CHAPTER 17

Security at the Transport Layer: SSL and TLS

Objectives

This chapter has several objectives:

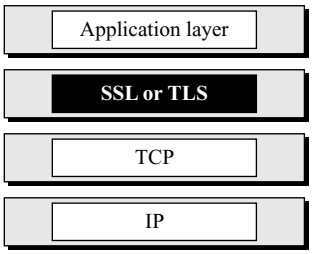
- ❏ To discuss the need for security services at the transport layer of the Internet model
- ❏ To discuss the general architecture of SSL
- ❏ To discuss the general architecture of TLS
- ❏ To compare and contrast SSL and TLS

Transport layer security provides end-to-end security services for applications that use a reliable transport layer protocol such as TCP. The idea is to provide security services for transactions on the Internet. For example, when a customer shops online, the following security services are desired:

1. The customer needs to be sure that the server belongs to the actual vendor, not an impostor. The customer does not want to give an impostor her credit card number (entity authentication).
2. The customer and the vendor need to be sure that the contents of the message are not modified during transmission (message integrity).
3. The customer and the vendor need to be sure that an impostor does not intercept sensitive information such as a credit card number (confidentiality).

Two protocols are dominant today for providing security at the transport layer: the **Secure Sockets Layer (SSL) Protocol** and the **Transport Layer Security (TLS) Protocol.** The latter is actually an IETF version of the former. We first discuss SSL, then TLS, and then compare and contrast the two. Figure 17.1 shows the position of SSL and TLS in the Internet model.

Figure 17.1 *Location of SSL and TLS in the Internet model*



One of the goals of these protocols is to provide server and client authentication, data confidentiality, and data integrity. Application-layer client/server programs, such as **Hypertext Transfer Protocol (HTTP),** that use the services of TCP can encapsulate their data in SSL packets. If the server and client are capable of running SSL (or TLS) programs then the client can use the URL *https://*… instead of *http://*… to allow HTTP messages to be encapsulated in SSL (or TLS) packets. For example, credit card numbers can be safely transferred via the Internet for online shoppers.

17.1 SSL ARCHITECTURE

SSL is designed to provide security and compression services to data generated from the application layer. Typically, SSL can receive data from any application layer protocol, but usually the protocol is HTTP. The data received from the application is compressed (optional), signed, and encrypted. The data is then passed to a reliable transport layer protocol such as TCP. Netscape developed SSL in 1994. Versions 2 and 3 were released in 1995. In this chapter, we discuss SSLv3.

Services

SSL provides several services on data received from the application layer.

Fragmentation

First, SSL divides the data into blocks of 2^{14} bytes or less.

Compression

Each fragment of data is compressed using one of the lossless compression methods negotiated between the client and server. This service is optional.

Message Integrity

To preserve the integrity of data, SSL uses a keyed-hash function to create a MAC.

Confidentiality

To provide confidentiality, the original data and the MAC are encrypted using symmetric-key cryptography.

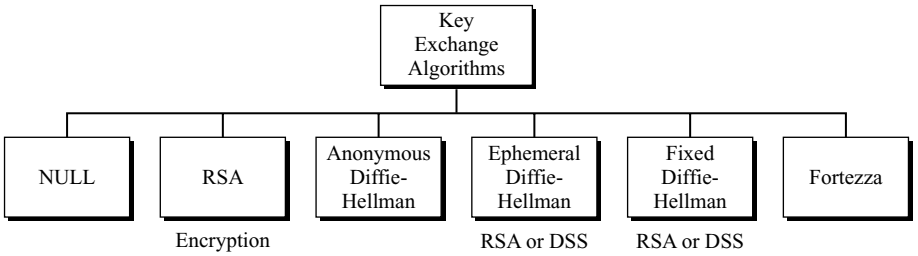
Framing

A header is added to the encrypted payload. The payload is then passed to a reliable transport layer protocol.

Key Exchange Algorithms

As we will see later, to exchange an authenticated and confidential message, the client and the server each need six cryptographic secrets (four keys and two initialization vectors). However, to create these secrets, one pre-master secret must be established between the two parties. SSL defines six key-exchange methods to establish this pre-master secret: NULL, RSA, anonymous Diffie-Hellman, ephemeral Diffie-Hellman, fixed Diffie-Hellman, and Fortezza, as shown in Figure 17.2.

Figure 17.2 *Key-exchange methods*



NULL

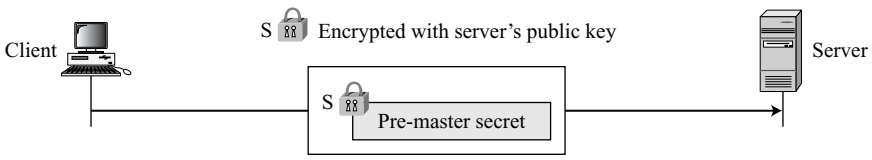
There is no key exchange in this method. No pre-master secret is established between the client and the server.

Both client and server need to know the value of the pre-master secret.

RSA

In this method, the pre-master secret is a 48-byte random number created by the client, encrypted with the server's RSA public key, and sent to the server. The server needs to send its RSA encryption/decryption certificate. Figure 17.3 shows the idea.

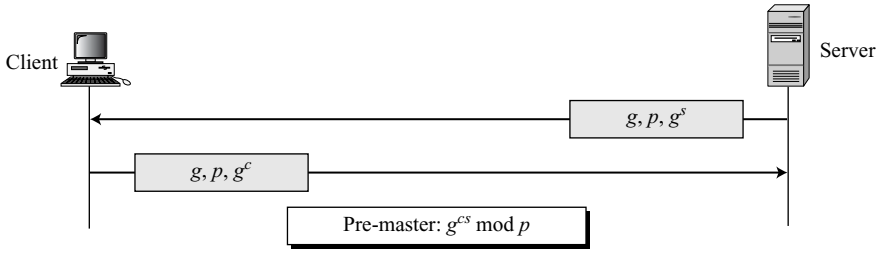
Figure 17.3 *RSA key exchange; server public key*



Anonymous Diffie-Hellman

This is the simplest and most insecure method. The pre-master secret is established between the client and server using the Diffie-Hellman (DH) protocol. The Diffie-Hellman half-keys are sent in plaintext. It is called **anonymous Diffie-Hellman** because neither party is known to the other. As we have discussed, the most serious disadvantage of this method is the man-in-the-middle attack. Figure 17.4 shows the idea.

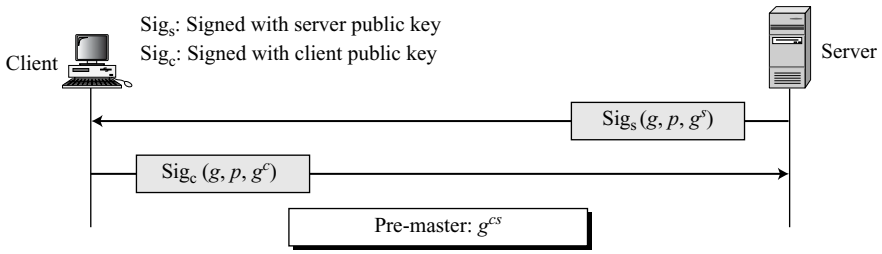
Figure 17.4 *Anonymous Diffie-Hellman key exchange*



Ephemeral Diffie-Hellman

To thwart the man-in-the-middle attack, the **ephemeral Diffie-Hellman** key exchange can be used. Each party sends a Diffie-Hellman key signed by its private key. The receiving party needs to verify the signature using the public key of the sender. The public keys for verification are exchanged using either RSA or DSS digital signature certificates. Figure 17.5 shows the idea.

Figure 17.5 *Ephemeral Diffie-Hellman key exchange*



Fixed Diffie-Hellman

Another solution is the **fixed Diffie-Hellman** method. All entities in a group can prepare fixed Diffie-Hellman parameters (g and p). Then each entity can create a fixed Diffie-Hellman half-key (g^x). For additional security, each individual half-key is inserted into a certificate verified by a certification authority (CA). In other words, the

two parties do not directly exchange the half-keys; the CA sends the half-keys in an RSA or DSS special certificate. When the client needs to calculate the pre-master, it uses its own fixed half-key and the server half-key received in a certificate. The server does the same, but in the reverse order. Note that no key-exchange messages are passed in this method; only certificates are exchanged.

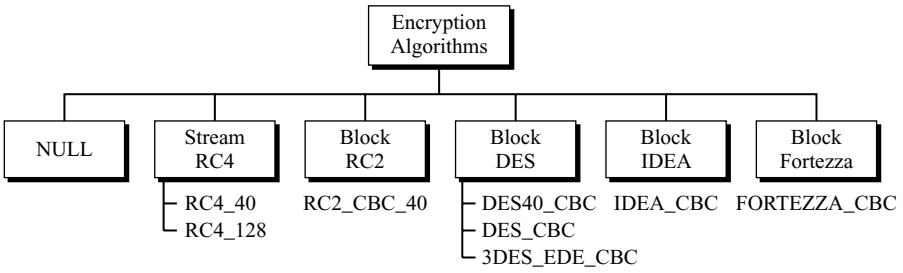
Fortezza

Fortezza (derived from the Italian word for fortress) is a registered trademark of the U.S. National Security Agency (NSA). It is a family of security protocols developed for the Defense Department. We do not discuss Fortezza in this text because of its complexity.

Encryption/Decryption Algorithms

There are several choices for the encryption/decryption algorithm. We can divide the algorithms into 6 groups as shown in Figure 17.6. All block protocols use an 8-byte initialization vector (IV) except for Fortezza, which uses a 20-byte IV.

Figure 17.6 *Encryption/decryption algorithms*



NULL

The NULL category simply defines the lack of an encryption/decryption algorithm.

Stream RC

Two RC algorithms are defined in stream mode: RC4-40 (40-bit key) and RC4-128 (128-bit key).

Block RC

One RC algorithm is defined in block mode: RC2_CBC_40 (40-bit key).

DES

All DES algorithms are defined in block mode. DES40_CBC uses a 40-bit key. Standard DES is defined as DES_CBC. 3DES_EDE_CBC uses a 168-bit key.

IDEA

The one IDEA algorithm defined in block mode is IDEA_CBC, with a 128-bit key.

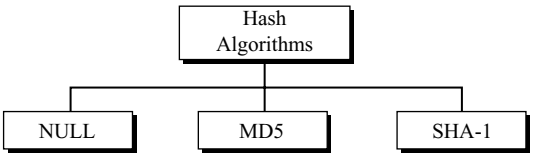
Fortezza

The one Fortezza algorithm defined in block mode is FORTEZZA_CBC, with a 96-bit key.

Hash Algorithms

SSL uses hash algorithms to provide message integrity (message authentication). Three hash functions are defined, as shown in Figure 17.7.

Figure 17.7 *Hash algorithms for message integrity*



Null

The two parties may decline to use an algorithm. In this case, there is no hash function and the message is not authenticated.

MD5

The two parties may choose MD5 as the hash algorithm. In this case, a 128-key MD5 hash algorithm is used.

SHA-1

The two parties may choose SHA as the hash algorithm. In this case, a 160-bit SHA-1 hash algorithm is used.

Cipher Suite

The combination of key exchange, hash, and encryption algorithms defines a **cipher suite** for each SSL session. Table 17.1 shows the suites used in the United States. We have not included those that are used for export. Note that not all combinations of key exchange, message integrity, and message authentication are in the list.

Each suite starts with the term "SSL" followed by the key exchange algorithm. The word "WITH" separates the key exchange algorithm from the encryption and hash algorithms. For example,

```
SSL_DHE_RSA_WITH_DES_CBC_SHA
```

defines DHE_RSA (ephemeral Diffie-Hellman with RSA digital signature) as the key exchange with DES_CBC as the encryption algorithm and SHA as the hash algorithm.

Table 17.1 *SSL cipher suite list*

Cipher suite	*Key Exchange*	*Encryption*	*Hash*
SSL_NULL_WITH_NULL_NULL	NULL	NULL	NULL
SSL_RSA_WITH_NULL_MD5	RSA	NULL	MD5
SSL_RSA_WITH_NULL_SHA	RSA	NULL	SHA-1
SSL_RSA_WITH_RC4_128_MD5	RSA	RC4	MD5
SSL_RSA_WITH_RC4_128_SHA	RSA	RC4	SHA-1
SSL_RSA_WITH_IDEA_CBC_SHA	RSA	IDEA	SHA-1
SSL_RSA_WITH_DES_CBC_SHA	RSA	DES	SHA-1
SSL_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES	SHA-1
SSL_DH_anon_WITH_RC4_128_MD5	DH_anon	RC4	MD5
SSL_DH_anon_WITH_DES_CBC_SHA	DH_anon	DES	SHA-1
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	DH_anon	3DES	SHA-1
SSL_DHE_RSA_WITH_DES_CBC_SHA	DHE_RSA	DES	SHA-1
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES	SHA-1
SSL_DHE_DSS_WITH_DES_CBC_SHA	DHE_DSS	DES	SHA-1
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES	SHA-1
SSL_DH_RSA_WITH_DES_CBC_SHA	DH_RSA	DES	SHA-1
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES	SHA-1
SSL_DH_DSS_WITH_DES_CBC_SHA	DH_DSS	DES	SHA-1
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH_DSS	3DES	SHA-1
SSL_FORTEZZA_DMS_WITH_NULL_SHA	Fortezza	NULL	SHA-1
SSL_FORTEZZA_DMS_WITH_FORTEZZA_CBC_SHA	Fortezza	Fortezza	SHA-1
SSL_FORTEZZA_DMS_WITH_RC4_128_SHA	Fortezza	RC4	SHA-1

Note that *DH* is fixed Diffie-Hellman, *DHE* is ephemeral Diffie-Hellman, and *DH-anon* is anonymous Diffie-Hellman.

Compression Algorithms

As we said before, compression is optional in SSLv3. No specific compression algorithm is defined for SSLv3. Therefore, the default compression method is NULL. However, a system can use whatever compression algorithm it desires.

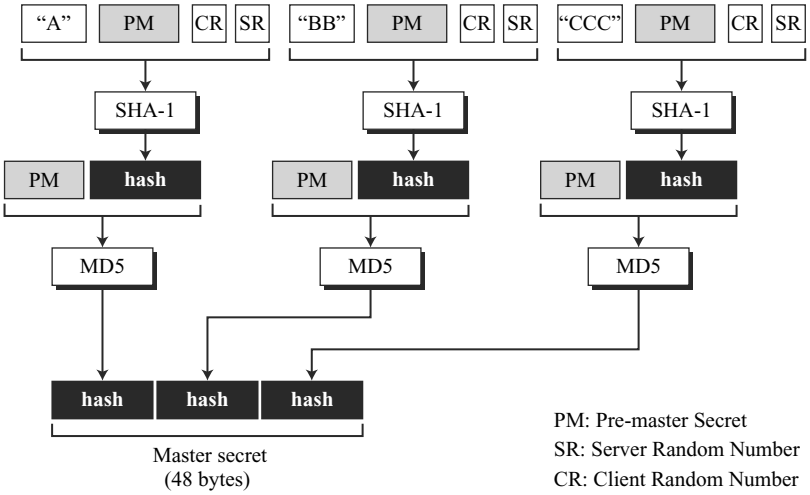
Cryptographic Parameter Generation

To achieve message integrity and confidentiality, SSL needs six cryptographic secrets, four keys and two IVs. The client needs one key for message authentication (HMAC), one key for encryption, and one IV for block encryption. The server needs the same. SSL requires that the keys for one direction be different from those for the other direction. If there is an attack in one direction, the other direction is not affected. The parameters are generated using the following procedure:

1. The client and server exchange two random numbers; one is created by the client and the other by the server.
2. The client and server exchange one pre-master secret using one of the key-exchange algorithms we discussed previously.

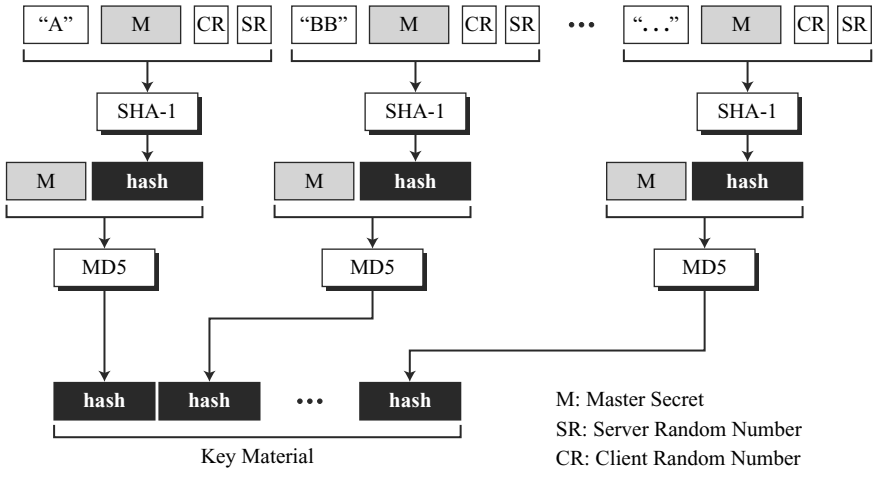
3. A 48-byte **master secret** is created from the **pre-master secret** by applying two hash functions (SHA-1 and MD5), as shown in Figure 17.8.

Figure 17.8 *Calculation of master secret from pre-master secret*



4. The master secret is used to create variable-length **key material** by applying the same set of hash functions and prepending with different constants as shown in Figure 17.9. The module is repeated until key material of adequate size is created.

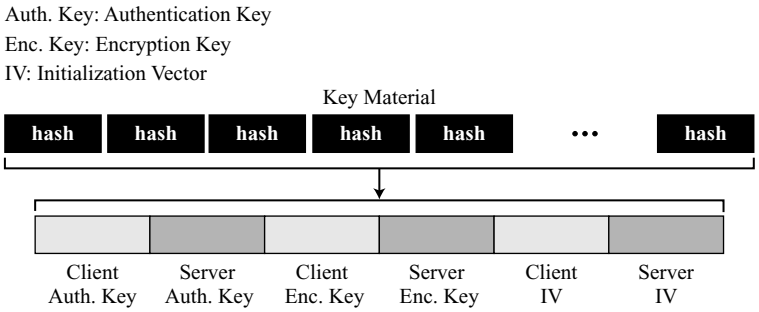
Figure 17.9 *Calculation of key material from master secret*



Note that the length of the key material block depends on the cipher suite selected and the size of keys needed for this suite.

5. Six different keys are extracted from the key material, as shown in Figure 17.10

Figure 17.10 *Extractions of cryptographic secrets from key material*



Sessions and Connections

SSL differentiates a **connection** from a **session.** Let us elaborate on these two terms here. A session is an association between a client and a server. After a session is established, the two parties have common information such as the session identifier, the certificate authenticating each of them (if necessary), the compression method (if needed), the cipher suite, and a master secret that is used to create keys for message authentication encryption.

For two entities to exchange data, the establishment of a session is necessary, but not sufficient; they need to create a connection between themselves. The two entities exchange two random numbers and create, using the master secret, the keys and parameters needed for exchanging messages involving authentication and privacy.

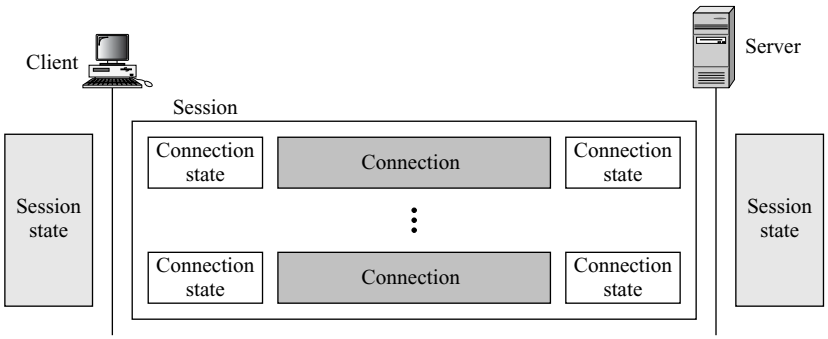
A session can consist of many connections. A connection between two parties can be terminated and reestablished within the same session. When a connection is terminated, the two parties can also terminate the session, but it is not mandatory. A session can be suspended and resumed again.

To create a new session, the two parties need to go through a negotiation process. To resume an old session and create only a new connection, the two parties can skip part of the negotiation process and go through a shorter one. There is no need to create a master secret when a session is resumed.

The separation of a session from a connection prevents the high cost of creating a master secret. By allowing a session to be suspended and resumed, the process of the master secret calculation can be eliminated. Figure 17.11 shows the idea of a session and connections inside that session.

In a session, one party has the role of a client and the other the role of a server; in a connection, both parties have equal roles, they are peers.

Figure 17.11 *A session and connections*



Session State

A session is defined by a session state, a set of parameters established between the server and the client. Table 17.2 shows the list of parameters for a session state.

Table 17.2 *Session state parameters*

Parameter	*Description*
Session ID	A server-chosen 8-bit number defining a session.
Peer Certificate	A certificate of type X509.v3. This parameter may by empty (null).
Compression Method	The compression method.
Cipher Suite	The agreed-upon cipher suite.
Master Secret	The 48-byte secret.
Is resumable	A yes-no flag that allows new connections in an old session.

Connection State

A connection is defined by a connection state, a set of parameters established between two peers. Table 17.3 shows the list of parameters for a connection state.

SSL uses two attributes to distinguish cryptographic secrets: *write* and *read.* The term *write* specifies the key used for signing or encrypting outbound messages. The term *read* specifies the key used for verifying or decrypting inbound messages. Note that the *write* key of the client is the same as the *read* key of the server; the *read* key of the client is the same as the *write* key of the server.

The client and the server have six different cryptography secrets: three *read* secrets and three *write* secrets.
The *read* secrets for the client are the same as the write secrets for the server and vice versa.

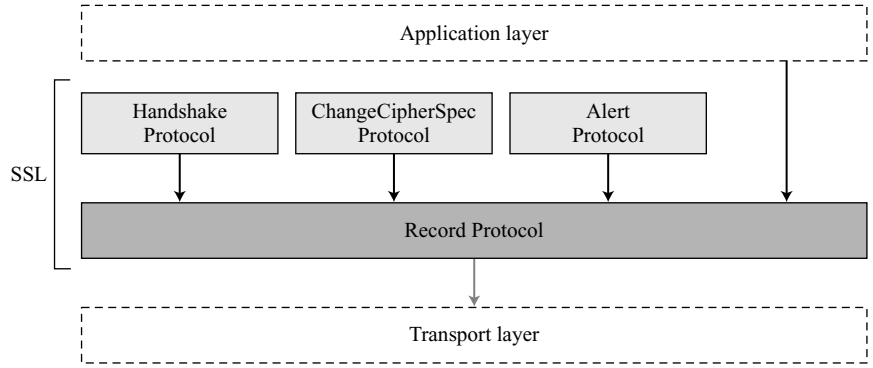
Table 17.3 *Connection state parameters*

Parameter	*Description*
Server and client random numbers	A sequence of bytes chosen by the server and client for each connection.
Server write MAC secret	The outbound server MAC key for message integrity. The server uses it to sign; the client uses it to verify.
Client write MAC secret	The outbound client MAC key for message integrity. The client uses it to sign; the server uses it to verify.
Server write secret	The outbound server encryption key for message integrity.
Client write secret	The outbound client encryption key for message integrity.
Initialization vectors	The block ciphers in CBC mode use initialization vectors (IVs). One initialization vector is defined for each cipher key during the negotiation, which is used for the first block exchange. The final cipher text from a block is used as the IV for the next block.
Sequence numbers	Each party has a sequence number. The sequence number starts from 0 and increments. It must not exceed $2^{64} - 1$.

17.2 FOUR PROTOCOLS

We have discussed the idea of SSL without showing how SSL accomplishes its tasks. SSL defines four protocols in two layers, as shown in Figure 17.12. The Record Protocol is the carrier. It carries messages from three other protocols as well as the data coming from the application layer. Messages from the Record Protocol are payloads to the transport layer, normally TCP. The Handshake Protocol provides security parameters for the Record Protocol. It establishes a cipher set and provides keys and security

Figure 17.12 *Four SSL protocols*

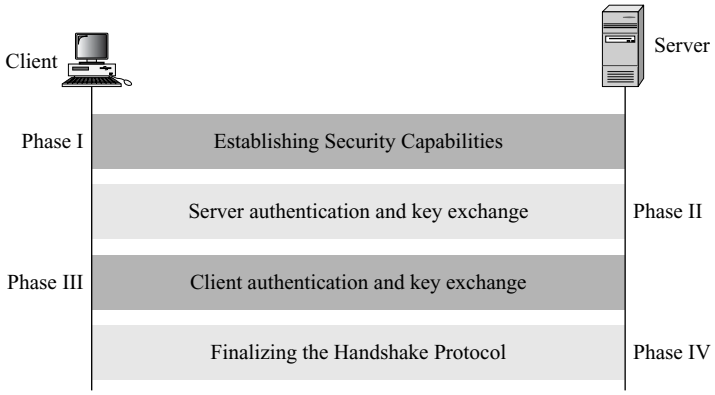


parameters. It also authenticates the server to the client and the client to the server if needed. The ChangeCipherSpec Protocol is used for signalling the readiness of cryptographic secrets. The Alert Protocol is used to report abnormal conditions. We will briefly discuss these protocols in this section.

Handshake Protocol

The **Handshake Protocol** uses messages to negotiate the cipher suite, to authenticate the server to the client and the client to the server if needed, and to exchange information for building the cryptographic secrets. The handshaking is done in four phases, as shown in Figure 17.13.

Figure 17.13 *Handshake Protocol*



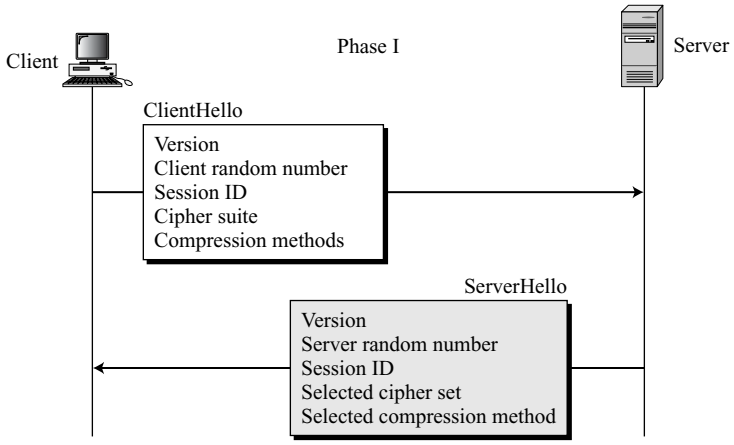
Phase I: Establishing Security Capability

In Phase I, the client and the server announce their security capabilities and choose those that are convenient for both. In this phase, a session ID is established and the cipher suite is chosen. The parties agree upon a particular compression method. Finally, two random numbers are selected, one by the client and one by the server, to be used for creating a master secret as we saw before. Two messages are exchanged in this phase: ClientHello and ServerHello messages. Figure 17.14 gives additional details about Phase I.

ClientHello The client sends the ClientHello message. It contains the following:

a. The highest SSL version number the client can support.
b. A 32-byte random number (from the client) that will be used for master secret generation.
c. A session ID that defines the session.
d. A cipher suite that defines the list of algorithms that the client can support.
e. A list of compression methods that the client can support.

Figure 17.14 *Phase I of Handshake Protocol*



ServerHello The server responds to the client with a ServerHello message. It contains the following:

a. An SSL version number. This number is the lower of two version numbers: the highest supported by the client and the highest supported by the server.
b. A 32-byte random number (from the server) that will be used for master secret generation.
c. A session ID that defines the session.
d. The selected cipher set from the client list.
e. The selected compression method from the client list.

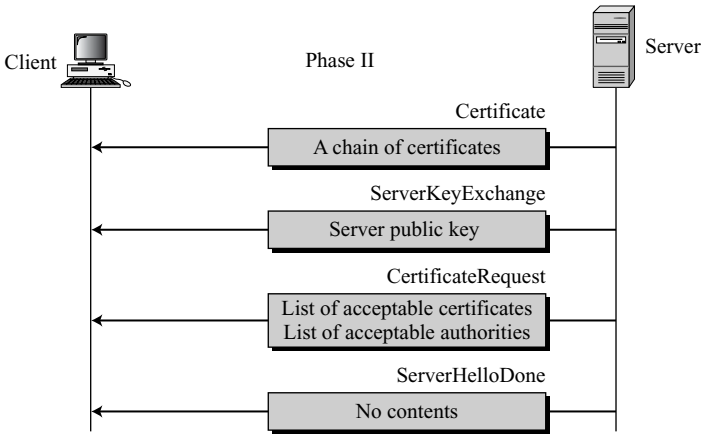
After Phase I, the client and server know the following:

- ❑ ***The version of SSL***
- ❑ ***The algorithms for key exchange, message authentication, and encryption***
- ❑ ***The compression method***
- ❑ ***The two random numbers for key generation***

Phase II: Server Key Exchange and Authentication

In phase II, the server authenticates itself if needed. The sender may send its certificate, its public key, and may also request certificates from the client. At the end, the server announces that the serverHello process is done. Figure 17.15 gives additional details about Phase II.

Figure 17.15 *Phase II of Handshake Protocol*



Certificate If it is required, the server sends a Certificate message to authenticate itself. The message includes a list of certificates of type X.509. The certificate is not needed if the key-exchange algorithm is anonymous Diffie-Hellman.

ServerKeyExchange After the Certificate message, the server sends a ServerKeyExchange message that includes its contribution to the pre-master secret. This message is not required if the key-exchange method is RSA or fixed Diffie-Hellman.

CertificateRequest The server may require the client to authenticate itself. In this case, the server sends a CertificateRequest message in Phase II that asks for certification in Phase III from the client. The server cannot request a certificate from the client if it is using anonymous Diffie-Hellman.

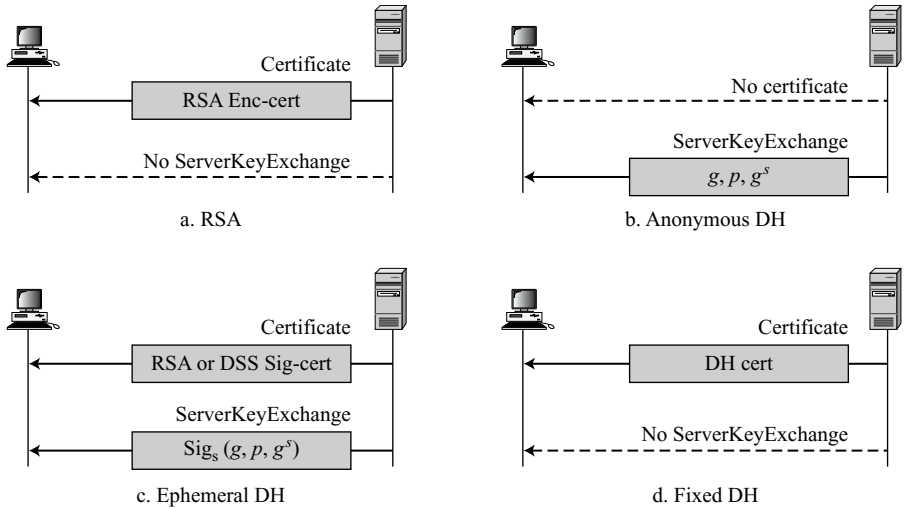
ServerHelloDone The last message in Phase II is the ServerHelloDone message, which is a signal to the client that Phase II is over and that the client needs to start Phase III.

After Phase II,

- ***The server is authenticated to the client.***
- ***The client knows the public key of the server if required.***

Let us elaborate on the server authentication and the key exchange in this phase. The first two messages in this phase are based on the key-exchange method. Figure 17.16 shows four of six methods we discussed before. We have not included the NULL method because there is no exchange. We have not included the Fortezza method because we do not discuss it in depth in this book.

Figure 17.16 *Four cases in Phase II*



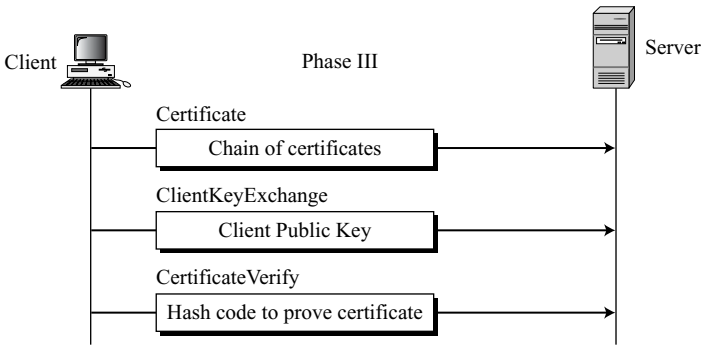
- **RSA.** In this method, the server sends its RSA encryption/decryption public-key certificate in the first message. The second message, however, is empty because the pre-master secret is generated and sent by the client in the next phase. Note that the public-key certificate authenticates the server to the client. When the server receives the pre-master secret, it decrypts it with its private key. The possession of the private key by the server is proof that the server is the entity that it claims to be in the public-key certificate sent in the first message.
- **Anonymous DH.** In this method, there is no Certificate message. An anonymous entity does not have a certificate. In the ServerKeyExchange message, the server sends the Diffie-Hellman parameters and its half-key. Note that the server is not authenticated in this method.
- **Ephemeral DH.** In this method, the server sends either an RSA or a DSS digital signature certificate. The private key associated with the certificate allows the server to sign a message; the public key allows the recipient to verify the signature. In the second message, the server sends the Diffie-Hellman parameters and the half-key signed by its private key. Other text is also sent. The server is authenticated to the client in this method, not because it sends the certificate, but because it signs the parameters and keys with its private key. The possession of the private key is proof that the server is the entity that it claims to be in the certificate. If an impostor copies and sends the certificate to the client, pretending that it is the server claimed in the certificate, it cannot sign the second message because it does not have the private key.
- **Fixed DH.** In this method, the server sends an RSA or DSS digital signature certificate that includes its registered DH half-key. The second message is empty. The

certificate is signed by the CA's private key and can be verified by the client using the CA's public key. In other words, the CA is authenticated to the client and the CA claims that the half-key belongs to the server.

Phase III: Client Key Exchange and Authentication

Phase III is designed to authenticate the client. Up to three messages can be sent from the client to the server, as shown in Figure 17.17.

Figure 17.17 *Phase III of Handshake Protocol*



Certificate To certify itself to the server, the client sends a Certificate message. Note that the format is the same as the Certificate message sent by the server in Phase II, but the contents are different. It includes the chain of certificates that certify the client.This message is sent only if the server has requested a certificate in Phase II. If there is a request and the client has no certificate to send, it sends an Alert message (part of the Alert Protocol to be discussed later) with a warning that there is no certificate. The server may continue with the session or may decide to abort.

ClientKeyExchange After sending the Certificate message, the client sends a ClientKeyExchange message, which includes its contribution to the pre-master secret. The contents of this message are based on the key-exchange algorithm used. If the method is RSA, the client creates the entire pre-master secret and encrypts it with the RSA public key of the server. If the method is anonymous or ephemeral Diffie-Hellman, the client sends its Diffie-Hellman half-key. If the method is Fortezza, the client sends the Fortezza parameters. The contents of this message are empty if the method is fixed Diffie-Hellman.

CertificateVerify If the client has sent a certificate declaring that it owns the public key in the certificate, it needs to prove that it knows the corresponding private key. This is needed to thwart an impostor who sends the certificate and claims that it comes from the client. The proof of private-key possession is done by creating a message and signing it with the private key. The server can verify the message with the public key

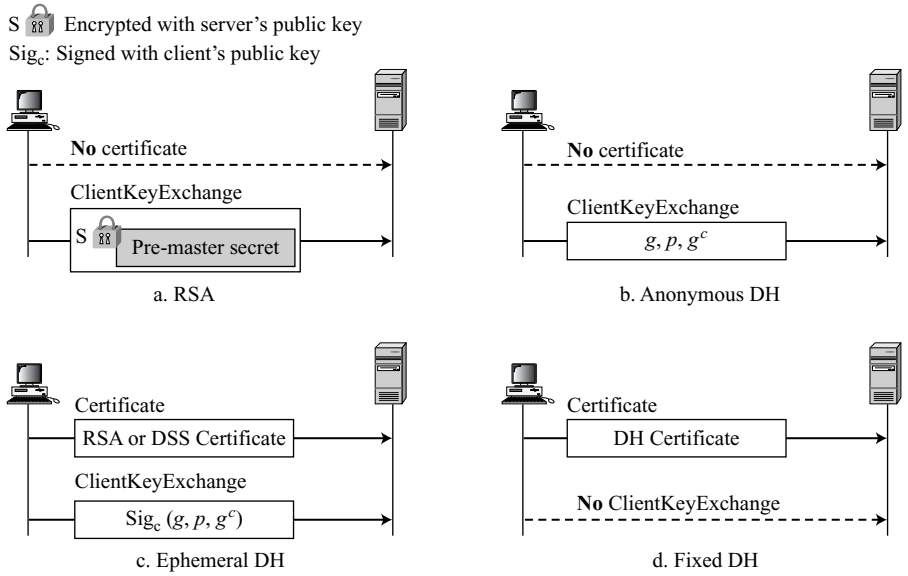
already sent to ensure that the certificate actually belongs to the client. Note that this is possible if the certificate has a signing capability; a pair of keys, public and private, is involved. The certificate for fixed Diffie-Hellman cannot be verified this way.

After Phase III,

- ***The client is authenticated for the server.***
- ***Both the client and the server know the pre-master secret.***

Let us elaborate on the client authentication and the key exchange in this phase. The three messages in this phase are based on the key-exchange method. Figure 17.18 shows four of the six methods we discussed before. Again, we have not included the NULL method or the Fortezza method.

Figure 17.18 *Four cases in Phase III*



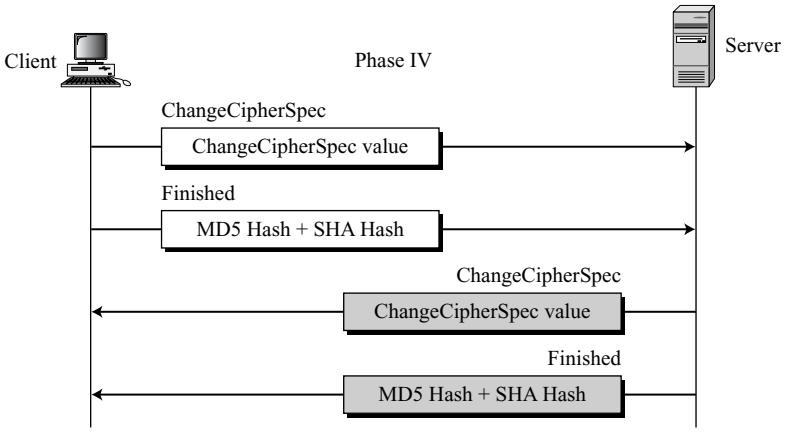
- **RSA.** In this case, there is no Certificate message unless the server has explicitly requested one in Phase II. The ClientKeyExchange method includes the pre-master key encrypted with the RSA public key received in Phase II.
- **Anonymous DH.** In this method, there is no Certificate message. The server does not have the right to ask for the certificate (in Phase II) because both the client and the server are anonymous. In the ClientKeyExchange message, the server sends the Diffie-Hellman parameters and its half-key. Note that the client is not authenticated to the server in this method.

- **Ephemeral DH.** In this method, the client usually has a certificate. The server needs to send its RSA or DSS certificate (based on the agreed-upon cipher set). In the ClientKeyExchange message, the client signs the DH parameters and its half-key and sends them. The client is authenticated to the server by signing the second message. If the client does not have the certificate, and the server asks for it, the client sends an Alert message to warn the client. If this is acceptable to the server, the client sends the DH parameters and key in plaintext. Of course, the client is not authenticated to the server in this situation.
- **Fixed DH.** In this method, the client usually sends a DH certificate in the first message. Note that the second message is empty in this method. The client is authenticated to the server by sending the DH certificate.

Phase IV: Finalizing and Finishing

In Phase IV, the client and server send messages to change cipher specification and to finish the handshaking protocol. Four messages are exchanged in this phase, as shown in Figure 17.19.

Figure 17.19 *Phase IV of Handshake Protocol*



ChangeCipherSpec The client sends a ChangeCipherSpec message to show that it has moved all of the cipher suite set and the parameters from the pending state to the active state. This message is actually part of the ChangeCipherSpec Protocol that we will discuss later.

Finished The next message is also sent by the client. It is a Finished message that announces the end of the handshaking protocol by the client.

ChangeCipherSpec The server sends a ChangeCipherSpec message to show that it has also moved all of the cipher suite set and parameters from the pending state to the active state. This message is part of the ChangeCipherSpec Protocol, which will be discussed later.

After Phase IV, the client and server are ready to exchange data.

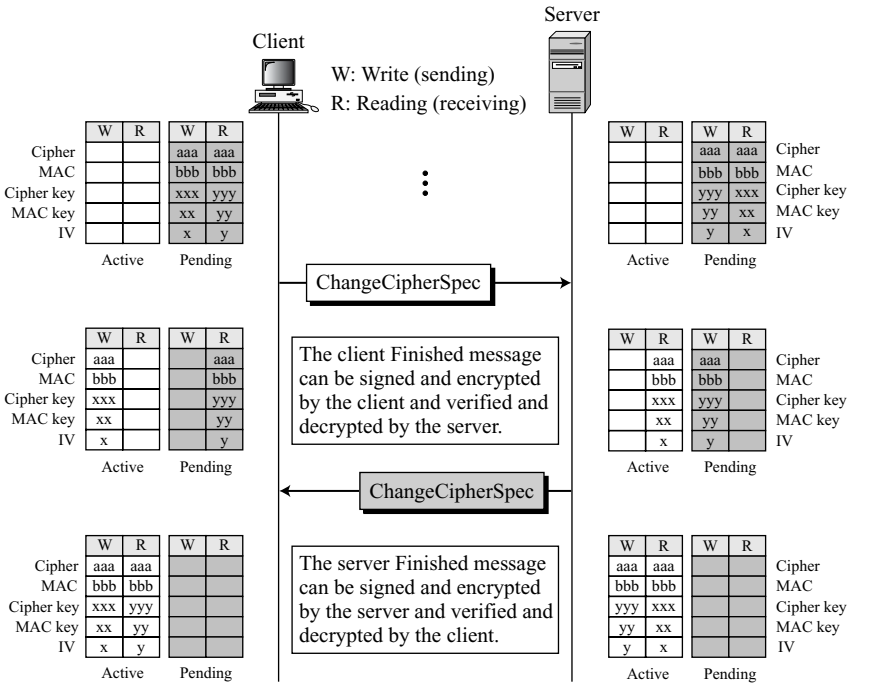
Finished Finally, the server sends a Finished message to show that handshaking is totally completed.

ChangeCipherSpec Protocol

We have seen that the negotiation of the cipher suite and the generation of cryptographic secrets are formed gradually during the Handshake Protocol. The question now is: When can the two parties use these parameter secrets? SSL mandates that the parties cannot use these parameters or secrets until they have sent or received a special message, the ChangeCipherSpec message, which is exchanged during the Handshake protocol and defined in the ChangeCipherSpec Protocol. The reason is that the issue is not just sending or receiving a message. The sender and the receiver need two states, not one. One state, the pending state, keeps track of the parameters and secrets. The other state, the active state, holds parameters and secrets used by the Record Protocol to sign/verify or encrypt/decrypt messages. In addition, each state holds two sets of values: *read* (inbound) and *write* (outbound).

The ChangeCipherSpec Protocol defines the process of moving values between the pending and active states. Figure 17.20 shows a hypothetical situation, with hypothetical

Figure 17.20 *Movement of parameters from pending state to active state*



values, to show the concept. Only a few parameters are shown. Before the exchange of any ChangeCipherSpec messages, only the pending columns have values.

First the client sends a ChangeCipherSpec message. After the client sends this message, it moves the write (outbound) parameters from pending to active. The client can now use these parameters to sign or encrypt outbound messages. After the receiver receives this message, it moves the read (inbound) parameters from the pending to the active state. Now the server can verify and decrypt messages. This means that the Finished message sent by the client can be signed and encrypted by the client and verified and decrypted by the server.

The server sends the ChangeCipherSpec message after receiving the Finish message from the client. After sending this message it moves the write (outbound) parameters from pending to active. The server can now use these parameters to sign or encrypt outbound messages. After the client receives this message, it moves the read (inbound) parameters from the pending to the active state. Now the client can verify and decrypt messages.

Of course, after the exchanged Finished messages, both parties can communicate in both directions using the read/write active parameters.

Alert Protocol

SSL uses the **Alert Protocol** for reporting errors and abnormal conditions. It has only one message type, the Alert message, that describes the problem and its level (warning or fatal). Table 17.4 shows the types of Alert messages defined for SSL.

Table 17.4 *Alerts defined for SSL*

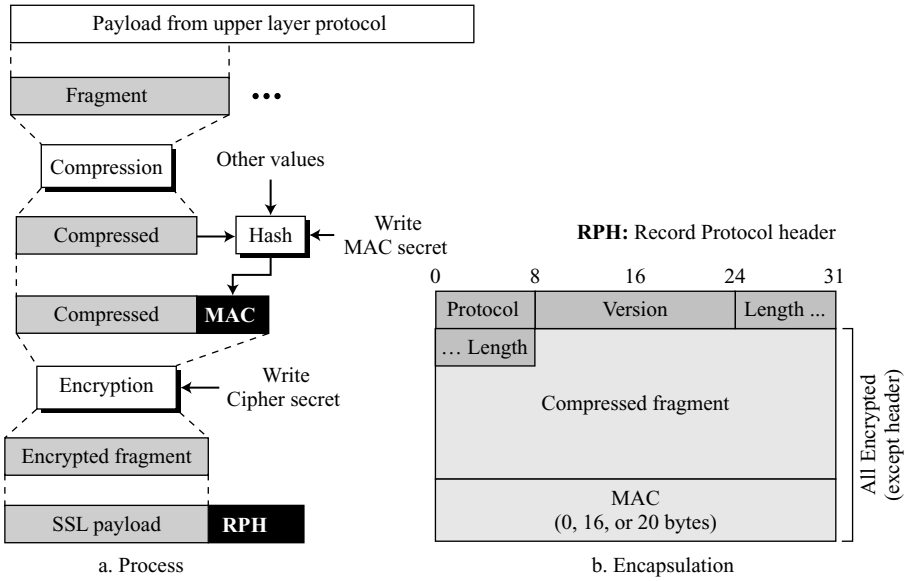
Value	*Description*	*Meaning*
0	*CloseNotify*	Sender will not send any more messages.
10	*UnexpectedMessage*	An inappropriate message received.
20	*BadRecordMAC*	An incorrect MAC received.
30	*DecompressionFailure*	Unable to decompress appropriately.
40	*HandshakeFailure*	Sender unable to finalize the handshake.
41	*NoCertificate*	Client has no certificate to send.
42	*BadCertificate*	Received certificate corrupted.
43	*UnsupportedCertificate*	Type of received certificate is not supported.
44	*CertificateRevoked*	Signer has revoked the certificate.
45	*CertificateExpired*	Certificate expired.
46	*CertificateUnknown*	Certificate unknown.
47	*IllegalParameter*	An out-of-range or inconsistent field.

Record Protocol

The **Record Protocol** carries messages from the upper layer (Handshake Protocol, ChangeCipherSpec Protocol, Alert Protocol, or application layer). The message is fragmented and optionally compressed; a MAC is added to the compressed message using

the negotiated hash algorithm. The compressed fragment and the MAC are encrypted using the negotiated encryption algorithm. Finally, the SSL header is added to the encrypted message. Figure 17.21 shows this process at the sender. The process at the receiver is reversed.

Figure 17.21 *Processing done by the Record Protocol*



Note, however, that this process can only be done when the cryptographic parameters are in the active state. Messages sent before the movement from pending to active are neither signed nor encrypted. However, in the next sections, we will see some messages in the Handshake Protocol that use some defined hash values for message integrity.

Fragmentation/Combination

At the sender, a message from the application layer is fragmented into blocks of 2^{14} bytes, with the last block possibly less than this size. At the receiver, the fragments are combined together to make a replica of the original message.

Compression/Decompression

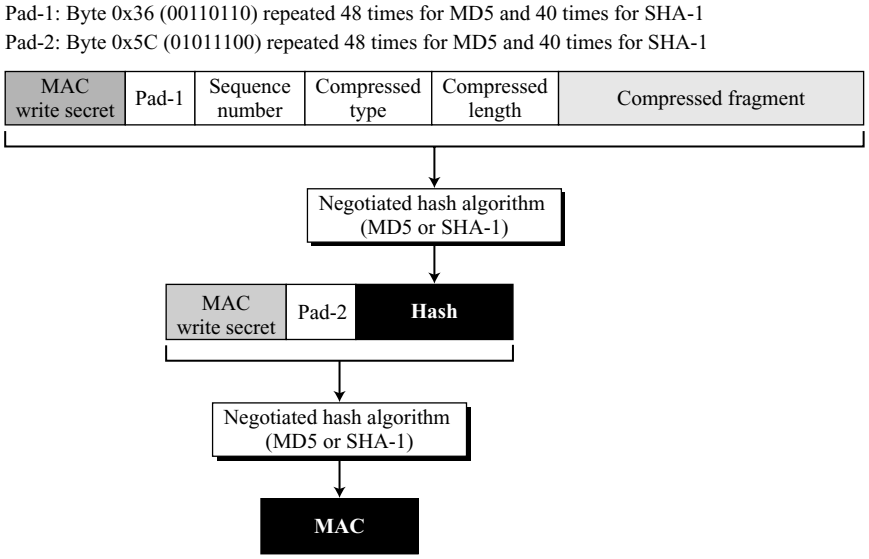
At the sender, all application layer fragments are compressed by the compression method negotiated during the handshaking. The compression method needs to be lossless (the decompressed fragment must be an exact replica of the original fragment). The size of the fragment must not exceed 1024 bytes. Some compression methods work

only on a predefined block size and if the size of the block is less than this, some padding is added. Therefore, the size of the compressed fragment may be greater than the size of the original fragment. At the receiver, the compressed fragment is decompressed to create a replica of the original. If the size of the decompressed fragment exceeds 2^{14}, a fatal decompression Alert message is issued. Note that compression/decompression is optional in SSL.

Signing/Verifying

At the sender, the authentication method defined during the handshake (NULL, MD5, or SHA-1) creates a signature (MAC), as shown in Figure 17.22.

Figure 17.22 *Calculation of MAC*



The hash algorithm is applied twice. First, a hash is created from the concatenations of the following values:

a. The MAC write secret (authentication key for the outbound message)
b. Pad-1, which is the byte 0x36 repeated 48 times for MD5 and 40 times for SHA-1
c. The sequence number for this message
d. The compressed type, which defines the upper-layer protocol that provided the compressed fragment
e. The compressed length, which is the length of the compressed fragment
f. The compressed fragment itself

Second, the final hash (MAC) is created from the concatenation of the following values:

a. The MAC write secret

b. Pad-2, which is the byte 0x5C repeated 48 times for MD5 and 40 times for SHA-1
c. The hash created from the first step

At the receiver, the verifying is done by calculating a new hash and comparing it to the received hash.

Encryption/Decryption

At the sender, the compressed fragment and the hash are encrypted using the cipher write secret. At the receiver, the received message is decrypted using the cipher read secret. For block encryption, padding is added to make the size of the encryptable message a multiple of the block size.

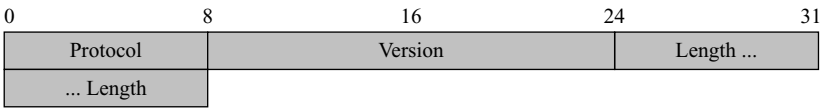
Framing/Deframing

After the encryption, the Record Protocol header is added at the sender. The header is removed at the receiver before decryption.

17.3 SSL MESSAGE FORMATS

As we have discussed, messages from three protocols and data from the application layer are encapsulated in the Record Protocol messages. In other words, the Record Protocol message encapsulates messages from four different sources at the sender site. At the receiver site, the Record Protocol decapsulates the messages and delivers them to different destinations. The Record Protocol has a general header that is added to each message coming from the sources, as shown in Figure 17.23.

Figure 17.23 *Record Protocol general header*



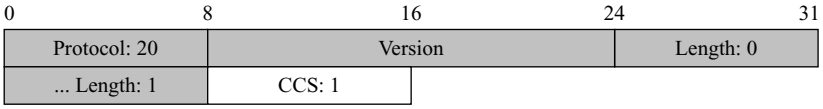
The fields in this header are listed below.

- **Protocol.** This 1-byte field defines the source or destination of the encapsulated message. It is used for multiplexing and demultiplexing. The values are 20 (ChangeCipherSpec Protocol), 21 (Alert Protocol), 22 (Handshake Protocol), and 23 (data from the application layer).
- **Version.** This 2-byte field defines the version of the SSL; one byte is the major version and the other is the minor. The current version of SSL is 3.0 (major 3 and minor 0).
- **Length.** This 2-byte field defines the size of the message (without the header) in bytes.

ChangeCipherSpec Protocol

As we said before, the ChangeCipherSpec Protocol has one message, the ChangeCipherSpec message. The message is only one byte, encapsulated in the Record Protocol message with protocol value 20, as shown in Figure 17.24.

Figure 17.24 *ChangeCipherSpec message*

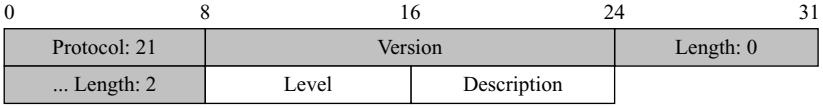


The one-byte field in the message is called the CCS and its value is currently 1.

Alert Protocol

The Alert Protocol, as we discussed before, has one message that reports errors in the process. Figure 17.25 shows the encapsulation of this single message in the Record Protocol with protocol value 21.

Figure 17.25 *Alert message*



The two fields of the Alert message are listed below.

- **Level.** This one-byte field defines the level of the error. Two levels have been defined so far: warning and fatal.
- **Description.** The one-byte description defines the type of error.

Handshake Protocol

Several messages have been defined for the Handshake Protocol. All of these messages have the four-byte generic header shown in Figure 17.26. The figure shows the Record Protocol header and the generic header for the Handshake Protocol. Note that the value of the protocol field is 22.

- **Type.** This one-byte field defines the type of message. So far ten types have been defined as listed in Table 17.5.

Figure 17.26 *Generic header for Handshake Protocol*

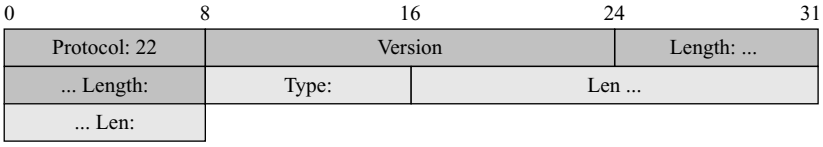


Table 17.5 *Types of Handshake messages*

Type	*Message*
0	HelloRequest
1	ClientHello
2	ServerHello
11	Certificate
12	ServerKeyExchange
13	CertificateRequest
14	ServerHelloDone
15	CertificateVerify
16	ClientKeyExchange
20	Finished

- **Length (Len).** This three-byte field defines the length of the message (excluding the length of the type and length field). The reader may wonder why we need two length fields, one in the general Record header and one in the generic header for the Handshake messages. The answer is that a Record message may carry two Handshake messages at the same time if there is no need for another message in between.

HelloRequest Message

The HelloRequest message, which is rarely used, is a request from the server to the client to restart a session. This may be needed if the server feels that something is wrong with the session and a fresh session is needed. For example, if the session becomes so long that it threatens the security of the session, the server may send this message. The client then needs to send a ClientHello message and negotiate the security parameters. Figure 17.27 shows the format of this message. It is four bytes with a type value of 0. The message has no body, so the value of the length field is also 0.

ClientHello Message

The ClientHello message is the first message exchanged during handshaking. Figure 17.28 shows the format of the message.

Figure 17.27 *HelloRequest message*

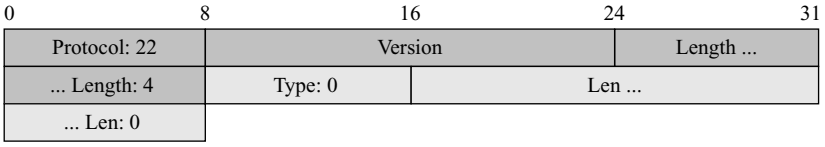
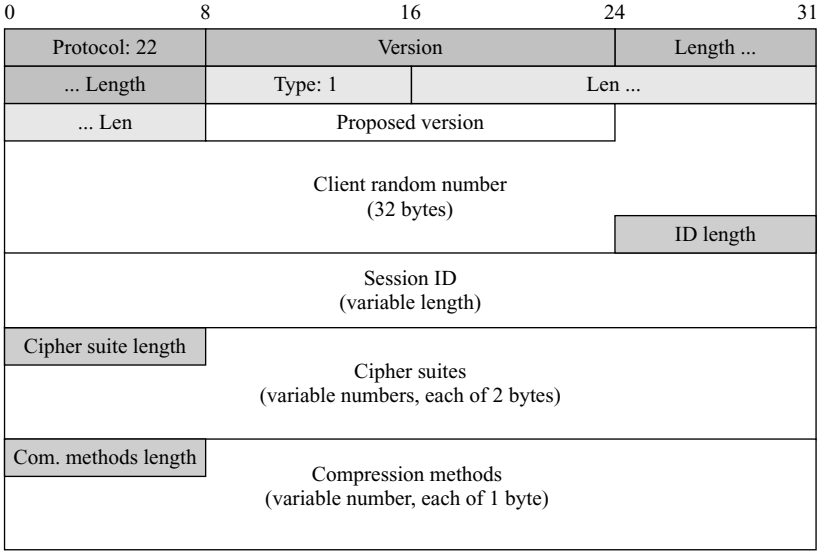


Figure 17.28 *ClientHello message*



The type and length fields are as discussed previously. The following is a brief description of the other fields.

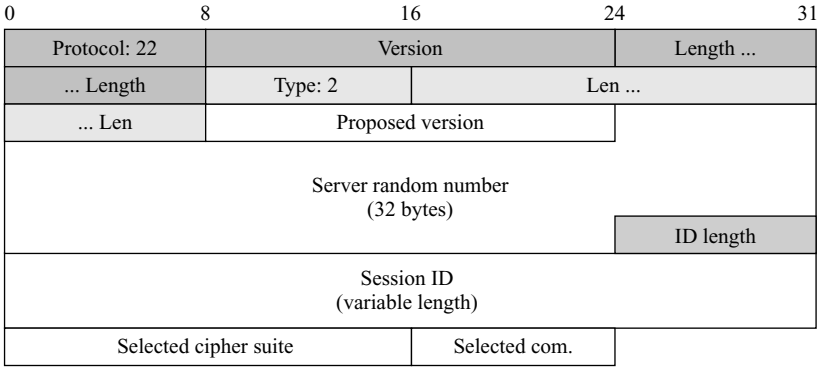
- **Version.** This 2-byte field shows the version of the SSL used. The version is 3.0 for SSL and 3.1 for TLS. Note that the version value, for example, 3.0, is stored in two bytes: 3 in the first byte and 0 in the second.
- **Client Random Number.** This 32-byte field is used by the client to send the client random number, which creates security parameters.
- **Session ID Length.** This 1-byte field defines the length of the session ID (next field). If there is no session ID, the value of this field is 0.
- **Session ID.** The value of this variable-length field is 0 when the client starts a new session. The session ID is initiated by the server. However, if a client wants to resume a previously stopped session, it can include the previously-defined session ID in this field. The protocol defines a maximum of 32 bytes for the session ID.

- **Cipher Suite Length.** This 2-byte field defines the length of the client-proposed cipher suite list (next field).
- **Cipher Suite List.** This variable-length field gives the list of cipher suites that the client supports. The field lists the cipher suites from the most preferred to the least preferred. Each cipher suite is encoded as a two-byte number.
- **Compression Methods Length.** This 1-byte field defines the length of client-proposed compression methods (next field).
- **Compression Method List.** This variable-length field gives the list of compression methods that the client supports. The field lists the methods from the most preferred to the least preferred. Each method is encoded as a one-byte number. So far, the only method is the *NULL* method (no compression). In this case, the value of the compression method length is 1 and the compression method list has only one element with the value of 0.

ServerHello Message

The ServerHello message is the server response to the ClientHello message. The format is similar to the ClientHello message, but with fewer fields. Figure 17.29 shows the format of the message.

Figure 17.29 *ServerHello message*



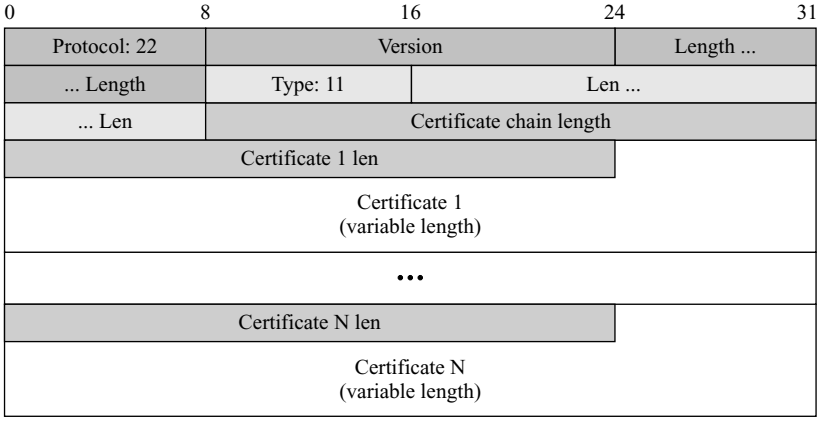
The version field is the same. The server random number field defines a value selected by the server. The session ID length and the session ID field are the same as those in the ClientHello message. However, the session ID is usually blank (and the length is usually set to 0) unless the server is resuming an old session. In other words, if the server allows a session to resume, it inserts a value in the session ID field to be used by the client (in the ClientHello message) if the client wishes to reopen an old session.

The selected cipher suite field defines the single cipher suite selected by the server from the list sent by the client. The compression method field defines the method selected by the server from the list sent by the client.

Certificate Message

The Certificate message can be sent by the client or the server to list the chain of public-key certificates. Figure 17.30 shows the format.

Figure 17.30 *Certificate message*



The value of the type field is 11. The body of the message includes the following fields:

- **Certificate Chain Length.** This three-byte field shows the length of the certificate chain. This field is redundant because its value is always 3 less than the value of the length field.
- **Certificate Chain.** This variable-length field lists the chain of public-key certificates that the client or the server carries. For each certificate, there are two sub-fields:
  a. A three-byte length field
  b. The variable-size certificate itself

ServerKeyExchange Message

The ServerKeyExchange message is sent from the server to the client. Figure 17.31 shows the general format.

The message contains the keys generated by the server. The format of the message is dependent on the cipher suite selected in the previous message. The client that receives the message needs to interpret the message according to the previous information. If the server has sent a certificate message, then the message also contains a signed parameter.

CertificateRequest Message

The CertificateRequest message is sent from the server to the client. The message asks the client to authenticate itself to the server using one of the acceptable certificates and one of the certificate authorities named in the message. Figure 17.32 shows the format.

Figure 17.31 *ServerKeyExchange message*

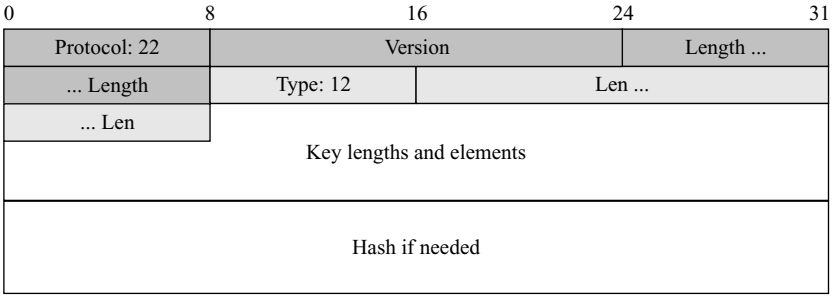
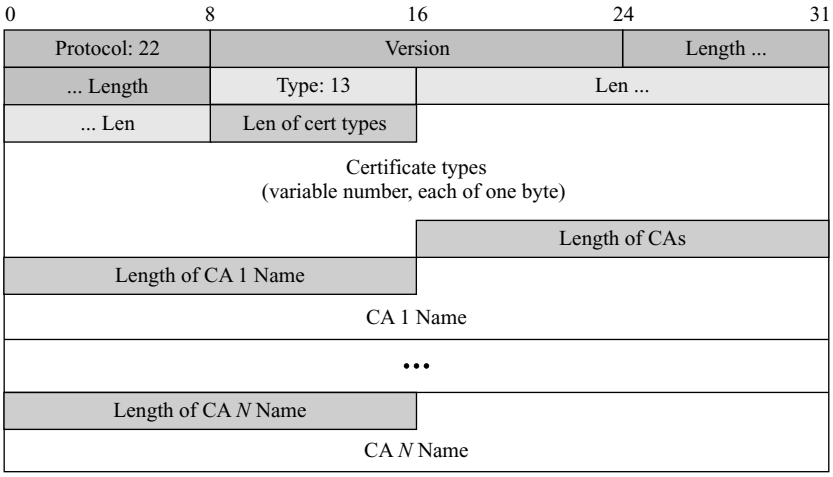


Figure 17.32 *CertificateRequest message*



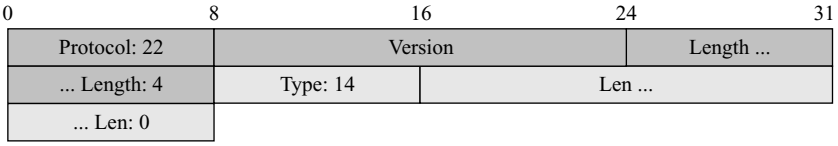
The value of the type field is 13. The body of the message includes the following fields:

- **Len of Cert Types.** This one-byte field shows the length of the certificate types.
- **Certificates Types.** This variable-length field gives the list of the public-key certificate types that the server accepts. Each type is one byte.
- **Length of CAs.** This two-byte field gives the length of the certificate authorities (the rest of the packet).
- **Length of CA *x* Name.** This two-byte field defines the length of the *x*th certificate authority name. The value of *x* can be between 1 to *N*.
- **CA *x* Name.** This variable-length field defines the name of the *x*th certificate authority. The value of *x* can be between 1 to *N*.

ServerHelloDone Message

The ServerHelloDone message is the last message sent in the second phase of handshaking. The message signals that phase II does not carry any extra information. Figure 17.33 shows the format.

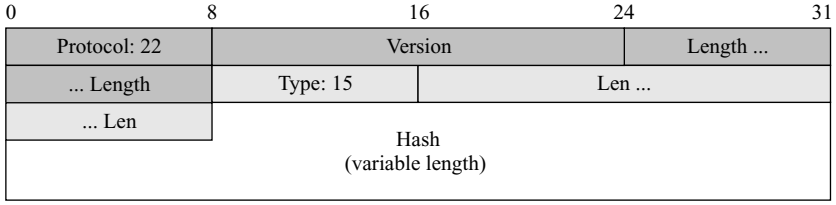
Figure 17.33 *ServerHelloDone message*



CertificateVerify Message

The CertificateVerify message is the last message of Phase III. In this message, the client proves that it actually owns the private key related to its public-key certificate. To do so, the client creates a hash of all handshake messages sent before this message, and signs them using the MD5 or SHA-1 algorithm based on the certificate type of the client. Figure 17.34 shows the format.

Figure 17.34 *CertificateVerify message*



If the client private key is related to a DSS certificate, then the hash is based only on the SHA-1 algorithm and the length of the hash is 20 bytes. If the client private key is related to an RSA certificate, then there are two hashes (concatenated), one based on MD5 and the other based on SHA-1. The total length is 16 + 20 = 36 bytes. Figure 17.35 shows the hash calculation.

ClientKeyExchange Message

The ClientKeyExchange is the second message sent during the third phase of handshaking. In this message, the client provides the keys. The format of the message depends on the specific key exchange algorithms selected by two parties. Figure 17.36 shows the general idea.

Figure 17.35 *Hash calculation for CertificateVerify message*

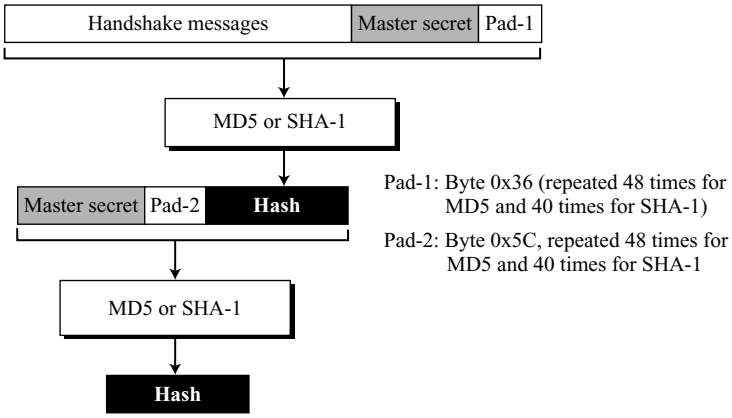
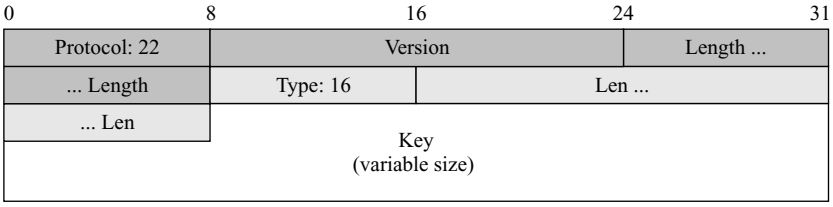


Figure 17.36 *ClientKeyExchange message*



Finished Message

The Finished message shows that the negotiation is over. It contains all of the messages exchanged during handshaking, followed by the sender role, the master secret, and the padding. The exact format depends on the type of cipher suite used. The general format is shown in Figure 17.37.

Figure 17.37 shows that there is a concatenation of two hashes in the message. Figure 17.38 shows how each is calculated.

Note that when the client or server sends the Finished message, it has already sent the ChangeCipherSpec message. In other words, the write cryptographic secrets are in the active state. The client or the server can treat the Finished message like a data fragment coming from the application layer. The Finished message can be authenticated (using the MAC in the cipher suite) and encrypted (using the encryption algorithm in the cipher suite).

Application Data

The Record Protocol adds a signature (MAC) at the end of the (possibly compressed) fragment coming from the application layer and then encrypts the fragment and the

Figure 17.37 *Finished message*

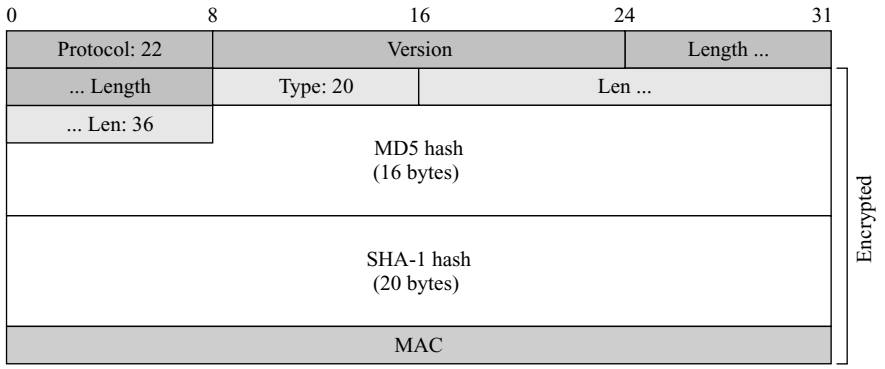
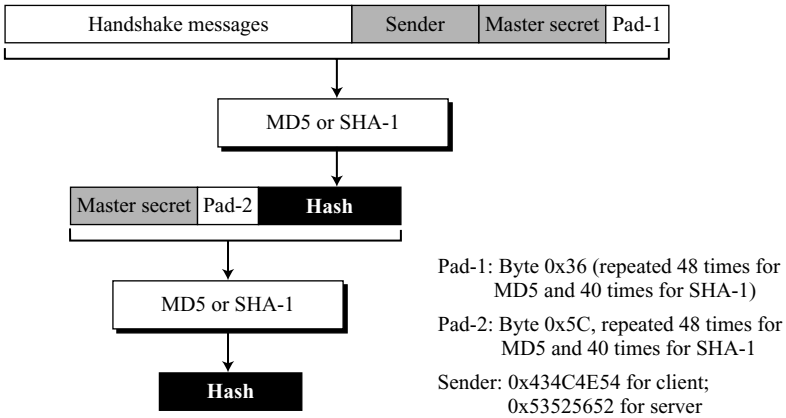


Figure 17.38 *Hash calculation for Finished message*

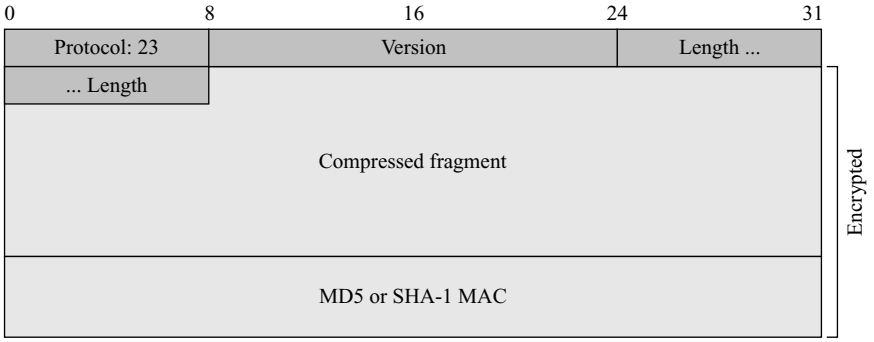


MAC. After adding the general header with protocol value 23, the Record message is transmitted. Note that the general header is not encrypted. Figure 17.39 shows the format.

17.4 TRANSPORT LAYER SECURITY

The Transport Layer Security (TLS) protocol is the IETF standard version of the SSL protocol. The two are very similar, with slight differences. Instead of describing TLS in full, we highlight the differences between TLS and SSL protocols in this section.

Figure 17.39 *Record Protocol message for application data*



Version

The first difference is the version number (major and minor). The current version of SSL is 3.0; the current version of TLS is 1.0. In other words, SSLv3.0 is compatible with TLSv1.0.

Cipher Suite

Another minor difference between SSL and TLS is the lack of support for the Fortezza method. TLS does not support Fortezza for key exchange or for encryption/decryption. Table 17.6 shows the cipher suite list for TLS (without export entries).

Generation of Cryptographic Secrets

The generation of cryptographic secrets is more complex in TLS than in SSL. TLS first defines two functions: the data-expansion function and the pseudorandom function. Let us discuss these two functions.

Data-Expansion Function

The **data-expansion function** uses a predefined HMAC (either MD5 or SHA-1) to expand a secret into a longer one. This function can be considered a multiple-section function, where each section creates one hash value. The extended secret is the concatenation of the hash values. Each section uses two HMACs, a secret and a seed. The data-expansion function is the chaining of as many sections as required. However, to make the next section dependent on the previous, the second seed is actually the output of the first HMAC of the previous section as shown in Figure 17.40.

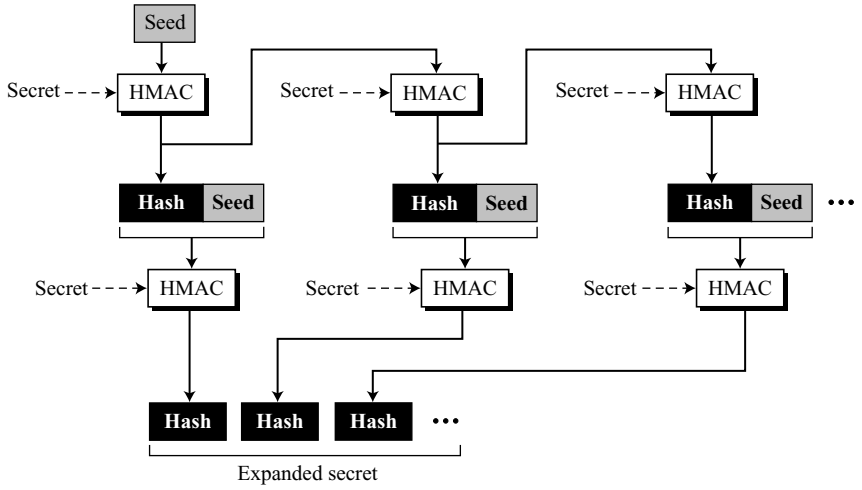
Pseudorandom Function (PRF)

TLS defines a **pseudorandom function** (**PRF**) to be the combination of two data-expansion functions, one using MD5 and the other SHA-1. PRF takes three inputs, a secret, a

Table 17.6 *Cipher Suite for TLS*

Cipher suite	*Key Exchange*	*Encryption*	*Hash*
TLS_NULL_WITH_NULL_NULL	NULL	NULL	NULL
TLS_RSA_WITH_NULL_MD5	RSA	NULL	MD5
TLS_RSA_WITH_NULL_SHA	RSA	NULL	SHA-1
TLS_RSA_WITH_RC4_128_MD5	RSA	RC4	MD5
TLS_RSA_WITH_RC4_128_SHA	RSA	RC4	SHA-1
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	IDEA	SHA-1
TLS_RSA_WITH_DES_CBC_SHA	RSA	DES	SHA-1
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES	SHA-1
TLS_DH_anon_WITH_RC4_128_MD5	DH_anon	RC4	MD5
TLS_DH_anon_WITH_DES_CBC_SHA	DH_anon	DES	SHA-1
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	DH_anon	3DES	SHA-1
TLS_DHE_RSA_WITH_DES_CBC_SHA	DHE_RSA	DES	SHA-1
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES	SHA-1
TLS_DHE_DSS_WITH_DES_CBC_SHA	DHE_DSS	DES	SHA-1
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES	SHA-1
TLS_DH_RSA_WITH_DES_CBC_SHA	DH_RSA	DES	SHA-1
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES	SHA-1
TLS_DH_DSS_WITH_DES_CBC_SHA	DH_DSS	DES	SHA-1
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH_DSS	3DES	SHA-1

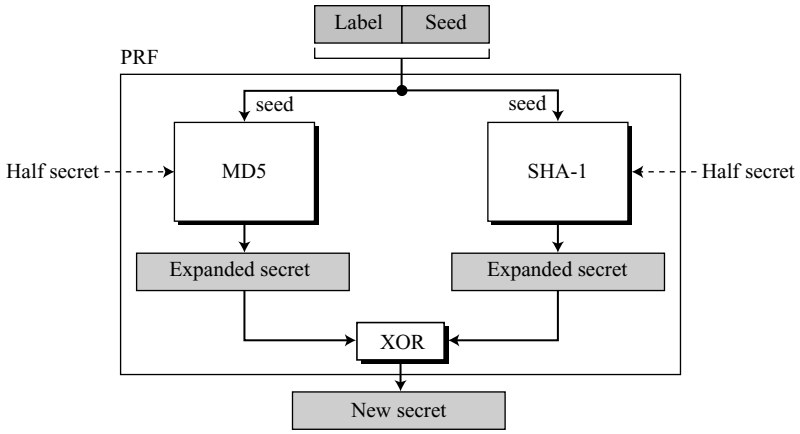
Figure 17.40 *Data-expansion function*



label, and a seed. The label and seed are concatenated and serve as the seed for each data-expansion function. The secret is divided into two halves; each half is used as the secret for each data-expansion function. The output of two data-expansion functions is exclusive-ored together to create the final expanded secret. Note that because the hashes created from

MD5 and SHA-1 are of different sizes, extra sections of MD5-based functions must be created to make the two outputs the same size. Figure 17.41 shows the idea of PRF.

Figure 17.41 *PRF*



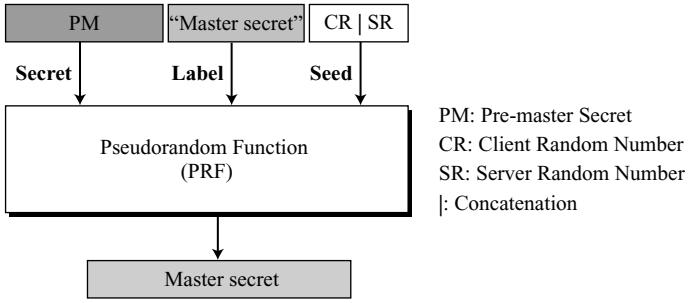
Pre-master Secret

The generation of the pre-master secret in TLS is exactly the same as in SSL.

Master Secret

TLS uses the PRF function to create the master secret from the pre-master secret. This is achieved by using the pre-master secret as the secret, the string "master secret" as the label, and concatenation of the client random number and server random number as the seed. Note that the label is actually the ASCII code of the string "master secret". In other words, the label defines the output we want to create, the master secret. Figure 17.42 shows the idea.

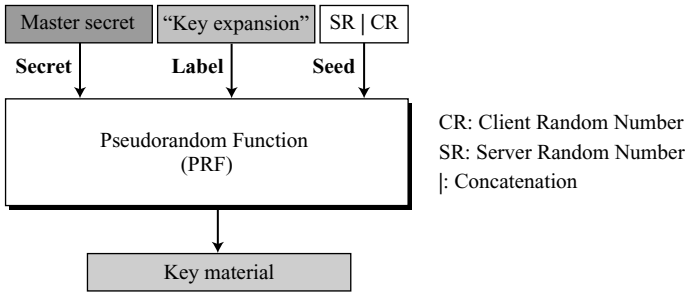
Figure 17.42 *Master secret generation*



Key Material

TLS uses the PRF function to create the key material from the master secret. This time the secret is the master secret, the label is the string "key expansion", and the seed is the concatenation of the server random number and the client random number, as shown in Figure 17.43.

Figure 17.43 *Key material generation*



Alert Protocol

TLS supports all of the alerts defined in SSL except for *NoCertificate*. TLS also adds some new ones to the list. Table 17.7 shows the full list of alerts supported by TLS.

Table 17.7 *Alerts defined for TLS*

Value	*Description*	*Meaning*
0	*CloseNotify*	Sender will not send any more messages.
10	*UnexpectedMessage*	An inappropriate message received.
20	*BadRecordMAC*	An incorrect MAC received.
21	*DecryptionFailed*	Decrypted message is invalid.
22	*RecordOverflow*	Message size is more than $2^{14} + 2048$.
30	*DecompressionFailure*	Unable to decompress appropriately.
40	*HandshakeFailure*	Sender unable to finalize the handshake.
42	*BadCertificate*	Received certificate corrupted.
43	*UnsupportedCertificate*	Type of received certificate is not supported.
44	*CertificateRevoked*	Signer has revoked the certificate.
45	*CertificateExpired*	Certificate has expired.
46	*CertificateUnknown*	Certificate unknown.
47	*IllegalParameter*	A field out of range or inconsistent with others.
48	*UnknownCA*	CA could not be identified.

Table 17.7 *Alerts defined for TLS (continued)*

Value	*Description*	*Meaning*
49	*AccessDenied*	No desire to continue with negotiation.
50	*DecodeError*	Received message could not be decoded.
51	*DecryptError*	Decrypted ciphertext is invalid.
60	*ExportRestriction*	Problem with U.S. restriction compliance.
70	*ProtocolVersion*	The protocol version is not supported.
71	*InsufficientSecurity*	More secure cipher suite needed.
80	*InternalError*	Local error.
90	*UserCanceled*	The party wishes to cancel the negotiation.
100	*NoRenegotiation*	The server cannot renegotiate the handshake.

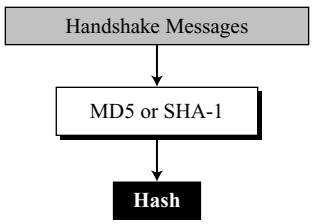
Handshake Protocol

TLS has made some changes in the Handshake Protocol. Specifically, the details of the CertificateVerify message and the Finished message have been changed.

CertificateVerify Message

In SSL, the hash used in the CertificateVerify message is the two-step hash of the handshake messages plus a pad and the master secret. TLS has simplified the process. The hash in the TLS is only over the handshake messages, as shown in Figure 17.44.

Figure 17.44 *Hash for CertificateVerify message in TLS*



Finished Message

The calculation of the hash for the Finished message has also been changed. TLS uses the PRF to calculate two hashes used for the Finished message, as shown in Figure 17.45.

Record Protocol

The only change in the Record Protocol is the use of HMAC for signing the message. TLS uses the MAC, as defined in Chapter 11, to create the HMAC. TLS also adds the protocol version (called Compressed version) to the text to be signed. Figure 17.46 shows how the HMAC is formed.

Figure 17.45 *Hash for Finished message in TLS*

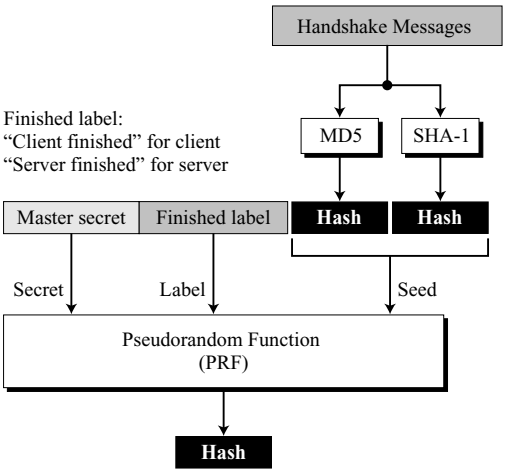
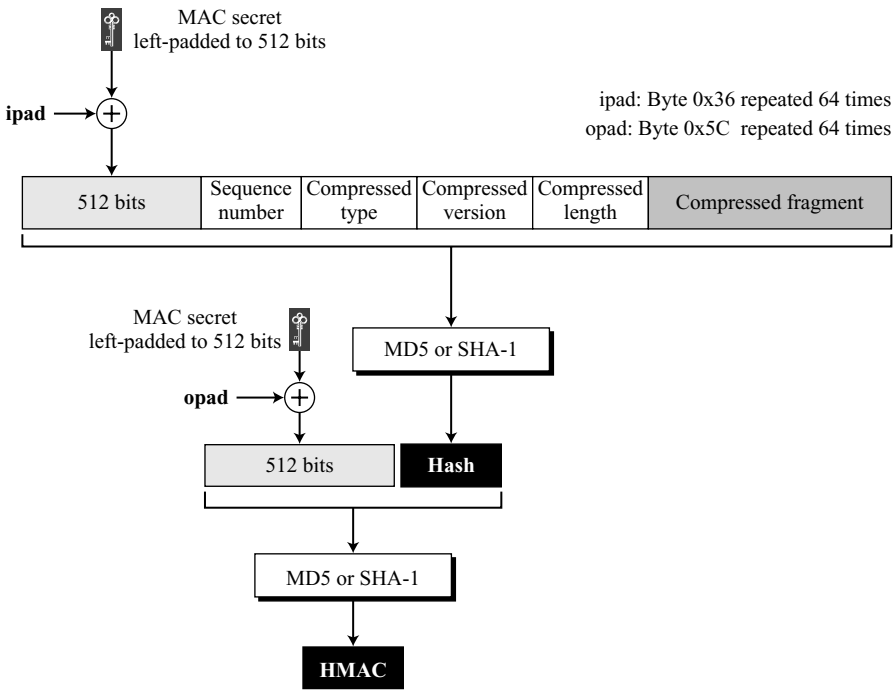


Figure 17.46 *HMAC for TLS*



17.5 RECOMMENDED READING

The following books and websites give more details about subjects discussed in this chapter. The items enclosed in brackets refer to the reference list at the end of the book.

Books

[Res01], [Tho00], [Sta06], [Rhe03], and [PHS03] discuss SSL and TLS.

WebSites

The following website give more information about topics discussed in this chapter.

http://www.ietf.org/rfc/rfc2246.txt

17.6 KEY TERMS

Alert Protocol
anonymous Diffie-Hellman
ChangeCipherSpec Protocol
cipher suite
connection
data-expansion function
ephemeral Diffie-Hellman
fixed Diffie-Hellman
Fortezza
Handshake Protocol
Hypertext Transfer Protocol (HTTP)
key material
master secret
pre-master secret
pseudorandom function (PRF)
Record Protocol
Secure Sockets Layer (SSL) Protocol
session
Transport Layer Security (TLS) Protocol

17.7 SUMMARY

- A transport layer security protocol provides end-to-end security services for applications that use the services of a reliable transport layer protocol such as TCP. Two protocols are dominant today for providing security at the transport layer: Secure Sockets Layer (SSL) and Transport Layer Security (TLS).
- SSL (or TLS) provides services such as fragmentation, compression, message integrity, confidentiality, and framing on data received from the application layer. Typically, SSL (or TLS) can receive application data from any application layer protocol, but the protocol is normally HTTP.
- The combination of key exchange, hash, and encryption algorithm defines a cipher suite for each session. The name of each suite is descriptive of the combination.

- To exchange authenticated and confidential messages, the client and the server each need six cryptographic secrets (four keys and two initialization vectors).
- SSL (or TLS) makes a distinction between a connection and a session. In a session, one party has the role of a client and the other the role of a server; in a connection, both parties have equal roles, they are peers.
- SSL (or TLS) defines four protocols in two layers: the Handshake Protocol, the ChangeCipherSpec Protocol, the Alert Protocol, and the Record Protocol. The Handshake Protocol uses several messages to negotiate cipher suite, to authenticate the server for the client and the client for the server if needed, and to exchange information for building the cryptographic secrets. The ChangeCipherSpec protocol defines the process of moving values between the pending and active states. The Alert Protocol reports errors and abnormal conditions. The Record Protocol carries messages from the upper layer (Handshake Protocol, Alert Protocol, ChangeCipherSpec Protocol, or application layer).

17.8 PRACTICE SET

Review Questions

1. List services provided by SSL or TLS.
2. Describe how master secret is created from pre-master secret in SSL.
3. Describe how master secret is created from pre-master secret in TLS.
4. Describe how key materials are created from master secret in SSL.
5. Describe how key materials are created from master secret in TLS.
6. Distinguish between a session and a connection.
7. List and give the purpose of four protocols defined in SSL or TLS.
8. Define the goal of each phase in the Handshake protocol.
9. Compare and contrast the Handshake protocols in SSL and TLS.
10. Compare and contrast the Record protocols in SSL and TLS.

Exercises

11. What is the length of the key material if the cipher suite is one of the following:
    a. SSL_RSA_WITH_NULL_MD5
    b. SSL_RSA_WITH_NULL_SHA
    c. TLS_RSA_WITH_DES_CBC_SHA
    d. TLS_RSA_WITH_3DES_EDE_CBC_SHA
    e. TLS_DHE_RSA_WITH_DES_CBC_SHA
    f. TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA

12. Show the number of repeated modules needed for each case in Exercise 11 (see Figure 17.9).
13. Compare the calculation of the master secret in SSL with that in TLS. In SSL, the pre-master is included three times in the calculation, in TLS only once. Which calculation is more efficient in terms of space and time?
14. Compare the calculation of the key material in SSL and TLS. Answer the following questions:
    a. Which calculation provides more security?
    b. Which calculation is more efficient in terms of space and time?
15. The calculation of key material in SSL requires several iterations, the one for TLS does not. How can TLS calculate key material of variable length?
16. When a session is resumed with a new connection, SSL does not require the full handshaking process. Show the messages that need to be exchanged in a partial handshaking.
17. When a session is resumed, which of the following cryptographic secrets need to be recalculated?
    a. Pre-master secret
    b. Master secret
    c. Authentication keys
    d. Encryption keys
    e. IVs
18. In Figure 17.20, what happens if the server sends the ChangeCipherSpec message, but the client does not? Which messages in the Handshake Protocol can follow? Which cannot?
19. Compare the calculation of MAC in SSL and TLS (see Figure 17.22 and Figure 17.46). Which one is more efficient?
20. Compare the calculation of the hash for CertificateVerify messages in SSL and TLS (see Figure 17.35 and Figure 17.44). Which one is more efficient?
21. Compare the calculation of the hash for Finished messages in SSL and TLS (see – Figure 17.38 and Figure 17.45). Answer the following questions:
    a. Which one is more secure?
    b. Which one is more efficient?
22. TLS uses PRF for all hash calculations except for CertificateVerify message. Give a reason for this exception.
23. Most protocols have a formula to show the calculations of cryptographic secrets and hashes. For example, in SSL, the calculation of the master secret (see Figure 17.8) is as follows (concatenation is designated by a bar):

$$\begin{aligned}\text{Master Secret} = {} & \text{MD5 (pre-master} \mid \text{SHA-1 (“A” / pre-master / CR / SR))} \mid \\ & \text{MD5 (pre-master} \mid \text{SHA-1 (“A” } \mid \text{ pre-master } \mid \text{ CR } \mid \text{ SR))} \mid \\ & \text{MD5 (pre-master / SHA-1 (“A” / pre-master} \mid \text{CR} \mid \text{SR))}\end{aligned}$$

Show the formula for the following:

a. Key material in SSL (Figure 17.9)
b. MAC in SSL (Figure 17.22)
c. Hash calculation for CertificateVerify message in SSL (Figure 17.35)
d. Hash calculation for Finished message in SSL (Figure 17.38)
e. Data expansion in TLS (Figure 17.40)
f. PRF in TLS (Figure 17.41)
g. Master secret in TLS (Figure 17.42)
h. Key material in TLS (Figure 17.43)
i. Hash calculation for CertificateVerify message in TLS (Figure 17.44)
j. Hash calculation for Finished message in TLS (Figure 17.45)
k. MAC in TLS (Figure 17.46)

24. Show how SSL or TLS reacts to a replay attack. That is, show how SSL or TLS responds to an attacker that tries to replay one or more handshake messages.
25. Show how SSL or TLS reacts to a brute-force attack. Can an intruder use an exhaustive computer search to find the encryption key in SSL or TLS? Which protocol is more secure in this respect, SSL or TLS?
26. What is the risk of using short-length keys in SSL or TLS? What type of attack can an intruder try if the keys are short?
27. Is SSL or TLS more secure to a man-in-the-middle attack? Can an intruder create key material between the client and herself and between the server and herself?