The protocol messages field carries the actual message, the type of which is specified in the type field of the header. There are five message protocols in TLS, including the Handshake, Alert, Change Cipher Spec, Heartbeat, and Application Protocols. See Figure 36.2.

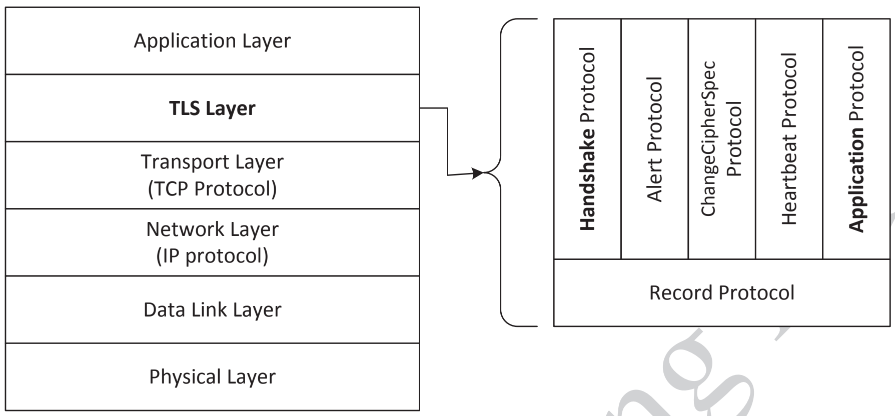

Figure 36.2: TCP/IP network stack with the TLS layer

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 (including key agreement), while the Application protocol is used for actual data transmission using the channel. We will only focus on these two protocols in this chapter.

36.2 TLS Handshake (Version 1.2)

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].

36.2.1 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 36.3 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 the key generation.

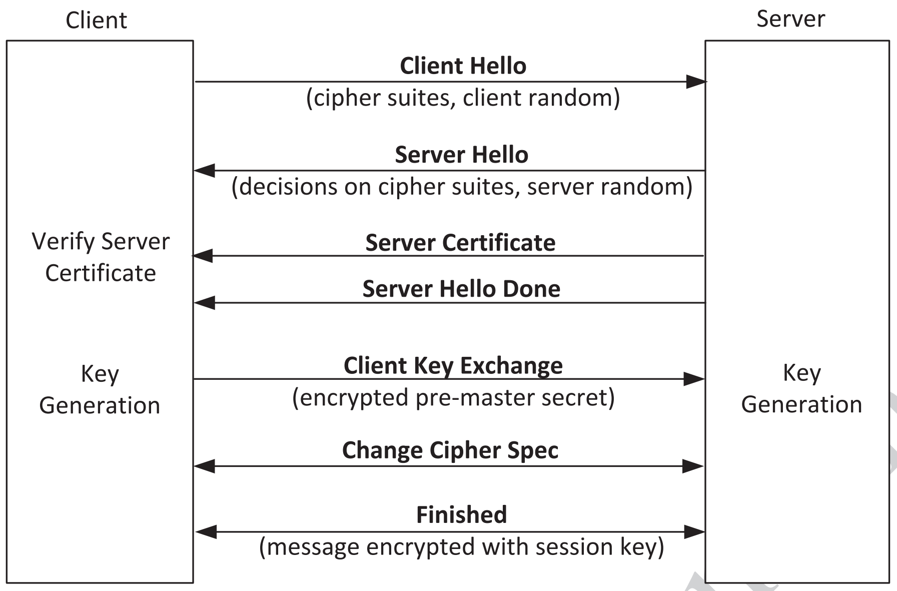
Client
Server
Client Hello
(cipher suites, client random)
Server Hello
(decisions on cipher suites, server random)
Verify Server Certificate
Server Certificate
Server Hello Done
Key Generation
Client Key Exchange
(encrypted pre-master secret)
Key Generation
Change Cipher Spec
Finished
(message encrypted with session key)

Figure 36.3: TLS handshake protocol

- 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 the 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 will provide the details of the verification in §36.2.2.

- 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 §36.2.3.

- 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 36.1 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 36.1: TLS traffic captured by Wireshark

36.2.2 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 learned from the 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 §36.6.

36.2.3 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 36.4 illustrates these three steps.

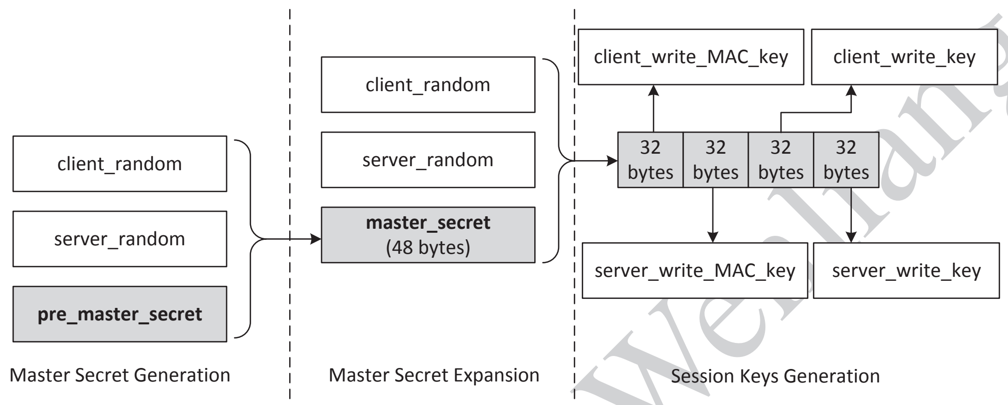

Figure 36.4: TLS key generation (master Secret and session keys)

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 36.4 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 36.4. 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.

36.3 TLS Handshake (Version 1.3)

TLS 1.3 was defined in RFC 8446 [Rescorla, 2018] in August 2018. It is based on the earlier TLS 1.2 specification, but it made many improvements over 1.2, especially in the Handshake protocol. We summarize some of the major differences in the following.

- The static RSA and static Diffie-Hellman key exchanges are no longer supported, because they do not provide perfect forward secrecy. In these two algorithms, the private key is static and will not change for different sessions. Therefore, if the private key is compromised in the future, all the session keys in the past will be compromised (if the attacker has recorded all the past sessions). In cryptography, this means they do not satisfy the *perfect forward secrecy* requirement. In TLS 1.3, this requirement is mandatory.
- TLS 1.3 uses the Ephemeral Diffie-Hellman for key exchange, which uses a new temporary Diffie-Hellman private key for every session. At the end of the session, the key is discarded. Therefore, it has perfect forward secrecy.
- The performance of the Handshake protocol is improved. In TLS 1.2, the client sends the Client Hello message to the server, waits for the server to select the key exchange protocol (part of the cipher suite) from the proposed list, and then starts the key exchange with the server. In TLS 1.3, the client makes a guess of what cipher suites that the server might select, and starts the key exchange protocol based on the guesses. Therefore, the key exchange starts in the Client Hello message. If the guess is wrong, the server will inform the client about its choice, and the client can redo the key exchange. If the guess is correct, one round of communication is saved.
- Because of the improvement in the key exchange, after one round of communication, both client and server will be able to agree upon the session key, so all the communication after the Server Hello step can be encrypted using the session key. In TLS 1.2, two rounds of communication are needed before the encryption can start.

Authentication. In TLS 1.2, when the RSA algorithm is used in the key agreement, the server certificate simultaneously serves two goals: (1) providing the server's public key to the client for the key exchange, (2) providing the certified server name to the client for authentication. This is the benefit of using a static public key algorithm. Since the public/private keys are static, one can obtain a certificate for the public key. If we use an ephemeral algorithm, we can no longer do this, because the public/private keys are for one-time use, and it is impractical to get a public key certificate for a one-time key.

Therefore, in TLS 1.3, the key agreement and authentication algorithms are separated. The Ephemeral Diffie-Hellman protocol is used only for the key agreement, while the authentication is still based on the algorithms like RSA and public-key certificates. It should be emphasized that without the authentication part, the Ephemeral Diffie-Hellman protocol is subject to the man-in-the-middle attack. The Public Key Infrastructure (PKI) and certificates are created to defeat such an attack.

During the authentication, in addition to checking the server name, the server also needs to put a signature on all the handshake messages received from the client, and sends the signature to the client. The client will use the public key from the certificate to verify the signature, checking whether the messages received by the server are the same as what the client has sent out. This verification is essential, because the man-in-the-middle attack requires making changes to the key exchange messages. The signature will help detect any change made by the attacker.

36.4 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 2022, 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 36.5. 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.

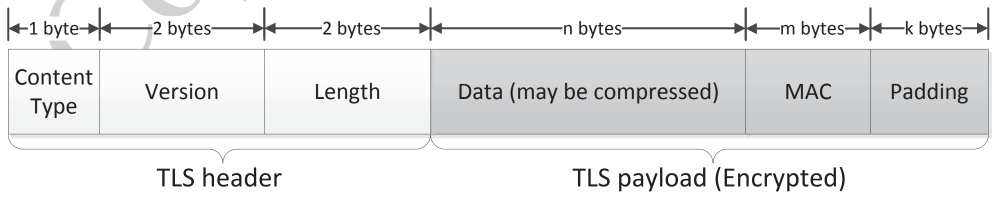

Figure 36.5: Record format of the TLS Record Protocol

36.4.1 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 36.6 depicts the entire flow. We further explain each step in the following.

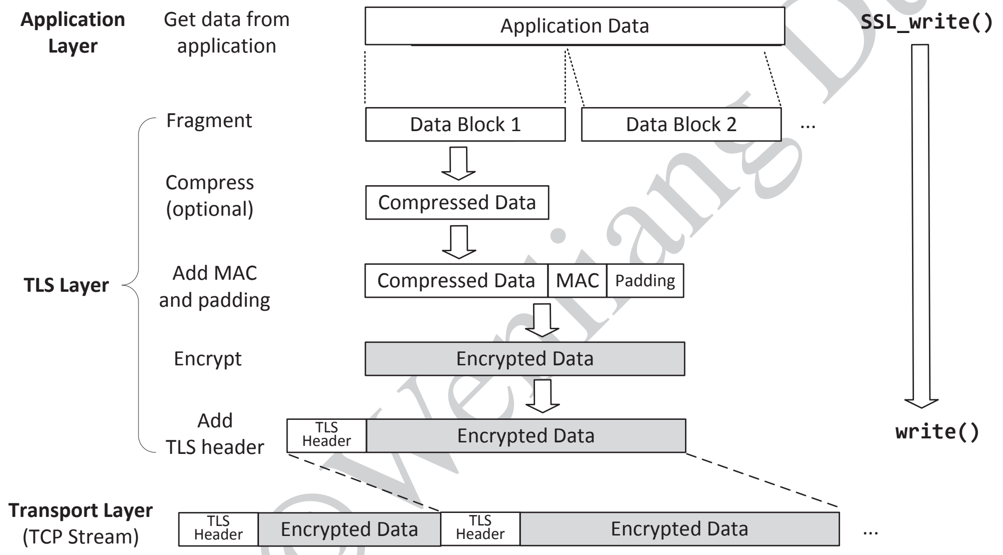

Figure 36.6: Sending data with TLS record protocol

- 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.

36.4.2 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 36.7 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.

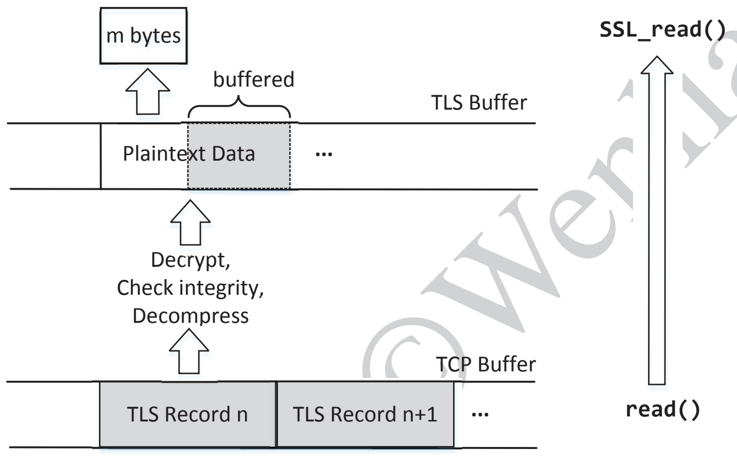

Figure 36.7: Reading data with TLS record protocol

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.

36.5 TLS Client Program in Python

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.

36.5.1 TLS Setup and Handshake

We first give a TLS client program in the following.

Listing 36.1: `tls_client.py`

```
#!/usr/bin/env python3

import socket, ssl, sys, pprint

hostname = sys.argv[1]  # Get the server’s hostname

# Set up the TLS context
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations(capath='/etc/ssl/certs') ①
context.verify_mode    = ssl.CERT_REQUIRED             ②
context.check_hostname = True                          ③

# Create TCP connection
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((hostname, 443))

# Bind the TLS to thE TCP and start TLS handshake
ssock = context.wrap_socket(sock, server_hostname=hostname,
                            do_handshake_on_connect=False)
ssock.do_handshake()   # Start the handshake

# Print out TLS information (cipher, hostname, and certificate)
print("=== Cipher used: {}".format(ssock.cipher()))
print("=== Server certificate:")
pprint.pprint(ssock.getpeercert())

... (data transmission code is omitted) ...

# Close the TLS Connection
ssock.shutdown(socket.SHUT_RDWR)
ssock.close()
```

The TLS program above consists of four major steps. The are depicted in Figure 36.8. We summarize these steps and provide the TLS client code in the following. More details will be discussed throughout this section.

1. TLS context setup: This step prepares everything needed in a TLS connection, including loading the cryptographic algorithms, specifying where the trusted CA certificates are stored (Line ①), deciding on whether to verify peer’s certificate (Line ②) and whether to check the server’s hostname (Line ③), 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. The default port for HTTPS server is `443`, which is used in the code.

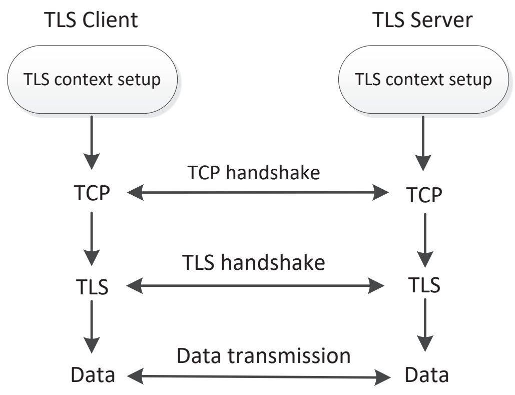

Figure 36.8: TLS programming overview

3. TLS handshake: Once the TCP connection is established, the client and server run the TLS Handshake protocol to establish a TLS session. After a session is established, we can print out information about the session, including the cipher used, the certificates, etc.

4. Data transmission: At this step, the client and the server can send data to each other using the established TLS session. We have omitted this part in the code, as we will discuss them later.

Testing. We can use this client program to communicate with an HTTPS web server in the real world. We should be able to establish a TLS connection with it. In the follow example, we have successfully connected to `www.example.com`.

```
$ client.py www.example.com
=== Cipher used: ('TLS_AES_256_GCM_SHA384', 'TLSv1.3', 256)
=== Server hostname: www.example.com
=== Server certificate:
{'OCSP': ('http://ocsp.digicert.com',),
  ...
```

36.5.2 The Trusted Certificates

In order for a TLS client to verify the server's certificate, the client needs to know where to find the trusted CA certificates. In Line ① of the TLS client program, we have specified the location as `/etc/ssl/certs`, which stores the CA certificates trusted by the operating system. In this folder, we will find many certificate files (`pem` files).

TLS does not use the file name to find the needed certificates, because file name is not part of a certificate. Instead TLS uses the subject's hash value. When TLS tries to verify a server certificate, it gets the issuer's subject information from the certificate, generates a hash value from the information, and then uses this hash to find the issuer's certificate in the trusted certificate folder.

Therefore, for each certificate file, there is typically a symbolic link connecting a certificate subject's hash to the actual certificate. We can use `openssl` to generate the hash value. The following example shows the symbolic link for the `GlobalSign_Root_CA.pem` certificate and how the subject hash can be generated from a certificate.

```
lrwxrwxrwx 1 root root ...  5ad8a5d6.0 -> GlobalSign_Root_CA.pem
lrwxrwxrwx 1 root root ...  GlobalSign_Root_CA.pem

$ openssl x509 -in GlobalSign_Root_CA.pem -noout -subject_hash
5ad8a5d6
```

Using a different certificate folder. If we want to use a customized folder, instead of the one provided by the system, we can specify a different folder in the code, such as `"./cert"`, and then put the trusted certificate in this folder. 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, ... O = Google LLC, CN = www.google.com
   i:C = US, O = Google Trust Services, CN = GTS CA 1O1
 1 s:C = US, O = Google Trust Services, CN = GTS CA 1O1
   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 is already in the `/etc/ssl/certs` folder. We can copy the `pem` file into our folder, and create a symbolic link from the subject's hash.

```
$ ls -l /etc/ssl/certs | grep GlobalSign_Root_CA_-_R2.pem
lrwxrwxrwx 1 root root ... 4a6481c9.0 -> GlobalSign_Root_CA_-_R2.pem
lrwxrwxrwx 1 root root ... GlobalSign_Root_CA_-_R2.pem

// Inside the ./certs folder
$ cp /etc/ssl/certs/GlobalSign_Root_CA_-_R2.pem .
$ ln -s GlobalSign_Root_CA_-_R2.pem 4a6481c9.0
```

This certificate is also preloaded by most browsers, so we can can export it 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 inside the `"./cert"` folder.

36.5.3 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 `ssock.sendall()` to write its data to the pipe, while the receiver side can use `ssock.recv()` 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, and then send the request to the server. We then use `ssock.recv()` to keep reading the data returned from the server. By default, `ssock.recv()` will block if no data is currently available, until data become available or the session is closed.

```
# Send HTTP Request to Server
request = b"GET / HTTP/1.0\r\nHost: " + \
          hostname.encode('utf-8') + b"\r\n\r\n"
ssock.sendall(request)

# Read HTTP Response from Server
response = ssock.recv(2048)
while response:
  pprint.pprint(response.split(b"\r\n"))
  response = ssock.recv(2048)
```

After adding the code above to our TLS client program, we should be able to get the response from an HTTPS web server.

36.6 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. We slightly modify our client program (from Listing 36.1) by changing the following line from `True` to `False`. By doing this, we ask the client program not to check the server's hostname.

```
context.check_hostname = False
```

36.6.1 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. Before launching the attack, we use our client program to interact with these two servers to confirm that we are able to establish TLS sessions with them.

```
$ client.py www.facebook.com
=== Cipher used: ('TLS_CHACHA20_POLY1305_SHA256', 'TLSv1.3', 256)
=== Server certificate:
 ...
 'subject': ...
        (('commonName', '*.facebook.com'),)),

$ client.py www.example.org
=== Cipher used: ('TLS_AES_256_GCM_SHA384', 'TLSv1.3', 256)
=== Server certificate:
 ...
 'subject': ...
        (('commonName', 'www.example.org'),)),
```

The first step for the MITM 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. 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, and we get the following result:

```
$ client.py www.facebook.com     ← The hostname
=== Cipher used: ('TLS_AES_256_GCM_SHA384', 'TLSv1.3', 256)
=== Server certificate:
  ...
  'subject': ...
            (('commonName', 'www.example.org'),)),
                                 ↖ Not matching with the hostname!
```

The result shows that our TLS connection was successful, and 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`. The printout also shows that the certificate's `commonName` field is `www.example.org`, not Facebook. Clearly, the TLS connection is made with the attacker's server, not the Facebook server.

Now imagine that `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, Listing 36.2: Firefox's Warning message after failing to verify the server's identity

```
Did Not Connect:  Potential Security Issue

Firefox detected a potential security threat and did not continue to
www.facebook.com because this website requires a secure connection.

...

Websites prove their identity via certificates. Firefox does not
trust this site because it uses a certificate that is not valid
for www.facebook.com. 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
```

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 36.2 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.

36.6.2 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. It is the application's responsibility 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 Python code shown in Listing 36.1, the following two lines tell TLS to conduct the hostname checking: the first line indicates that the checking is required, and the second line provides the expected hostname to TLS.

```
context.check_hostname = True
...
ssock = context.wrap_socket(sock, server_hostname = hostname, ...)
```

In our experiment, we set the hostname checking to `False`. Now, let's change it back to `True` and connect to Facebook again. This time, we get the error message and the TLS connection failed. That defeats the MITM attack.

```
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED]
      certificate verify failed: Hostname mismatch,
      certificate is not valid for 'www.facebook.com'. (_ssl.c:1123)
```

36.7 TLS Server Program in Python

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 simple web page. 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.

```
#!/usr/bin/python3

import socket, ssl, pprint

html = """
HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n
<!DOCTYPE html><html><body><h1>This is Bank32.com!</h1></body></html>
"""

SERVER_CERT    = './server-certs/mycert_cert.pem'
SERVER_PRIVATE = './server-certs/mycert_key.pem'

# Set up the TLS context
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)          ①
context.load_cert_chain(SERVER_CERT, SERVER_PRIVATE)       ②

# Set up the TCP server
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
sock.bind(('0.0.0.0', 443))
sock.listen(5)

while True:
   newsock, fromaddr = sock.accept()                       ③
   try :
     # Bind the TLS layer to the TCP connection
     ssock = context.wrap_socket(newsock, server_side=True) ④
```

```
    print("TLS connection established")

    data = ssock.recv(1024)
    pprint.pprint("Request: {}".format(data))
    ssock.sendall(html.encode('utf-8'))

    ssock.shutdown(socket.SHUT_RDWR)
    ssock.close()

  except Exception:
    print("TLS connection fails")
    continue
```

- *Creating context.* The program first creates an SSL context, specifying that this is a TLS server (Line ①), which by default, will not ask the client for certificates. 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.

- *Loading private key and certificates.* In Line ②, the server loads the private key and the corresponding certificates. 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. If the private key is password protected, users running the server program will be asked to provide the password.

  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.

- *Setting up TCP Server.* 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 TLS server needs to set up the TCP server. The part is quite standard for socket programming. It creates a TCP socket, binds it to a TCP port (443), 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.

- *Binding TLS to the TCP connection.* Once the TCP is set up, the server program enters the waiting state via the `accept()` system call, waiting for incoming connection requests (Line ③). 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 TLS layer via `context.wrap_socket()`, and the TLS layer will then wait for the client to initiate the TLS handshake protocol (Line ④).

- *Sending data.* 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 over the TLS connection.

Running the server. Before running the server, we need to create a public-key certificate for this server. In our experiment, the name of our HTTPS server is `www.bank32.com`, so we need to get a certificate for this name. We have already covered how to create public-key certificates in Chapter 35, so we will not repeat it.

We create our own root CA certificate, and use it to sign the certificate for `Bank32`. If we test this server using a browser, we need to import this root CA certificate into the browser (see the PKI chapter for details). If we test this server using our own TLS client, we need to tell the client program where to find this root CA certificate (see § 36.5.2).

We need to map the hostname `www.bank32.com` to the server's IP address. Once everything is done, we can start our server (we need to use the superuser privilege because the server listens to port `443`, which is a privileged port).

```
// On the server side:
$ sudo ./tls_server.py
Enter PEM pass phrase:
TLS connection established
"Request: b'GET / HTTP/1.0\\r\\nHost: www.bank32.com\\r\\n\\r\\n'"


// On the client side: use our own tls_client.py program
$ tls_client.py www.bank32.com
=== Cipher used: ('TLS_AES_256_GCM_SHA384', 'TLSv1.3', 256)
=== Server certificate:
{'issuer': ((('commonName', 'www.modelCA.com'),),
  ...
 'subject': ((('commonName', 'www.bank32.com'),),),
 'version': 3}
[b'\nHTTP/1.1 200 OK',
 b'Content-Type: text/html',
 b'',
 b'\n<!DOCTYPE html><html><body><h1>This is Bank32.com!</h1>
     </body></html>\n']
-----------------------------------
[b'']
-----------------------------------
```

36.8 TLS Proxy

TLS can effectively defeat the eavesdropping and Man-In-The-Middle (MITM) attack. That is good for users, but it creates challenges to the organizations that need to inspect the outgoing traffic to prevent important data from being leaked out, such as business secrets, customer data, etc. For non-TLS traffic, such as web traffic (non-web traffic is usually stopped by firewalls), these organizations direct all the HTTP requests to an HTTP proxy, which can inspect and filter the data in the requests, before forwarding them to the intended destination. This is exactly the Man-In-The-Middle "attack", except that in this scenario, it is set up by the organization for the protection purpose, not an attack.

The main purposes of TLS is exactly to prevent the MITM attack. With most websites switching to HTTPS, can these organizations still use web proxies to inspect the outgoing web traffic? This seems to contradict to what TLS can do, but it is possible. In this section, we will explain how such type of proxy works. Moreover, we will show how to implement a simple HTTPS proxy.

36.8.1 The Idea of TLS Proxy

We should keep in mind that TLS depends on the public-key infrastructure (PKI). If any essential element of PKI is compromised, the protection of TLS against MITM will be gone. One essential element of PKI is the trusted CA certificate installed on the user machines. If one of them gets compromised (e.g., its private key is stolen by the attack), the attacker can successfully launch the MITM attack. As we have discussed in Chapter 35, these attacks have happened before.

Organizations who want to establish an HTTPS proxy (basically a MITM "attack") can use the same idea: they "compromise" one of the trust CA certificates installed on their users' machines. Obviously, they will not actually compromise any CA; instead, they will install their own certificates on users' machines as trust CA certificates. This is equivalent to what is done in the successful MITM attacks. In this scenario, the machines belong to the organization, so the organization has the power to install trust CA certificates.

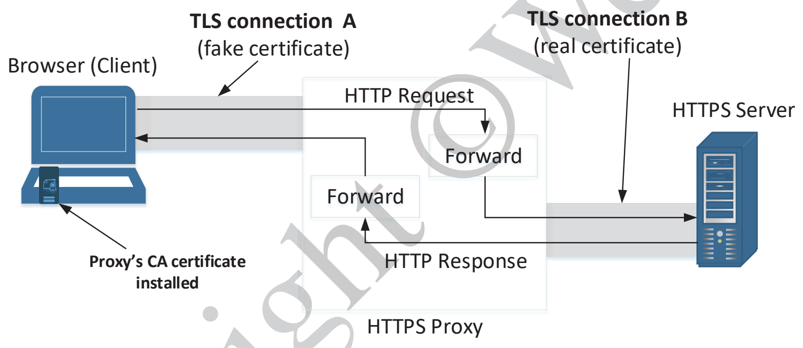

Figure 36.9: How the HTTPS proxy works

Once the proxy's CA certificate `T` is trusted by the client, the proxy can now use `T` to generate fake certificates for any HTTPS server, and can therefore pretend to be any HTTPS server. For example, if the proxy wants to become a proxy between the client and `www.example.com`, the proxy just needs to generate a fake certificate for this server, and sign it using `T`. During the TLS handshake with the client (see TLS connection A in Figure 36.9), this fake certificate will be sent to the client, who will use the trusted certificate `T` to verify the server certificate. Obviously, the verification will succeed. Moreover, since the common name field in the fake certificate is `www.example.com`, the hostname check on the client will also succeed. Therefore, the client is fooled to believe that the proxy is indeed `www.example.com`.

After the client and the proxy have established the TLS connection, the proxy will establish the second TLS connection, and this time, with the real HTTPS server (see the TLS connection B in Figure 36.9). The proxy then forward the request from the client to the real server, and forward the server's response back to the client.

36.8.2 Certificate Setup

We first create a CA certificate for the proxy. How to create CA certificate is already covered in the PKI chapter of this book, so we will not repeat it. We assume that the CA's public-key certificate is stored in `modelCA_cert.pem`, while its private key is stored in `modelCA_key.pem`. We also need to load the CA certificate into our browser as a trusted CA certificate.

We will now use the CA certificate to generate fake server certificates. A real-world HTTPS proxy generate the fake certificate automatically on the fly. For the sake of simplicity, we will do it manually. Assuming that we would like to implement a proxy for the `www.example.com` server, we need to create a fake certificate for this server. We wrote the following Shell script to generate a fake certificate:

```
#!/bin/bash

mySubject="/CN=www.example.com"

# Generate RSA key pair and certificate request
openssl req -newkey rsa:2048 -batch -sha256 \
            -keyout  proxy_key.pem -out proxy.csr  \
            -subj "$mySubject"  -nodes

# Generate certificate using the CA's certificate
openssl ca -policy policy_anything -md sha256 -days 3650 \
           -in proxy.csr -out proxy_cert.pem -batch \
           -cert modelCA_cert.pem -keyfile modelCA_key.pem \
           -passin pass:dees
```

After running the shell script above, we will get a fake certificate (`proxy_cert.pem`) with the common name being `www.example.com`. The corresponding private key is stored in `proxy_key.pem`. When the proxy establishes a TLS connection with the client, it will use this fake certificate and its private key.

36.8.3 The code for a Simple HTTPS Proxy

The proxy is actually a combination of the TLS client and server programs. To the browser, the TLS proxy is just a server program, which takes the HTTP requests from the browser (the client), and return HTTP responses to it. The proxy does not generate any HTTP responses; instead, it forwards the HTTP requests to the actual web server, and then get the HTTP responses from the web server. To the actual web server, the TLS proxy is just a client program. After getting the response, the proxy forwards the response to the browser, the real client. Therefore, by integrating the client and server programs implemented previously, we should be able to get a basic proxy working.

Because implementing an HTTPS proxy is a part of the task in a SEED lab, I will not provide the complete proxy code in the book. Instead, I will provide the skeleton code and guidelines on how to implement such a proxy.

The `main()` function. In the main function, we will set up the TLS server (using the fake certificate), and then we wait for the connection from the client. A browser may simultaneously send multiple HTTP requests to the server, each using a separate TCP connection. It is better to deal with each TCP connection in a separate thread, so the proxy program can handle multiple simultaneous requests. The skeleton of the main function is listed below. It shows how to create a thread to handle each TLS connection.

```
PROXY_CERT    = './server-certs/proxy_cert.pem'
PROXY_PRIVATE = './server-certs/proxy_key.pem'

def main():
  # Set up TCP server
  sock_listen = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
  sock_listen.bind(('0.0.0.0', 443))
  sock_listen.listen(5)

  # Initialize TLS server context
  context_srv = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
  context_srv.load_cert_chain(PROXY_CERT, PROXY_PRIVATE)

  while True:
    newsock, fromaddr = sock_listen.accept()
    ssock = context.wrap_socket(newsock, server_side=True)

    x = threading.Thread(target=process_request, args=(ssock,))
    x.start()

if __name__ == "__main__":
    main()
```

Processing request. The thread will execute the code in the `process_request` function, which forwards the HTTP request from the browser to the server, and then forward the HTTP response from the server to the browser. A code skeleton is provided in the following:

```
def process_request(client_ssock):

  # Create a TLS connection with the server (code omitted)
  # The connection is stored in the server_ssock variable

  request = client_ssock.recv(2048)    # Get request from client
  if request:
    server_ssock.sendall(request)      # Forward request to server

    response = server_ssock.recv(2048) # Get response from server
    while response:
        client_ssock.sendall(response)     # Forward to client
        response = server_ssock.recv(2048) # Get more response

  # Close the TLS connection with the client
  client_ssock.shutdown(socket.SHUT_RDWR)
  client_ssock.close()
```

Modifying data. Since the HTTPS proxy can intercept the data between the client and the server, it can also make changes to the data. For example, if we use this proxy to intercept the traffic between a browser and the Google search website, we can add the following line to the code after we get the request from the client. It will change the word `cybersecurity` in the query with `"SEED Labs"` (the space needs to be encoded using `+`).

```
request = request.replace("cybersecurity", "SEED+Labs")
```

With this change, every time you search for `cybersecurity`, the actual query sent to Google is `"SEED Labs"`. You will see that from the browser. We can apply the same method to modify the response from the server, but it should be noted that the responses data could be encoded or compressed, so the simple string replacement may not work.

36.9 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 tamper detection 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.

The chapter explains how TLS works under the hood, so readers can gain insights about TLS, which help them understand what is done by TLS and what is not done by TLS. A common mistake in TLS programming is failing to check the server's identity, so a client may be using TLS to "securely" communicate with an attacker, instead of with its intended server. The reason for this mistake is that many developers think that TLS automatically checks that. After reading this chapter, readers should now know that TLS does not actually do that; nor is TLS able to do that, because TLS does not know what the intended server is, unless the application tells TLS about that.

❐ Hands-on Lab Exercise

We have developed a SEED lab for this chapter. The lab is called *Transport Layer Security (TLS) Lab*. Another lab, the *VPN Lab*, also depends on the content of this chapter. Both labs are hosted on the SEED website: `https://seedsecuritylabs.org`.

❐ Problems and Resources

The homework problems, slides, and source code for this chapter can be downloaded from the book's website: `https://www.handsonsecurity.net/`.