CHAPTER 15

Key Management

Objectives

This chapter has several objectives:

- ❑ To explain the need for a key-distribution center (KDC)
- ❑ To show how a KDC can create a session key between two parties
- ❑ To show how two parties can use a symmetric-key agreement protocol to create a session key between themselves without using the services of a KDC
- ❑ To describe Kerberos as a KDC and an authentication protocol
- ❑ To explain the need for certification authorities (CAs) for public keys and how X.509 recommendation defines the format of certificates
- ❑ To introduce the idea of a Public-Key Infrastructure (PKI) and explain some of its duties

Previous chapters have discussed symmetric-key and asymmetric-key cryptography. However, we have not yet discussed how secret keys in symmetric-key cryptography, and public keys in asymmetric-key cryptography, are distributed and maintained. This chapter touches on these two issues.

We first discuss the distribution of symmetric keys using a trusted third party. Second, we show how two parties can establish a symmetric key between themselves without using a trusted third party. Third, we introduce Kerberos as both a KDC and an authentication protocol. Fourth, we discuss the certification of public keys using certification authorities (CAs) based on the X.509 recommendation. Finally, we briefly discuss the idea of a Public-Key Infrastructure (PKI) and mention some of its duties.

15.1 SYMMETRIC-KEY DISTRIBUTION

Symmetric-key cryptography is more efficient than asymmetric-key cryptography for enciphering large messages. Symmetric-key cryptography, however, needs a shared secret key between two parties.

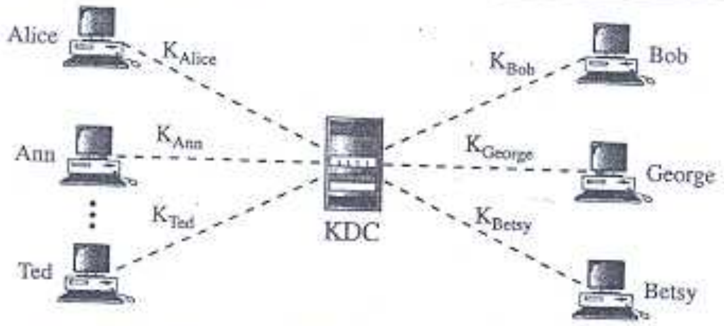
If Alice needs to exchange confidential messages with N people, she needs N different keys. What if N people need to communicate with each other? A total of $N(N-1)$ keys is needed if we require that Alice and Bob use two keys for bidirectional communication; only $N(N-1)/2$ keys are needed if we allow a key to be used for both directions. This means that if one million people need to communicate with each other, each person has almost one million different keys; in total, almost one trillion keys are needed. This is normally referred to as the N^2 problem because the number of required keys for N entities is N^2.

The number of keys is not the only problem; the distribution of keys is another. If Alice and Bob want to communicate, they need a way to exchange a secret key; if Alice wants to communicate with one million people, how can she exchange one million keys with one million people? Using the Internet is definitely not a secure method. It is obvious that we need an efficient way to maintain and distribute secret keys.

Key-Distribution Center: KDC

A practical solution is the use of a trusted third party, referred to as a **key-distribution center (KDC)**. To reduce the number of keys, each person establishes a shared secret key with the KDC, as shown in Figure 15.1.

Figure 15.1 *Key-distribution center (KDC)*



A secret key is established between the KDC and each member. Alice has a secret key with the KDC, which we refer to as K_{Alice}; Bob has a secret key with the KDC, which we refer to as K_{Bob}; and so on. Now the question is how Alice can send a confidential message to Bob. The process is as follows:

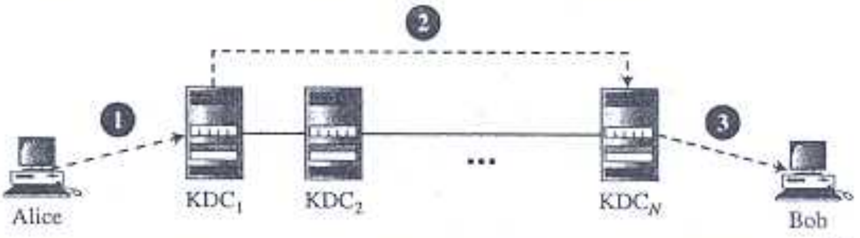
1. Alice sends a request to the KDC stating that she needs a session (temporary) secret key between herself and Bob.
2. The KDC informs Bob about Alice's request.
3. If Bob agrees, a session key is created between the two.

The secret key between Alice and Bob that is established with the KDC is used to authenticate Alice and Bob to the KDC and to prevent Eve from impersonating either of them. We discuss how a session key is established between Alice and Bob later in the chapter.

Flat Multiple KDCs

When the number of people using a KDC increases, the system becomes unmanageable and a bottleneck can result. To solve the problem, we need to have multiple KDCs. We can divide the world into domains. Each domain can have one or more KDCs (for redundancy in case of failure). Now if Alice wants to send a confidential message to Bob, who belongs to another domain, Alice contacts her KDC, which in turn contacts the KDC in Bob's domain. The two KDCs can create a secret key between Alice and Bob. Figure 15.2 shows KDCs all at the same level. We call this flat multiple KDCs.

Figure 15.2 *Flat multiple KDCs*



Hierarchical Multiple KDCs

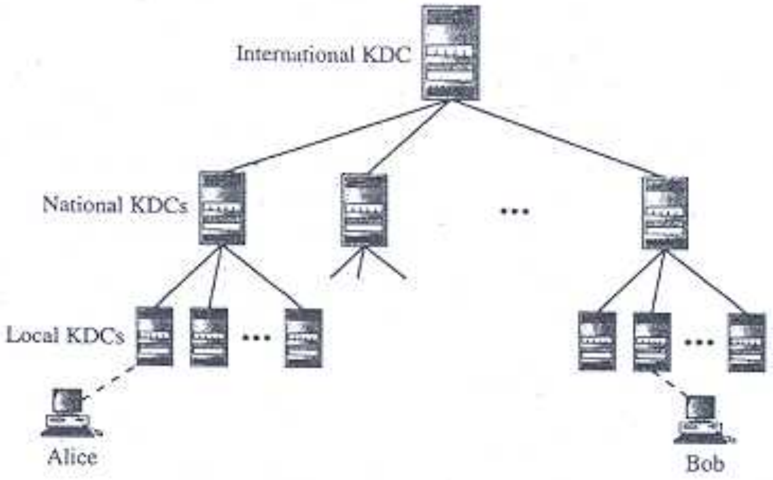
The concept of flat multiple KDCs can be extended to a hierarchical system of KDCs, with one or more KDCs at the top of the hierarchy. For example, there can be local KDCs, national KDCs, and international KDCs. When Alice needs to communicate with Bob, who lives in another country, she sends her request to a local KDC; the local KDC relays the request to the national KDC; the national KDC relays the request to an international KDC. The request is then relayed all the way down to the local KDC where Bob lives. Figure 15.3 shows a configuration of hierarchical multiple KDCs.

Session Keys

A KDC creates a secret key for each member. This secret key can be used only between the member and the KDC, not between two members. If Alice needs to communicate secretly with Bob, she needs a secret key between herself and Bob. A KDC can create a **session key** between Alice and Bob, using their keys with the center. The keys of Alice and Bob are used to authenticate Alice and Bob to the center and to each other before the session key is established. After communication is terminated, the session key is no longer useful.

A session symmetric key between two parties is used only once.

Figure 15.3 *Hierarchical multiple KDCs*

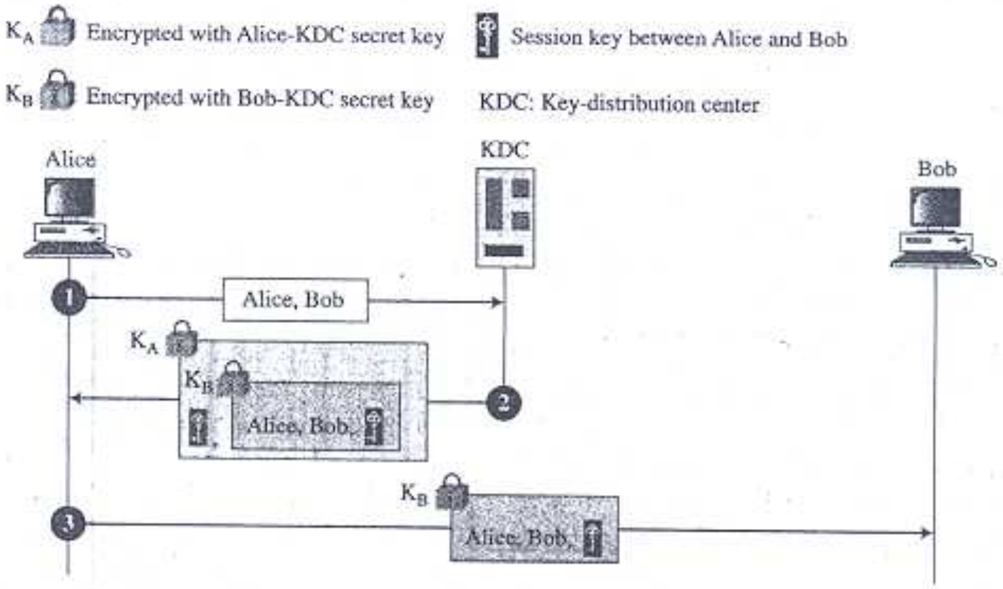


Several different approaches have been proposed to create the session key using ideas discussed in Chapter 14 for entity authentication.

A Simple Protocol Using a KDC

Let us see how a KDC can create a session key K_{AB} between Alice and Bob. Figure 15.4 shows the steps.

Figure 15.4 *First approach using KDC*


K_A Encrypted with Alice-KDC secret key
K_B Encrypted with Bob-KDC secret key
Session key between Alice and Bob
KDC: Key-distribution center
Alice
KDC
Bob
1
Alice, Bob
K_A
K_B
Alice, Bob
2
K_B
3
Alice, Bob

1. Alice sends a plaintext message to the KDC to obtain a symmetric session key between Bob and herself. The message contains her registered identity (the word *Alice* in the figure) and the identity of Bob (the word *Bob* in the figure). This message is not encrypted, it is public. The KDC does not care.
2. The KDC receives the message and creates what is called a **ticket.** The ticket is encrypted using Bob's key (K_B). The ticket contains the identities of Alice and Bob and the session key (K_{AB}). The ticket with a copy of the session key is sent to Alice. Alice receives the message, decrypts it, and extracts the session key. She cannot decrypt Bob's ticket; the ticket is for Bob, not for Alice. Note that this message contains a double encryption; the ticket is encrypted, and the entire message is also encrypted. In the second message, Alice is actually authenticated to the KDC, because only Alice can open the whole message using her secret key with KDC.
3. Alice sends the ticket to Bob. Bob opens the ticket and knows that Alice needs to send messages to him using K_{AB} as the session key. Note that in this message, Bob is authenticated to the KDC because only Bob can open the ticket. Because Bob is authenticated to the KDC, he is also authenticated to Alice, who trusts the KDC. In the same way, Alice is also authenticated to Bob, because Bob trusts the KDC and the KDC has sent Bob the ticket that includes the identity of Alice.

Unfortunately, this simple protocol has a flaw. Eve can use the replay attack discussed previously. That is, she can save the message in step 3 and replay it later.

Needham-Schroeder Protocol

Another approach is the elegant **Needham-Schroeder protocol,** which is a foundation for many other protocols. This protocol uses multiple challenge-response interactions between parties to achieve a flawless protocol. Needham and Schroeder uses two nonces: R_A and R_B. Figure 15.5 shows the five steps used in this protocol.

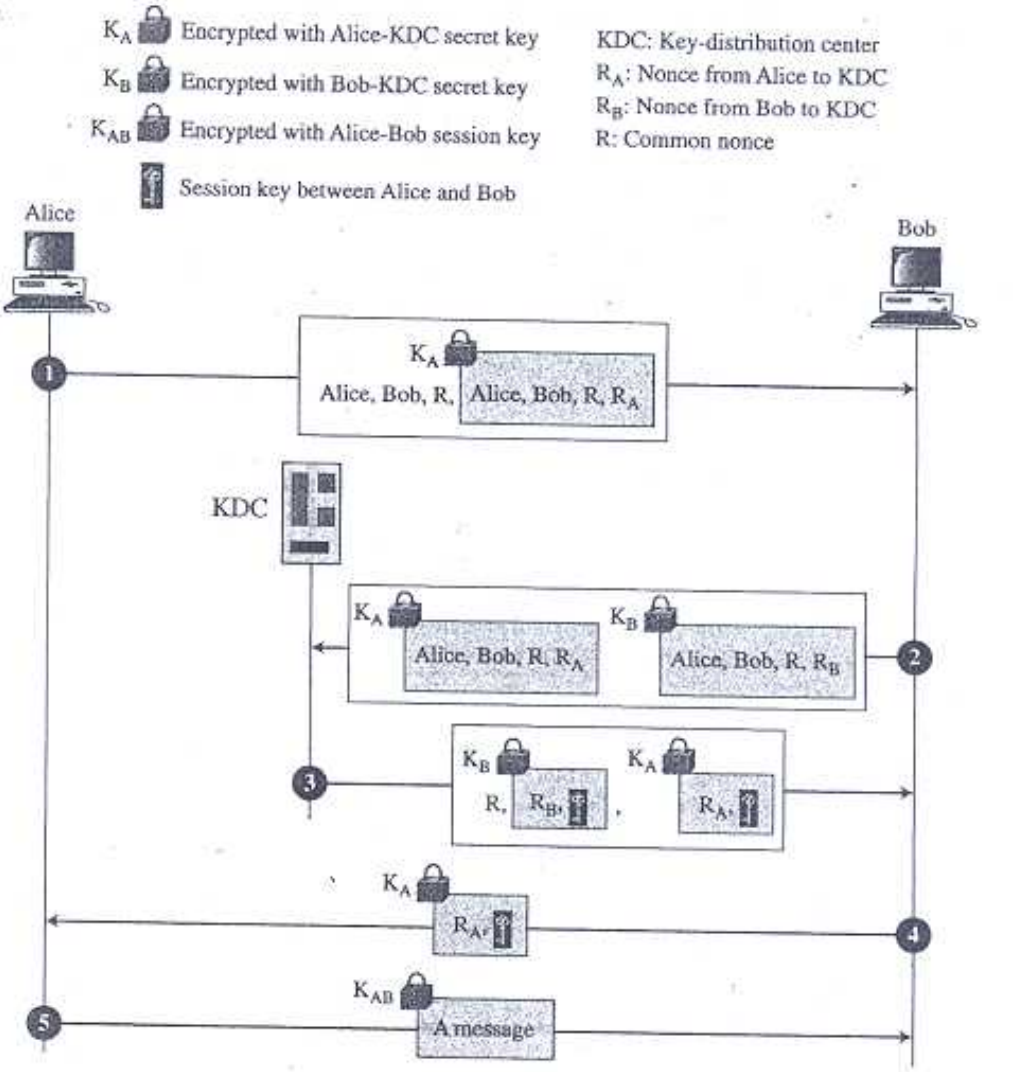
We briefly describe each step:

1. Alice sends a message to the KDC that includes her nonce, R_A, her identity, and Bob's identity.
2. The KDC sends an encrypted message to Alice that includes Alice's nonce, Bob's identity, the session key, and an encrypted ticket for Bob. The whole message is encrypted with Alice's key.
3. Alice sends Bob's ticket to him.
4. Bob sends his challenge to Alice (R_B), encrypted with the session key.
5. Alice responds to Bob's challenge. Note that the response carries $R_B - 1$ instead of R_B.

Otway-Rees Protocol

A third approach is the **Otway-Rees protocol,** another elegant protocol. Figure 15.6 shows this five-step protocol.

Figure 15.6 *Otway-Rees protocol*


K_A Encrypted with Alice-KDC secret key
K_B Encrypted with Bob-KDC secret key
K_{AB} Encrypted with Alice-Bob session key
Session key between Alice and Bob
KDC: Key-distribution center
R_A: Nonce from Alice to KDC
R_B: Nonce from Bob to KDC
R: Common nonce
Alice
Bob
KDC
1
K_A
Alice, Bob, R, Alice, Bob, R, R_A
2
K_A
Alice, Bob, R, R_A
K_B
Alice, Bob, R, R_B
3
K_B
R, R_B,
K_A
R_A,
4
K_A
R_A,
5
K_{AB}
A message

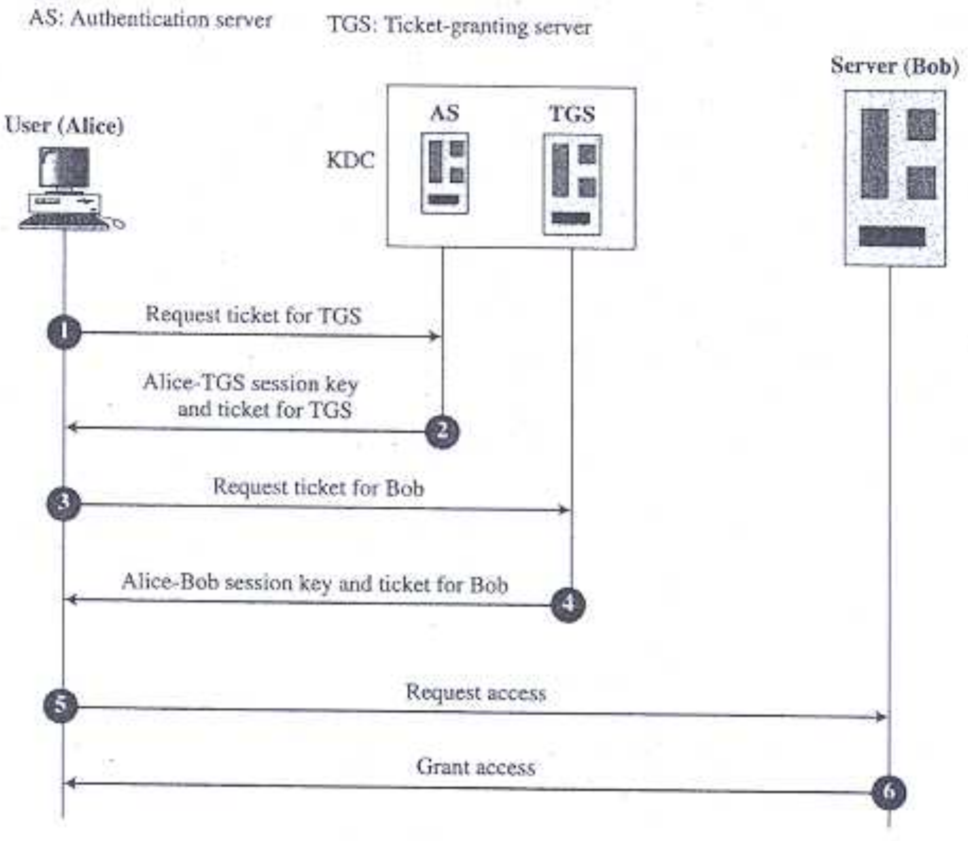
15.2 KERBEROS

Kerberos is an authentication protocol, and at the same time a KDC, that has become very popular. Several systems, including Windows 2000, use Kerberos. It is named after the three-headed dog in Greek mythology that guards the gates of Hades. Originally designed at MIT, it has gone through several versions. We only discuss version 4, the most popular, and we briefly explain the difference between version 4 and version 5 (the latest).

Servers

Three servers are involved in the Kerberos protocol: an authentication server (AS), a ticket-granting server (TGS), and a real (data) server that provides services to others. In our examples and figures, *Bob* is the real server and *Alice* is the user requesting service. Figure 15.7 shows the relationship between these three servers.

Figure 15.7 *Kerberos servers*


AS: Authentication server
TGS: Ticket-granting server
Server (Bob)
User (Alice)
KDC
AS
TGS
1
Request ticket for TGS
Alice-TGS session key and ticket for TGS
2
3
Request ticket for Bob
Alice-Bob session key and ticket for Bob
4
5
Request access
Grant access
6

Authentication Server (AS)

The **authentication server (AS)** is the KDC in the Kerberos protocol. Each user registers with the AS and is granted a user identity and a password. The AS has a database with these identities and the corresponding passwords. The AS verifies the user, issues a session key to be used between Alice and the TGS, and sends a ticket for the TGS.

Ticket-Granting Server (TGS)

The **ticket-granting server (TGS)** issues a ticket for the real server (Bob). It also provides the session key (K_{AB}) between Alice and Bob. Kerberos has separated user verification from the issuing of tickets. In this way, though Alice verifies her ID just once with the AS, she can contact the TGS multiple times to obtain tickets for different real servers.

Real Server

The real server (Bob) provides services for the user (Alice). Kerberos is designed for a client-server program, such as FTP, in which a user uses the client process to access the server process. Kerberos is not used for person-to-person authentication.

Operation

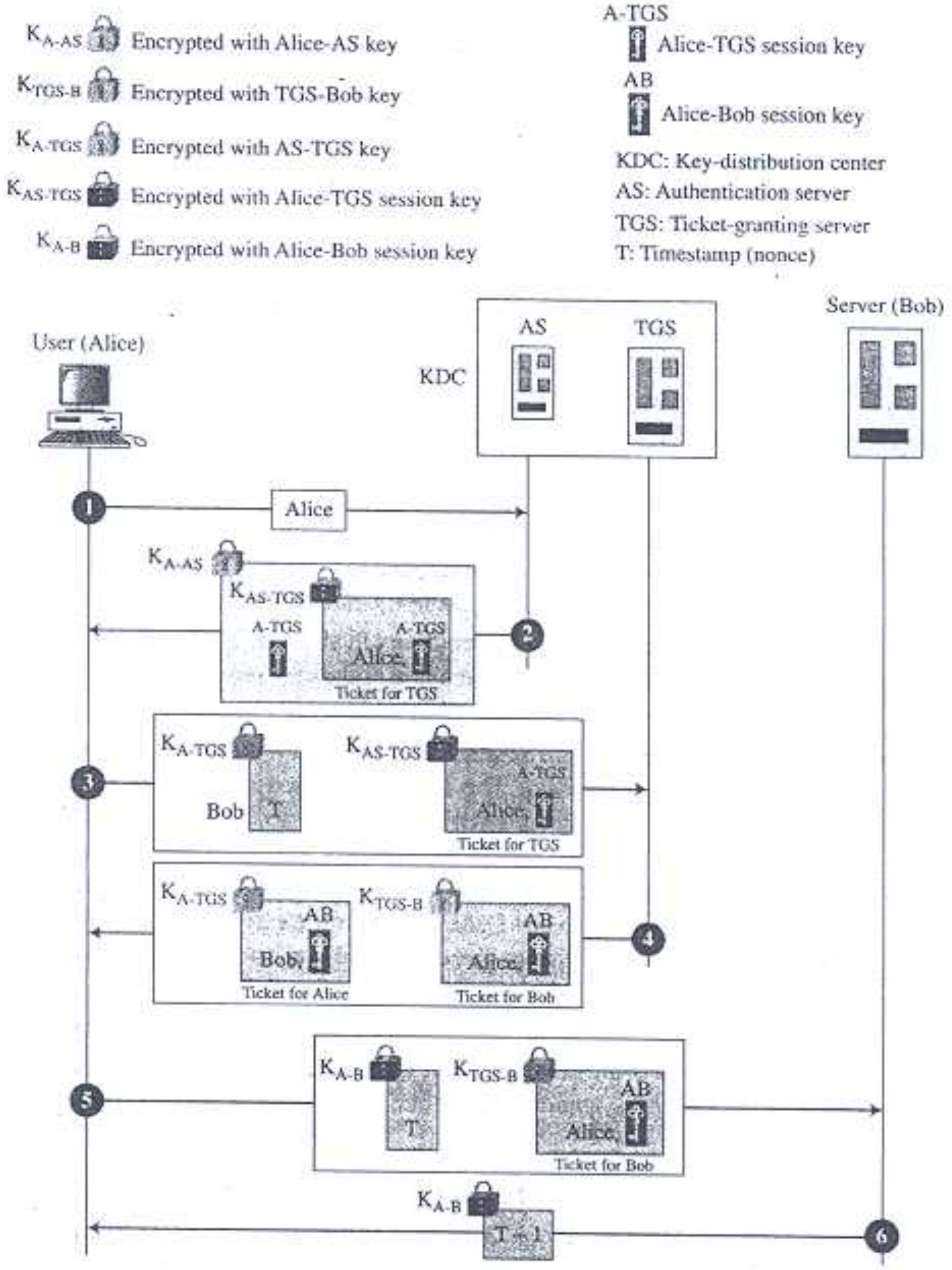
A client process (Alice) can access a process running on the real server (Bob) in six steps, as shown in Figure 15.8.

1. Alice sends her request to the AS in plain text using her registered identity.
2. The AS sends a message encrypted with Alice's permanent symmetric key, $K_{A\text{-}AS}$. The message contains two items: a session key, $K_{A\text{-}TGS}$, that is used by Alice to contact the TGS, and a ticket for the TGS that is encrypted with the TGS symmetric key, $K_{AS\text{-}TGS}$. Alice does not know $K_{A\text{-}AS}$, but when the message arrives, she types her symmetric password. The password and the appropriate algorithm together create $K_{A\text{-}AS}$ if the password is correct. The password is then immediately destroyed; it is not sent to the network and it does not stay in the terminal. It is used only for a moment to create $K_{A\text{-}AS}$. The process now uses $K_{A\text{-}AS}$ to decrypt the message sent. $K_{A\text{-}TGS}$ and the ticket are extracted.
3. Alice now sends three items to the TGS. The first is the ticket received from the AS. The second is the name of the real server (Bob), the third is a timestamp that is encrypted by $K_{A\text{-}TGS}$. The timestamp prevents a replay by Eve.
4. Now, the TGS sends two tickets, each containing the session key between Alice and Bob, $K_{A\text{-}B}$. The ticket for Alice is encrypted with $K_{A\text{-}TGS}$; the ticket for Bob is encrypted with Bob's key, $K_{TGS\text{-}B}$. Note that Eve cannot extract K_{AB} because Eve does not know $K_{A\text{-}TGS}$ or $K_{TGS\text{-}B}$. She cannot replay step 3 because she cannot replace the timestamp with a new one (she does not know $K_{A\text{-}TGS}$). Even if she is very quick and sends the step 3 message before the timestamp has expired, she still receives the same two tickets that she cannot decipher.
5. Alice sends Bob's ticket with the timestamp encrypted by $K_{A\text{-}B}$.
6. Bob confirms the receipt by adding 1 to the timestamp. The message is encrypted with $K_{A\text{-}B}$ and sent to Alice.

Using Different Servers

Note that if Alice needs to receive services from different servers, she need repeat only the last four steps. The first two steps have verified Alice's identity and need not be repeated. Alice can ask TGS to issue tickets for multiple servers by repeating steps 3 to 6.

Figure 15.8 *Kerberos example*



Kerberos Version 5

The minor differences between version 4 and version 5 are briefly listed below:

1. Version 5 has a longer ticket lifetime.
2. Version 5 allows tickets to be renewed.
3. Version 5 can accept any symmetric-key algorithm.
4. Version 5 uses a different protocol for describing data types.
5. Version 5 has more overhead than version 4.

Realms

Kerberos allows the global distribution of ASs and TGSs, with each system called a *realm*. A user may get a ticket for a local server or a remote server. In the second case, for example, Alice may ask her local TGS to issue a ticket that is accepted by a remote TGS. The local TGS can issue this ticket if the remote TGS is registered with the local one. Then Alice can use the remote TGS to access the remote real server.

15.3 SYMMETRIC-KEY AGREEMENT

Alice and Bob can create a session key between themselves without using a KDC. This method of session-key creation is referred to as the symmetric-key agreement. Although there are several ways to accomplish this, only two common methods, Diffie-Hellman and station-to-station, are discussed here.

Diffie-Hellman Key Agreement

In the **Diffie-Hellman protocol** two parties create a symmetric session key without the need of a KDC. Before establishing a symmetric key, the two parties need to choose two numbers p and g. The first number, p, is a large prime number on the order of 300 decimal digits (1024 bits). The second number, g, is a generator of order $p - 1$ in the group $<\mathbf{Z}_{p^*}, \times>$. These two (group and generator) do not need to be confidential. They can be sent through the Internet; they can be public. Figure 15.9 shows the procedure.

The steps are as follows:

1. Alice chooses a large random number x such that $0 \leq x \leq p - 1$ and calculates $R_1 = g^x \bmod p$.
2. Bob chooses another large random number y such that $0 \leq y \leq p - 1$ and calculates $R_2 = g^y \bmod p$.
3. Alice sends R_1 to Bob. Note that Alice does not send the value of x; she sends only R_1.
4. Bob sends R_2 to Alice. Again, note that Bob does not send the value of y, he sends only R_2.
5. Alice calculates $K = (R_2)^x \bmod p$.
6. Bob also calculates $K = (R_1)^y \bmod p$.

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

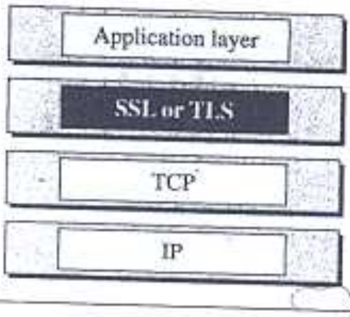
❑ To compare and contrast SSL and TLS

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.

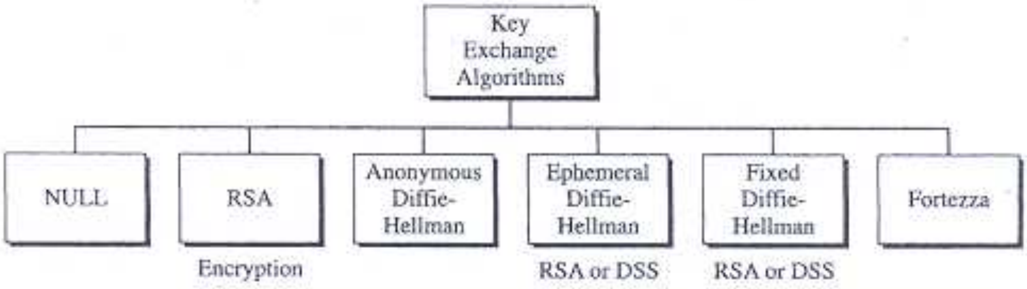
Framing

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

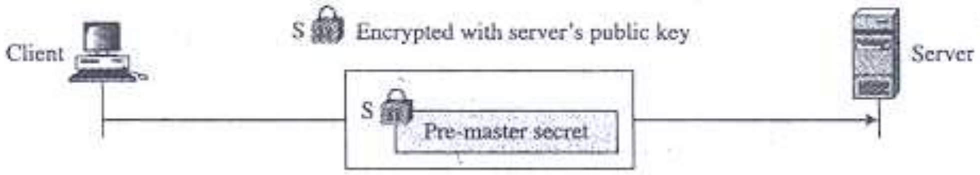
There is no key exchange in this method. No pre-master secret is established between the client and the server.

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.

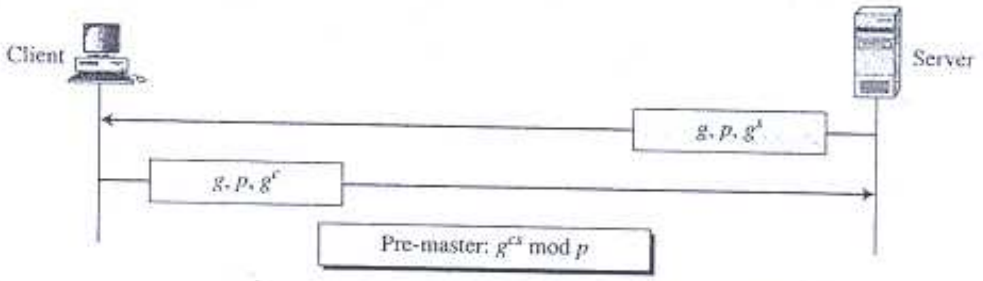
Figure 17.3 *RSA key exchange; server public key*



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.

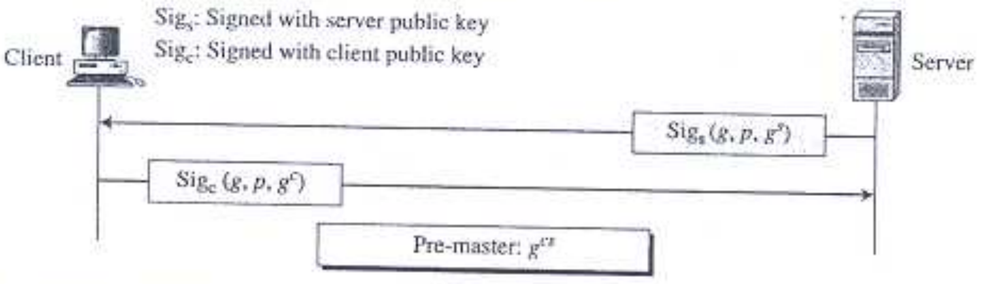
Figure 17.4 *Anonymous Diffie-Hellman key exchange*



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*


Sigs: Signed with server public key
Sigc: Signed with client public key
Client
Server
Sigs (g, p, g^s)
Sigc (g, p, g^c)
Pre-master: g^cs

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.

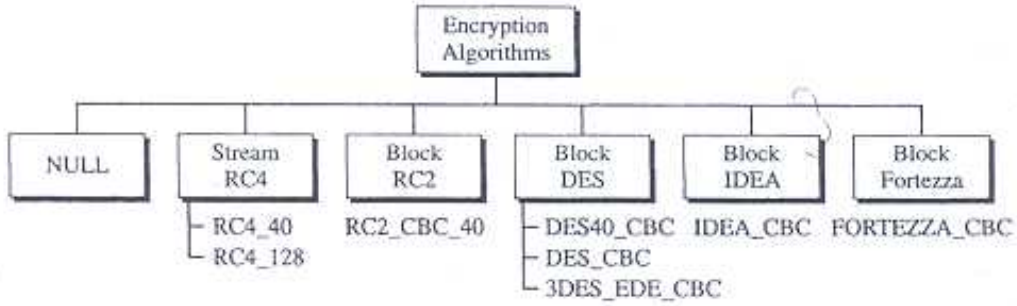
Fortezza

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.

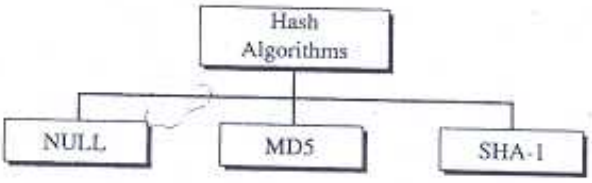
Fortezza

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.

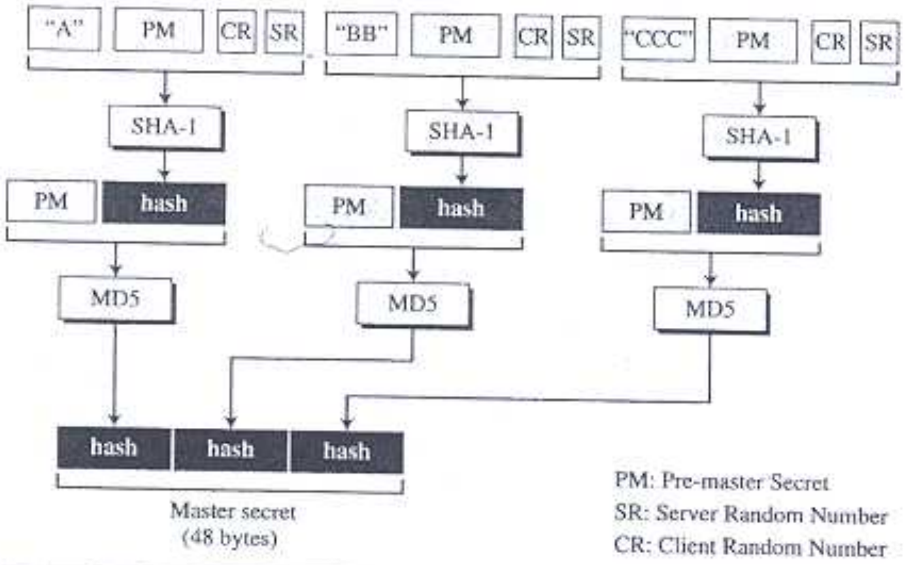
Cryptographic Parameter Generation

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.

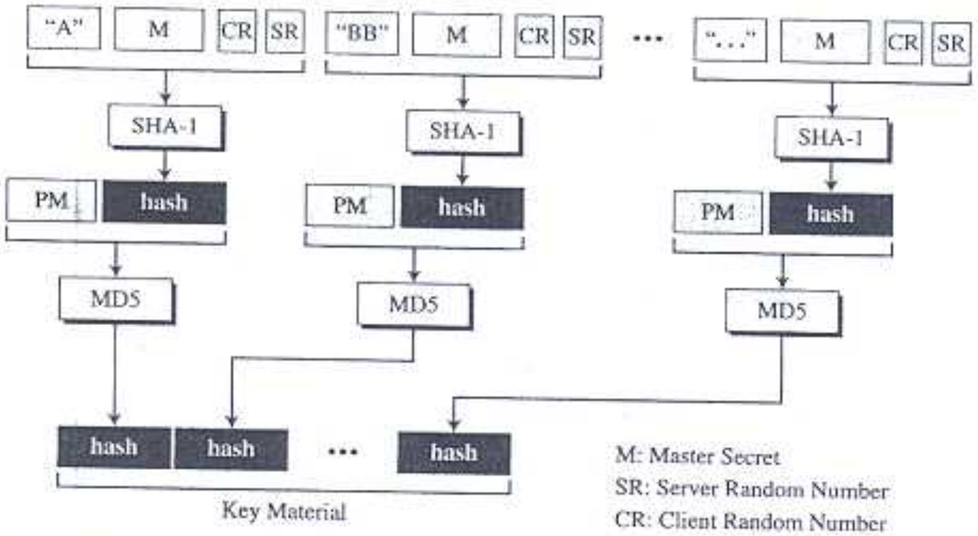
3. A 48-byte **master secret** is created from the **pre-master secret** by applying two hash functions (SHA-1 and MD5), as shown in Figure 17.8.

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.

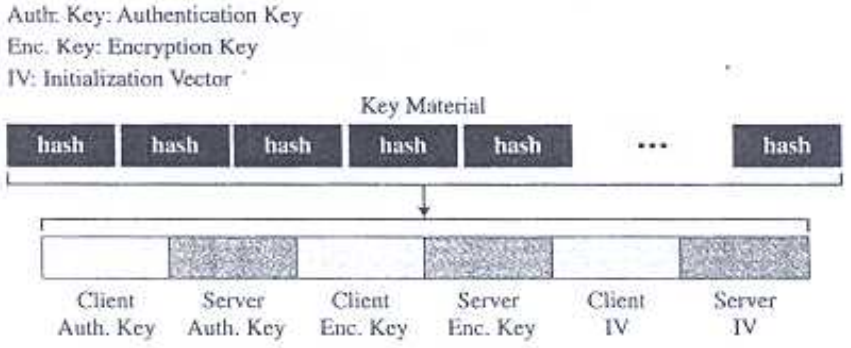
Figure 17.9 *Calculation of key material from master secret*



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.

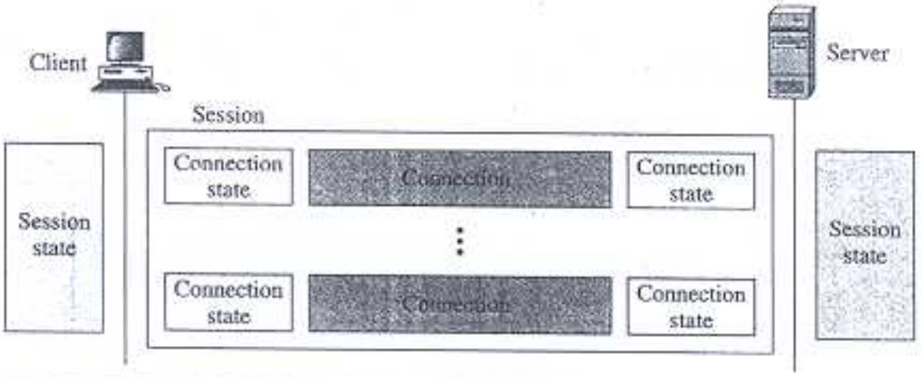
A session can consist of many connections. A connection between two parties can be terminated and reestablished within the same session. When a connection is terminated, the two parties can also terminate the session, but it is not mandatory. A session can be suspended and resumed again.

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.

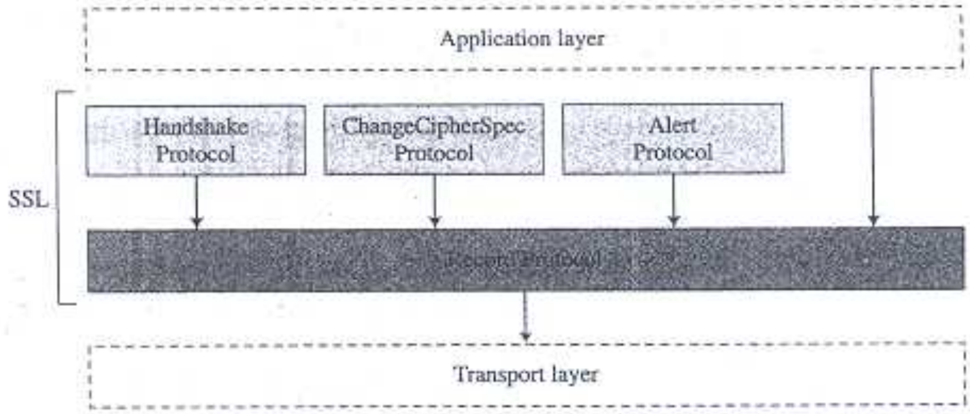
Table 17.3 *Connection state parameters*

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*

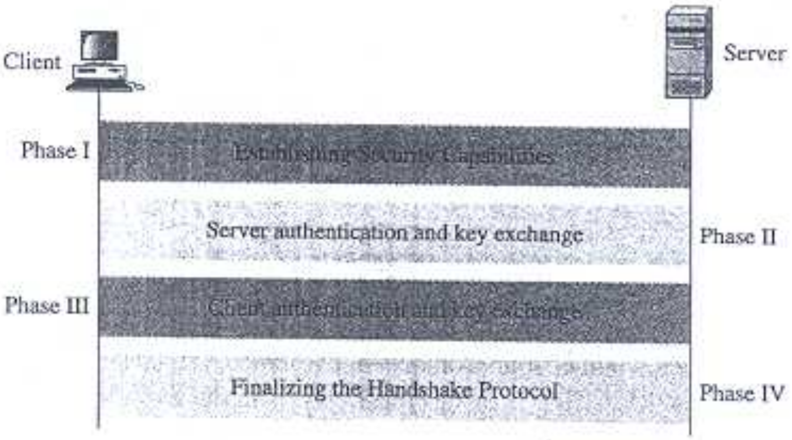


parameters. It also authenticates the server to the client and the client to the server if needed. The ChangeCipherSpec Protocol is used for signalling the readiness of cryptographic secrets. The Alert Protocol is used to report abnormal conditions. We will briefly discuss these protocols in this section.

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*


Client
Server
Phase I
Establishing Security Capabilities
Server authentication and key exchange
Phase II
Phase III
Client authentication and key exchange
Finalizing the Handshake Protocol
Phase IV

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.

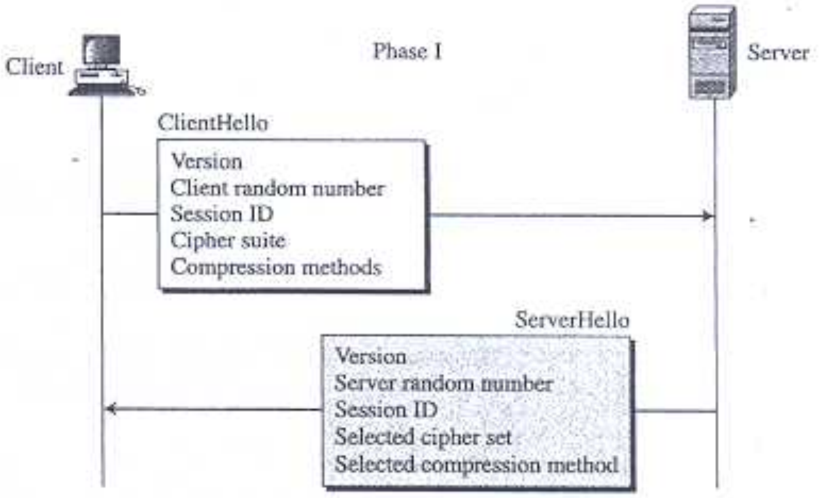
b. A 32-byte random number (from the client) that will be used for master secret generation.

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***

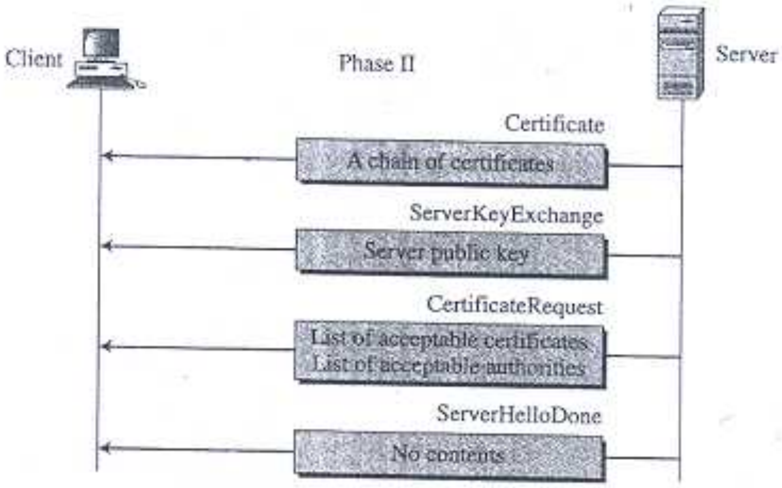
❑ ***The compression method***

❑ ***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.

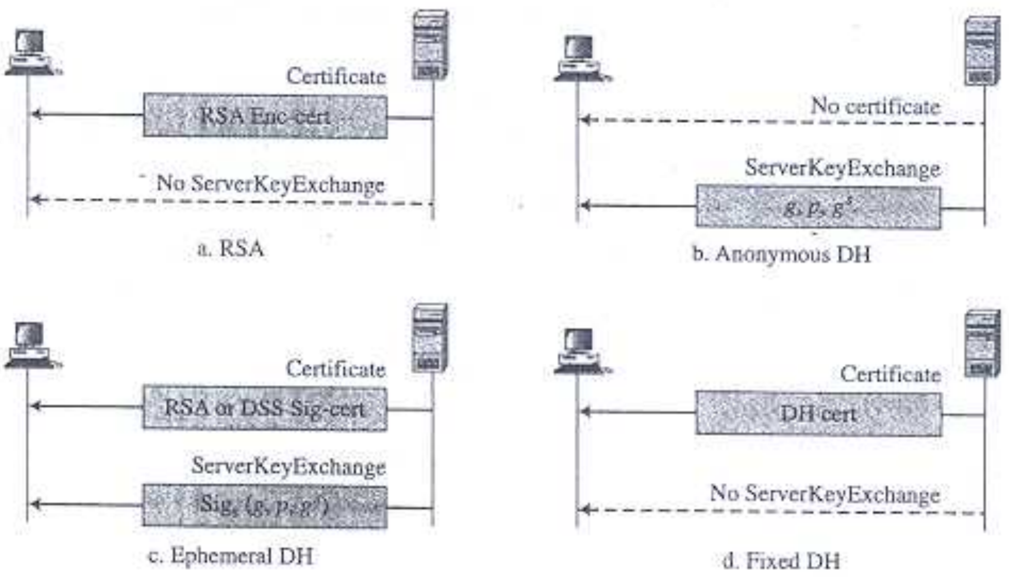
ServerHelloDone The last message in Phase II is the ServerHelloDone message, which is a signal to the client that Phase II is over and that the client needs to start Phase III.

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*



a. RSA

b. Anonymous DH

c. Ephemeral DH

d. Fixed DH

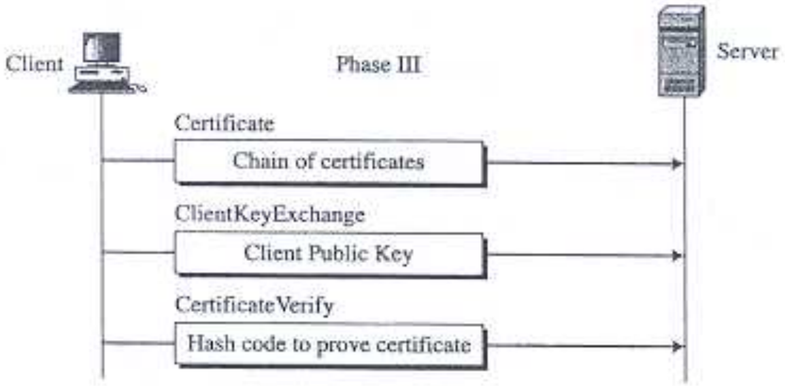
- ❑ **RSA.** In this method, the server sends its RSA encryption/decryption public-key certificate in the first message. The second message, however, is empty because the pre-master secret is generated and sent by the client in the next phase. Note that the public-key certificate authenticates the server 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.

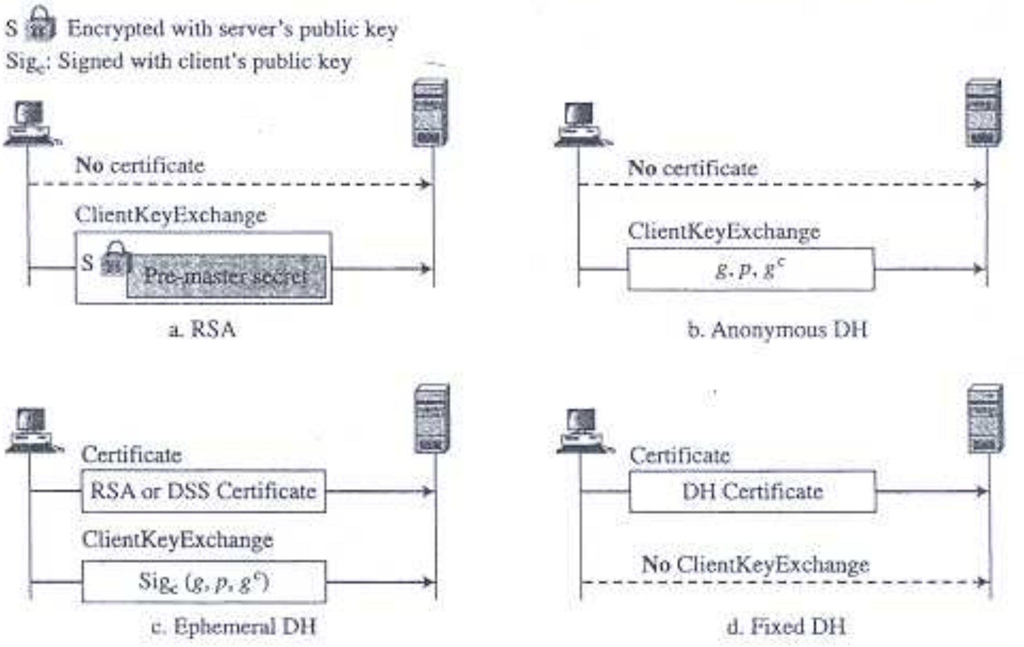
CertificateVerify If the client has sent a certificate declaring that it owns the public key in the certificate, it needs to prove that it knows the corresponding private key. This is needed to thwart an impostor who sends the certificate and claims that it comes from the client. The proof of private-key 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*



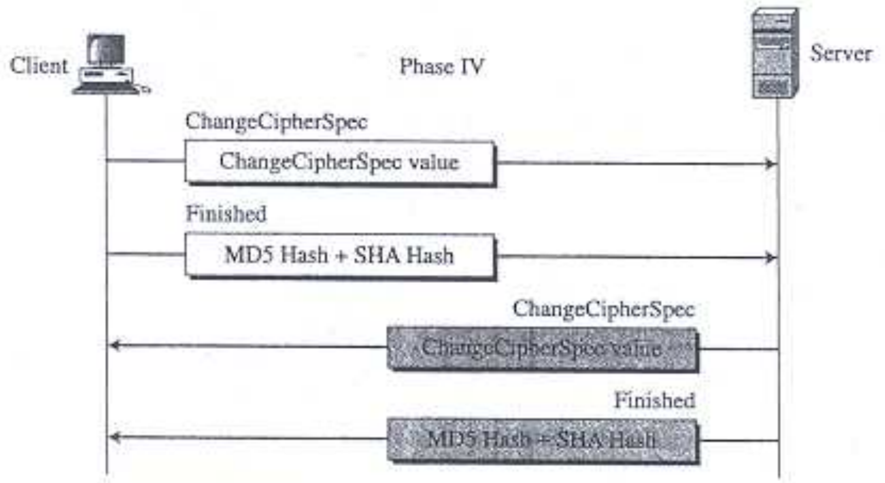
- ❑ **RSA.** In this case, there is no Certificate message unless the server has explicitly requested one in Phase II. The ClientKeyExchange method includes the pre-master key encrypted with the RSA public key received in Phase II.
- ❑ **Anonymous DH.** In this method, there is no Certificate message. 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.

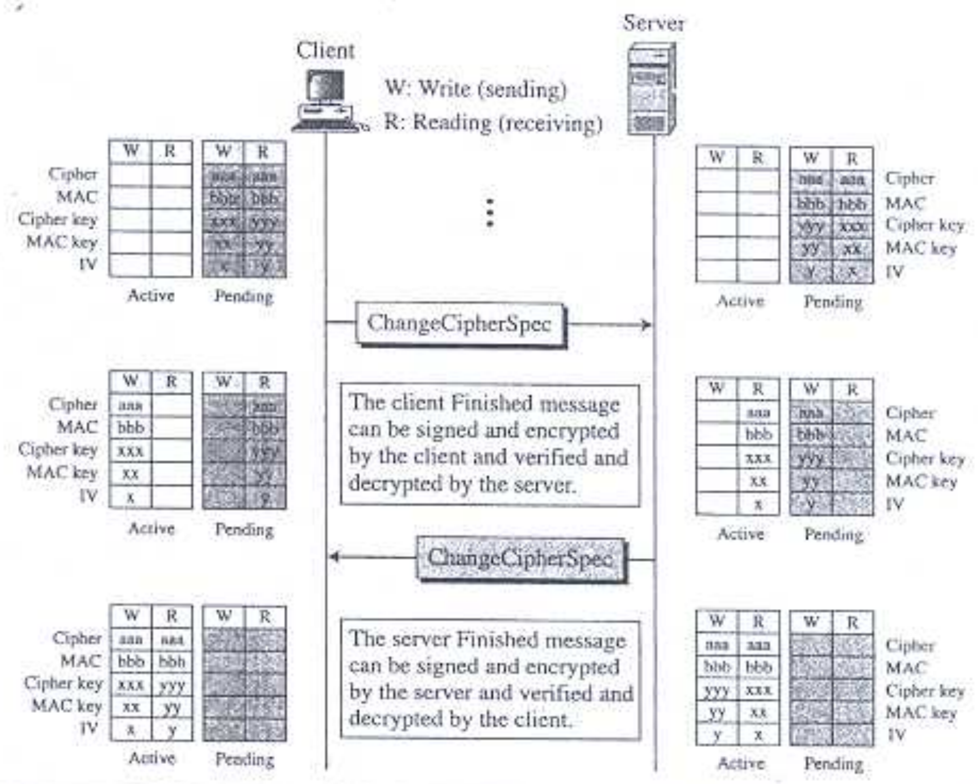
Finished Finally, the server sends a Finished message to show that handshaking is totally completed.

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*



The client Finished message can be signed and encrypted by the client and verified and decrypted by the server.

The server Finished message can be signed and encrypted by the server and verified and decrypted by the client.

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.

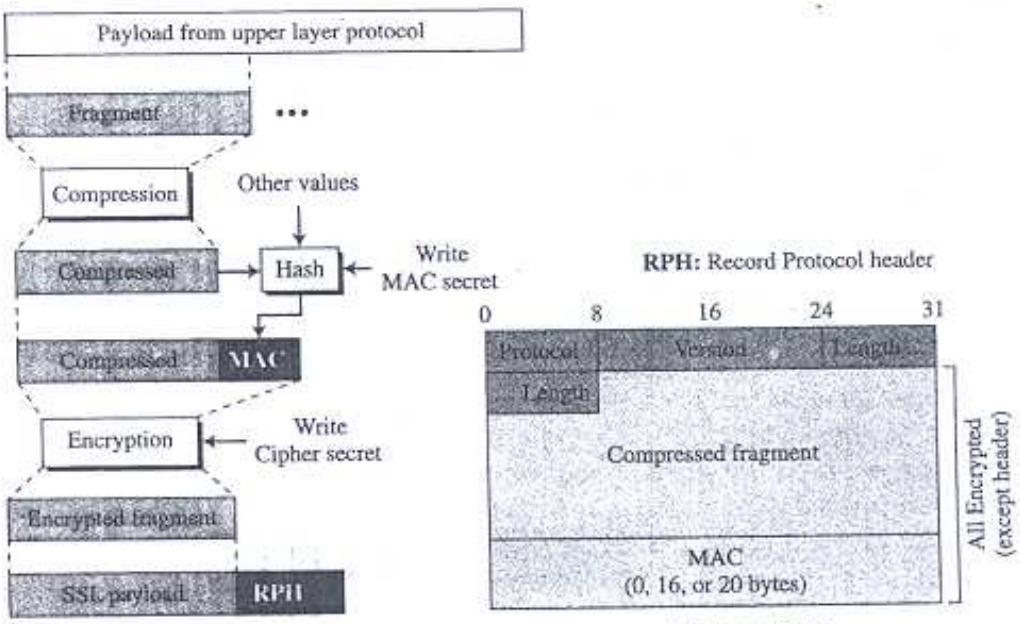
Table 17.4 *Alerts defined for SSL*

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*



a. Process

b. Encapsulation

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.

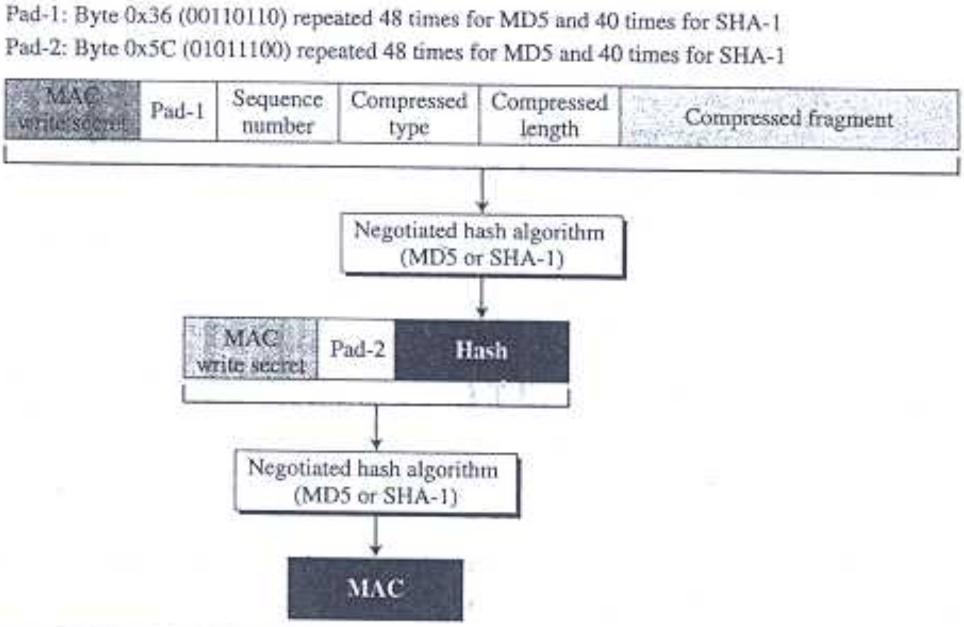
Compression/Decompression

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*


Pad-1: Byte 0x36 (00110110) repeated 48 times for MD5 and 40 times for SHA-1
Pad-2: Byte 0x5C (01011100) repeated 48 times for MD5 and 40 times for SHA-1
MAC write secret
Pad-1
Sequence number
Compressed type
Compressed length
Compressed fragment
Negotiated hash algorithm (MD5 or SHA-1)
MAC write secret
Pad-2
Hash
Negotiated hash algorithm (MD5 or SHA-1)
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.

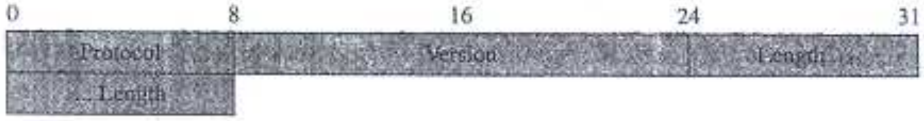
Framing/Deframing

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.

Figure 17.37 *Finished message*

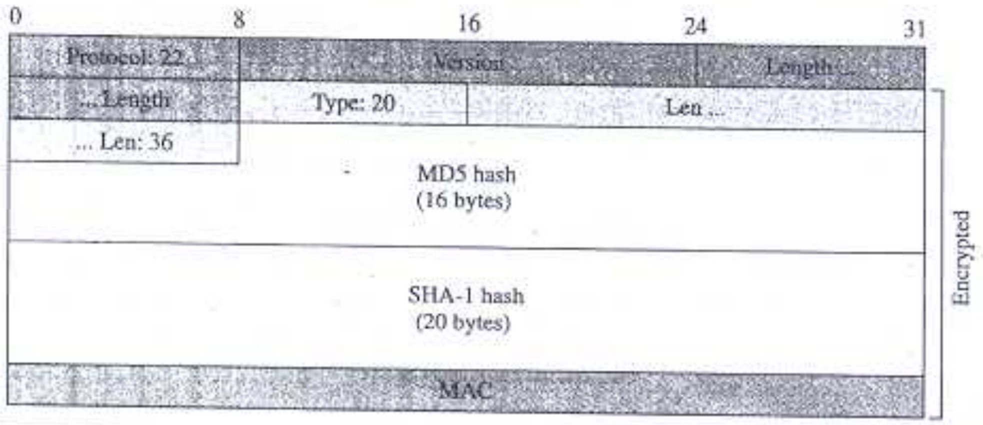
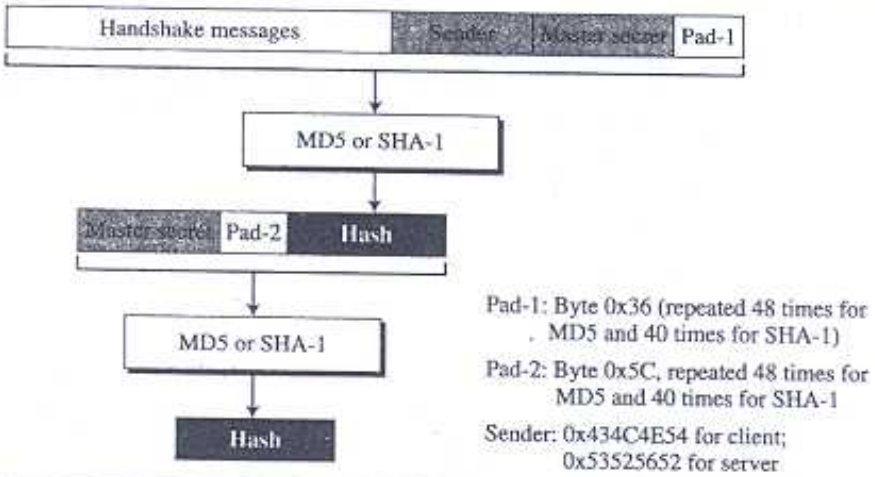


Figure 17.38 *Hash calculation for Finished message*



Pad-1: Byte 0x36 (repeated 48 times for MD5 and 40 times for SHA-1)

Pad-2: Byte 0x5C, repeated 48 times for MD5 and 40 times for SHA-1

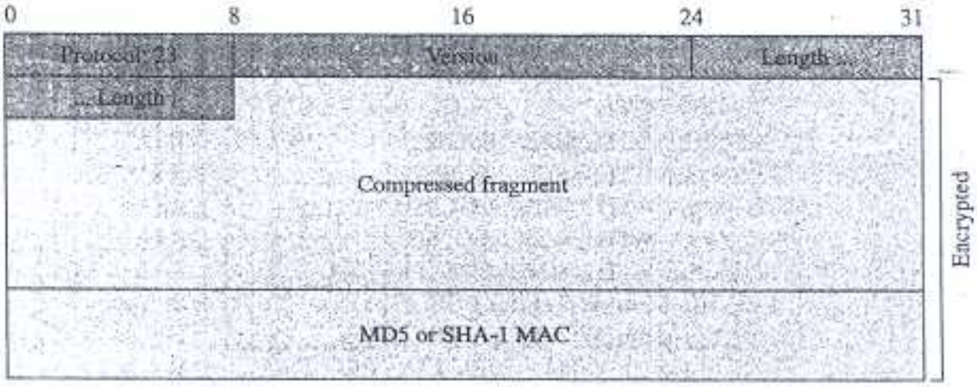
Sender: 0x434C4E54 for client; 0x53525652 for server

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

CHAPTER 18

ELECTRONIC MAIL SECURITY

Despite the refusal of VADM Poindexter and LtCol North to appear, the Board's access to other sources of information filled much of this gap. The FBI provided documents taken from the files of the National Security Advisor and relevant NSC staff members, including messages from the PROF system between VADM Poindexter and LtCol North. The PROF messages were conversations by computer, written at the time events occurred and presumed by the writers to be protected from disclosure. In this sense, they provide a first-hand, contemporaneous account of events.

—The Tower Commission Report to President Reagan on the Iran-Contra Affair, 1987

KEY POINTS

- PGP is an open-source, freely available software package for e-mail security. It provides authentication through the use of digital signature, confidentiality through the use of symmetric block encryption, compression using the ZIP algorithm, and e-mail compatibility using the radix-64 encoding scheme.
- PGP incorporates tools for developing a public-key trust model and public-key certificate management.
- S/MIME is an Internet standard approach to e-mail security that incorporates the same functionality as PGP.
- DKIM is a specification used by e-mail providers for cryptographically signing e-mail messages on behalf of the source domain.

In virtually all distributed environments, electronic mail is the most heavily used network-based application. Users expect to be able to, and do, send e-mail to others who are connected directly or indirectly to the Internet, regardless of host operating system or communications suite. With the explosively growing reliance on e-mail, there grows a demand for authentication and confidentiality services. Two schemes stand out as approaches that enjoy widespread use: Pretty Good Privacy (PGP) and S/MIME. Both are examined in this chapter. The chapter closes with a discussion of DomainKeys Identified Mail.

18.1 PRETTY GOOD PRIVACY

PGP is a remarkable phenomenon. Largely the effort of a single person, Phil Zimmermann, PGP provides a confidentiality and authentication service that can be used for electronic mail and file storage applications. In essence, Zimmermann has done the following:

1. Selected the best available cryptographic algorithms as building blocks.
2. Integrated these algorithms into a general-purpose application that is independent of operating system and processor and that is based on a small set of easy-to-use commands.
3. Made the package and its documentation, including the source code, freely available via the Internet, bulletin boards, and commercial networks such as AOL (America On Line).
4. Entered into an agreement with a company (Viacrypt, now Network Associates) to provide a fully compatible, low-cost commercial version of PGP.

PGP has grown explosively and is now widely used. A number of reasons can be cited for this growth.

1. It is available free worldwide in versions that run on a variety of platforms, including Windows, UNIX, Macintosh, and many more. In addition, the commercial version satisfies users who want a product that comes with vendor support.
2. It is based on algorithms that have survived extensive public review and are considered extremely secure. Specifically, the package includes RSA, DSS, and Diffie-Hellman for public-key encryption; CAST-128, IDEA, and 3DES for symmetric encryption; and SHA-1 for hash coding.
3. It has a wide range of applicability, from corporations that wish to select and enforce a standardized scheme for encrypting files and messages to individuals who wish to communicate securely with others worldwide over the Internet and other networks.
4. It was not developed by, nor is it controlled by, any governmental or standards organization. For those with an instinctive distrust of "the establishment," this makes PGP attractive.
5. PGP is now on an Internet standards track (RFC 3156; *MIME Security with OpenPGP*). Nevertheless, PGP still has an aura of an antiestablishment endeavor.

We begin with an overall look at the operation of PGP. Next, we examine how cryptographic keys are created and stored. Then, we address the vital issue of public-key management.

Notation

Most of the notation used in this chapter has been used before, but a few terms are new. It is perhaps best to summarize those at the beginning. The following symbols are used.

K_s = session key used in symmetric encryption scheme
PR_a = private key of user A, used in public-key encryption scheme
PU_a = public key of user A, used in public-key encryption scheme
EP = public-key encryption
DP = public-key decryption
EC = symmetric encryption
DC = symmetric decryption

H	=	hash function
||	=	concatenation
Z	=	compression using ZIP algorithm
R64	=	conversion to radix 64 ASCII format[1]

The PGP documentation often uses the term *secret key* to refer to a key paired with a public key in a public-key encryption scheme. As was mentioned earlier, this practice risks confusion with a secret key used for symmetric encryption. Hence, we use the term *private key* instead.

Operational Description

The actual operation of PGP, as opposed to the management of keys, consists of four services: authentication, confidentiality, compression, and e-mail compatibility (Table 18.1). We examine each of these in turn.

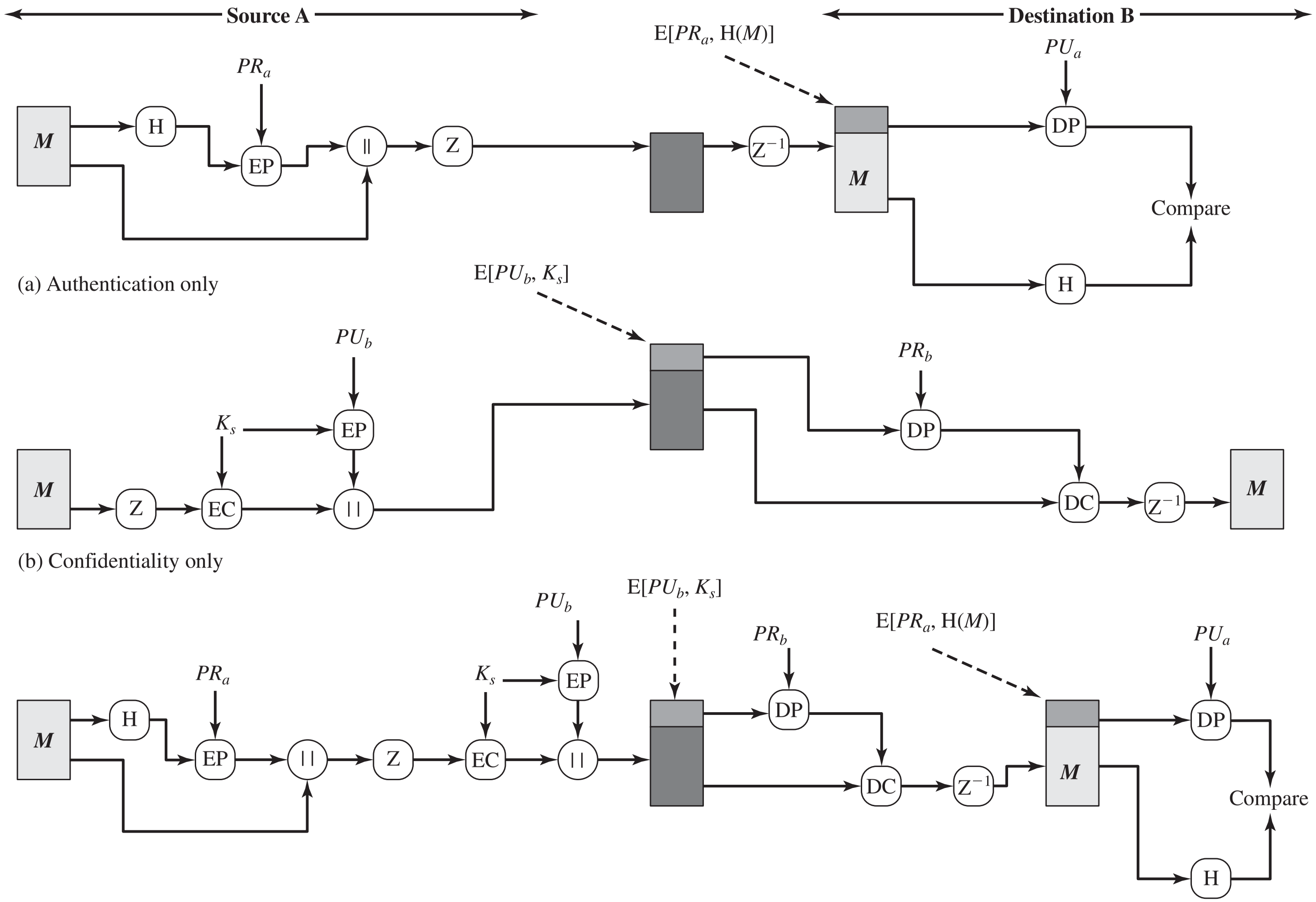
Authentication Figure 18.1a illustrates the digital signature service provided by PGP. This is the digital signature scheme discussed in Chapter 13 and illustrated in Figure 13.2. The sequence is as follows.

1. The sender creates a message.
2. SHA-1 is used to generate a 160-bit hash code of the message.
3. The hash code is encrypted with RSA using the sender's private key, and the result is prepended to the message.
4. The receiver uses RSA with the sender's public key to decrypt and recover the hash code.
5. The receiver generates a new hash code for the message and compares it with the decrypted hash code. If the two match, the message is accepted as authentic.

Table 18.1 Summary of PGP Services

Function	Algorithms Used	Description
Digital signature	DSS/SHA or RSA/SHA	A hash code of a message is created using SHA-1. This message digest is encrypted using DSS or RSA with the sender's private key and included with the message.
Message encryption	CAST or IDEA or Three-key Triple DES with Diffie-Hellman or RSA	A message is encrypted using CAST-128 or IDEA or 3DES with a one-time session key generated by the sender. The session key is encrypted using Diffie-Hellman or RSA with the recipient's public key and included with the message.
Compression	ZIP	A message may be compressed for storage or transmission using ZIP.
E-mail compatibility	Radix-64 conversion	To provide transparency for e-mail applications, an encrypted message may be converted to an ASCII string using radix-64 conversion.

[1] The American Standard Code for Information Interchange (ASCII) is described in Appendix Q.



(a) Authentication only

(b) Confidentiality only

(c) Confidentiality and authentication

Figure 18.1 PGP Cryptographic Functions

The combination of SHA-1 and RSA provides an effective digital signature scheme. Because of the strength of RSA, the recipient is assured that only the possessor of the matching private key can generate the signature. Because of the strength of SHA-1, the recipient is assured that no one else could generate a new message that matches the hash code and, hence, the signature of the original message.

As an alternative, signatures can be generated using DSS/SHA-1.

Although signatures normally are found attached to the message or file that they sign, this is not always the case: Detached signatures are supported. A detached signature may be stored and transmitted separately from the message it signs. This is useful in several contexts. A user may wish to maintain a separate signature log of all messages sent or received. A detached signature of an executable program can detect subsequent virus infection. Finally, detached signatures can be used when more than one party must sign a document, such as a legal contract. Each person's signature is independent and therefore is applied only to the document. Otherwise, signatures would have to be nested, with the second signer signing both the document and the first signature, and so on.

Confidentiality Another basic service provided by PGP is confidentiality, which is provided by encrypting messages to be transmitted or to be stored locally as files. In both cases, the symmetric encryption algorithm CAST-128 may be used. Alternatively, IDEA or 3DES may be used. The 64-bit cipher feedback (CFB) mode is used.

As always, one must address the problem of key distribution. In PGP, each symmetric key is used only once. That is, a new key is generated as a random 128-bit number for each message. Thus, although this is referred to in the documentation as a session key, it is in reality a one-time key. Because it is to be used only once, the session key is bound to the message and transmitted with it. To protect the key, it is encrypted with the receiver's public key. Figure 18.1b illustrates the sequence, which can be described as follows.

1. The sender generates a message and a random 128-bit number to be used as a session key for this message only.
2. The message is encrypted using CAST-128 (or IDEA or 3DES) with the session key.
3. The session key is encrypted with RSA using the recipient's public key and is prepended to the message.
4. The receiver uses RSA with its private key to decrypt and recover the session key.
5. The session key is used to decrypt the message.

As an alternative to the use of RSA for key encryption, PGP provides an option referred to as *Diffie-Hellman*. As was explained in Chapter 10, Diffie-Hellman is a key exchange algorithm. In fact, PGP uses a variant of Diffie-Hellman that does provide encryption/decryption, known as ElGamal (Chapter 10).

Several observations may be made. First, to reduce encryption time, the combination of symmetric and public-key encryption is used in preference to simply using

RSA or ElGamal to encrypt the message directly: CAST-128 and the other symmetric algorithms are substantially faster than RSA or ElGamal. Second, the use of the public-key algorithm solves the session-key distribution problem, because only the recipient is able to recover the session key that is bound to the message. Note that we do not need a session-key exchange protocol of the type discussed in Chapter 14, because we are not beginning an ongoing session. Rather, each message is a one-time independent event with its own key. Furthermore, given the store-and-forward nature of electronic mail, the use of handshaking to assure that both sides have the same session key is not practical. Finally, the use of one-time symmetric keys strengthens what is already a strong symmetric encryption approach. Only a small amount of plaintext is encrypted with each key, and there is no relationship among the keys. Thus, to the extent that the public-key algorithm is secure, the entire scheme is secure. To this end, PGP provides the user with a range of key size options from 768 to 3072 bits (the DSS key for signatures is limited to 1024 bits).

Confidentiality and Authentication As Figure 18.1c illustrates, both services may be used for the same message. First, a signature is generated for the plaintext message and prepended to the message. Then the plaintext message plus signature is encrypted using CAST-128 (or IDEA or 3DES), and the session key is encrypted using RSA (or ElGamal). This sequence is preferable to the opposite: encrypting the message and then generating a signature for the encrypted message. It is generally more convenient to store a signature with a plaintext version of a message. Furthermore, for purposes of third-party verification, if the signature is performed first, a third party need not be concerned with the symmetric key when verifying the signature.

In summary, when both services are used, the sender first signs the message with its own private key, then encrypts the message with a session key, and finally encrypts the session key with the recipient's public key.

Compression As a default, PGP compresses the message after applying the signature but before encryption. This has the benefit of saving space both for e-mail transmission and for file storage.

The placement of the compression algorithm, indicated by Z for compression and Z^{-1} for decompression in Figure 18.1, is critical.

1. The signature is generated before compression for two reasons:
   a. It is preferable to sign an uncompressed message so that one can store only the uncompressed message together with the signature for future verification. If one signed a compressed document, then it would be necessary either to store a compressed version of the message for later verification or to recompress the message when verification is required.
   b. Even if one were willing to generate dynamically a recompressed message for verification, PGP's compression algorithm presents a difficulty. The algorithm is not deterministic; various implementations of the algorithm achieve different tradeoffs in running speed versus compression ratio and, as a result, produce different compressed forms. However, these different compression algorithms are interoperable because any version of the algorithm can correctly decompress the output of any other version. Applying the hash

function and signature after compression would constrain all PGP implementations to the same version of the compression algorithm.

2. Message encryption is applied after compression to strengthen cryptographic security. Because the compressed message has less redundancy than the original plaintext, cryptanalysis is more difficult.

The compression algorithm used is ZIP, which is described in Appendix O.

E-mail Compatibility When PGP is used, at least part of the block to be transmitted is encrypted. If only the signature service is used, then the message digest is encrypted (with the sender's private key). If the confidentiality service is used, the message plus signature (if present) are encrypted (with a one-time symmetric key). Thus, part or all of the resulting block consists of a stream of arbitrary 8-bit octets. However, many electronic mail systems only permit the use of blocks consisting of ASCII text. To accommodate this restriction, PGP provides the service of converting the raw 8-bit binary stream to a stream of printable ASCII characters.

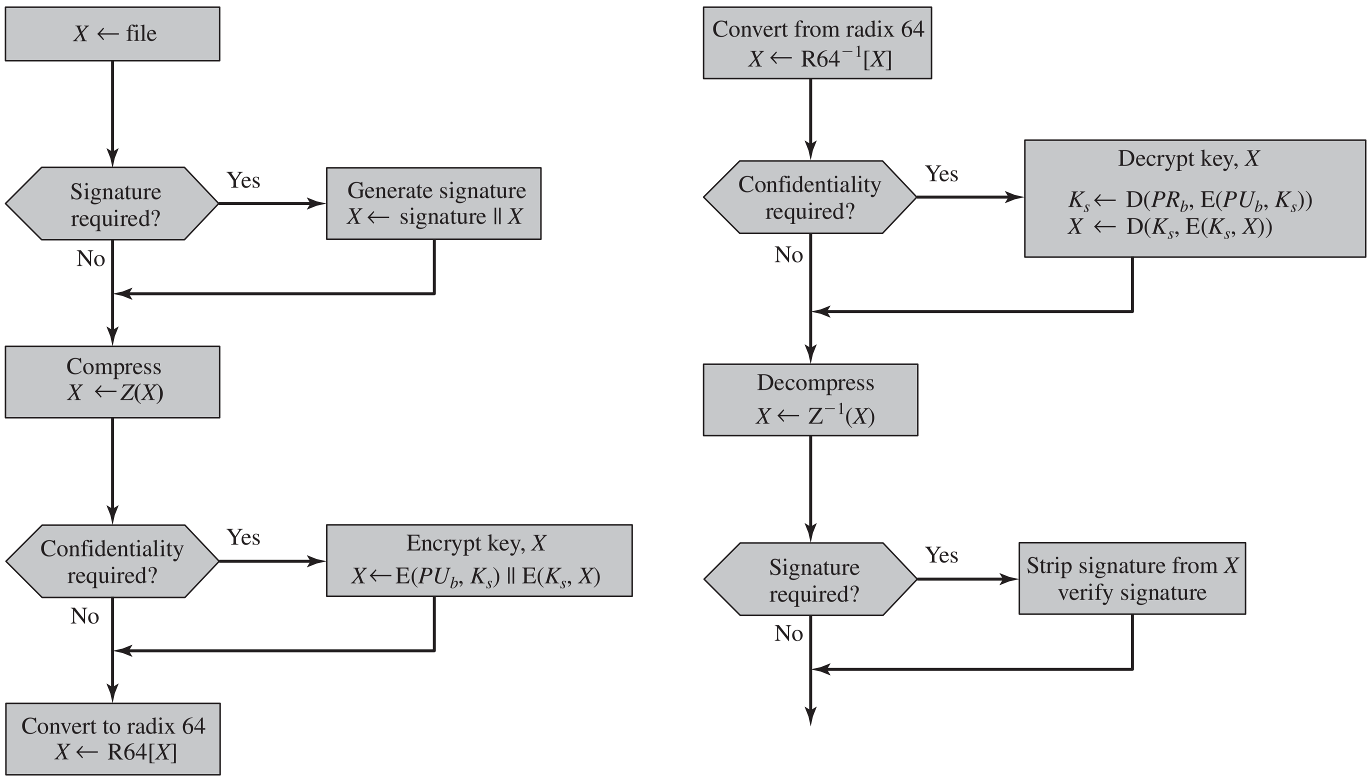
The scheme used for this purpose is radix-64 conversion. Each group of three octets of binary data is mapped into four ASCII characters. This format also appends a CRC to detect transmission errors. See Appendix 18A for a description.

The use of radix 64 expands a message by 33%. Fortunately, the session key and signature portions of the message are relatively compact, and the plaintext message has been compressed. In fact, the compression should be more than enough to compensate for the radix-64 expansion. For example, [HELD96] reports an average compression ratio of about 2.0 using ZIP. If we ignore the relatively small signature and key components, the typical overall effect of compression and expansion of a file of length X would be $1.33 \times 0.5 \times X = 0.665 \times X$. Thus, there is still an overall compression of about one-third.

One noteworthy aspect of the radix-64 algorithm is that it blindly converts the input stream to radix-64 format regardless of content, even if the input happens to be ASCII text. Thus, if a message is signed but not encrypted and the conversion is applied to the entire block, the output will be unreadable to the casual observer, which provides a certain level of confidentiality. As an option, PGP can be configured to convert to radix-64 format only the signature portion of signed plaintext messages. This enables the human recipient to read the message without using PGP. PGP would still have to be used to verify the signature.

Figure 18.2 shows the relationship among the four services so far discussed. On transmission (if it is required), a signature is generated using a hash code of the uncompressed plaintext. Then the plaintext (plus signature if present) is compressed. Next, if confidentiality is required, the block (compressed plaintext or compressed signature plus plaintext) is encrypted and prepended with the public-key-encrypted symmetric encryption key. Finally, the entire block is converted to radix-64 format.

On reception, the incoming block is first converted back from radix-64 format to binary. Then, if the message is encrypted, the recipient recovers the session key and decrypts the message. The resulting block is then decompressed. If the message is signed, the recipient recovers the transmitted hash code and compares it to its own calculation of the hash code.


X ← file
Signature required?
Yes
No
Generate signature X ← signature || X
Compress X ← Z(X)
Confidentiality required?
Yes
No
Encrypt key, X X ← E(PU_b, K_s) || E(K_s, X)
Convert to radix 64 X ← R64[X]
Convert from radix 64 X ← R64^{−1}[X]
Confidentiality required?
Yes
No
Decrypt key, X
K_s ← D(PR_b, E(PU_b, K_s))
X ← D(K_s, E(K_s, X))
Decompress X ← Z^{−1}(X)
Signature required?
Yes
No
Strip signature from X verify signature

(a) Generic transmission diagram (from A)

(b) Generic reception diagram (to B)

Figure 18.2 Transmission and Reception of PGP Messages

> PGP cannot assume that this key is legitimate simply because it came from a reputable server. The user must declare the key legitimate by signing it or by telling PGP that it is willing to trust fully one of the key's signatories.

A final point: Earlier it was mentioned that multiple user IDs may be associated with a single public key on the public-key ring. This could be because a person has changed names or has been introduced via signature under multiple names, indicating different e-mail addresses for the same person, for example. So we can think of a public key as the root of a tree. A public key has a number of user IDs associating with it, with a number of signatures below each user ID. The binding of a particular user ID to a key depends on the signatures associated with that user ID and that key, whereas the level of trust in this key (for use in signing other keys) is a function of all the dependent signatures.

Revoking Public Keys A user may wish to revoke his or her current public key either because compromise is suspected or simply to avoid the use of the same key for an extended period. Note that a compromise would require that an opponent somehow had obtained a copy of your unencrypted private key or that the opponent had obtained both the private key from your private-key ring and your passphrase.

The convention for revoking a public key is for the owner to issue a key revocation certificate, signed by the owner. This certificate has the same form as a normal signature certificate but includes an indicator that the purpose of this certificate is to revoke the use of this public key. Note that the corresponding private key must be used to sign a certificate that revokes a public key. The owner should then attempt to disseminate this certificate as widely and as quickly as possible to enable potential correspondents to update their public-key rings.

Note that an opponent who has compromised the private key of an owner can also issue such a certificate. However, this would deny the opponent as well as the legitimate owner the use of the public key, and therefore, it seems a much less likely threat than the malicious use of a stolen private key.

18.2 S/MIME

Secure/Multipurpose Internet Mail Extension (S/MIME) is a security enhancement to the MIME Internet e-mail format standard based on technology from RSA Data Security. Although both PGP and S/MIME are on an IETF standards track, it appears likely that S/MIME will emerge as the industry standard for commercial and organizational use, while PGP will remain the choice for personal e-mail security for many users. S/MIME is defined in a number of documents—most importantly RFCs 3370, 3850, 3851, and 3852.

To understand S/MIME, we need first to have a general understanding of the underlying e-mail format that it uses, namely MIME. But to understand the significance of MIME, we need to go back to the traditional e-mail format standard, RFC 822, which is still in common use. The most recent version of this format specification is RFC 5322 (*Internet Message Format*). Accordingly, this section first provides an introduction to these two earlier standards and then moves on to a discussion of S/MIME.

RFC 5322

RFC 5322 defines a format for text messages that are sent using electronic mail. It has been the standard for Internet-based text mail messages and remains in common use. In the RFC 5322 context, messages are viewed as having an envelope and contents. The envelope contains whatever information is needed to accomplish transmission and delivery. The contents compose the object to be delivered to the recipient. The RFC 5322 standard applies only to the contents. However, the content standard includes a set of header fields that may be used by the mail system to create the envelope, and the standard is intended to facilitate the acquisition of such information by programs.

The overall structure of a message that conforms to RFC 5322 is very simple. A message consists of some number of header lines (*the header*) followed by unrestricted text (*the body*). The header is separated from the body by a blank line. Put differently, a message is ASCII text, and all lines up to the first blank line are assumed to be header lines used by the user agent part of the mail system.

A header line usually consists of a keyword, followed by a colon, followed by the keyword's arguments; the format allows a long line to be broken up into several lines. The most frequently used keywords are *From*, *To*, *Subject*, and *Date*. Here is an example message:

```
Date: October 8, 2009 2:15:49 PM EDT
From: "William Stallings" <ws@shore.net>
Subject: The Syntax in RFC 5322
To: Smith@Other-host.com
Cc: Jones@Yet-Another-Host.com

Hello. This section begins the actual
message body, which is delimited from the
message heading by a blank line.
```

Another field that is commonly found in RFC 5322 headers is *Message-ID*. This field contains a unique identifier associated with this message.

Multipurpose Internet Mail Extensions

Multipurpose Internet Mail Extension (MIME) is an extension to the RFC 5322 framework that is intended to address some of the problems and limitations of the use of Simple Mail Transfer Protocol (SMTP), defined in RFC 821, or some other mail transfer protocol and RFC 5322 for electronic mail. [PARZ06] lists the following limitations of the SMTP/5322 scheme.

1. SMTP cannot transmit executable files or other binary objects. A number of schemes are in use for converting binary files into a text form that can be used by SMTP mail systems, including the popular UNIX UUencode/UUdecode scheme. However, none of these is a standard or even a *de facto* standard.
2. SMTP cannot transmit text data that includes national language characters, because these are represented by 8-bit codes with values of 128 decimal or higher, and SMTP is limited to 7-bit ASCII.

3. SMTP servers may reject mail message over a certain size.
4. SMTP gateways that translate between ASCII and the character code EBCDIC do not use a consistent set of mappings, resulting in translation problems.
5. SMTP gateways to X.400 electronic mail networks cannot handle nontextual data included in X.400 messages.
6. Some SMTP implementations do not adhere completely to the SMTP standards defined in RFC 821. Common problems include:
   - Deletion, addition, or reordering of carriage return and linefeed
   - Truncating or wrapping lines longer than 76 characters
   - Removal of trailing white space (tab and space characters)
   - Padding of lines in a message to the same length
   - Conversion of tab characters into multiple space characters

MIME is intended to resolve these problems in a manner that is compatible with existing RFC 5322 implementations. The specification is provided in RFCs 2045 through 2049.

Overview The MIME specification includes the following elements.

1. Five new message header fields are defined, which may be included in an RFC 5322 header. These fields provide information about the body of the message.
2. A number of content formats are defined, thus standardizing representations that support multimedia electronic mail.
3. Transfer encodings are defined that enable the conversion of any content format into a form that is protected from alteration by the mail system.

In this subsection, we introduce the five message header fields. The next two subsections deal with content formats and transfer encodings.

The five header fields defined in MIME are

- **MIME-Version:** Must have the parameter value 1.0. This field indicates that the message conforms to RFCs 2045 and 2046.
- **Content-Type:** Describes the data contained in the body with sufficient detail that the receiving user agent can pick an appropriate agent or mechanism to represent the data to the user or otherwise deal with the data in an appropriate manner.
- **Content-Transfer-Encoding:** Indicates the type of transformation that has been used to represent the body of the message in a way that is acceptable for mail transport.
- **Content-ID:** Used to identify MIME entities uniquely in multiple contexts.
- **Content-Description:** A text description of the object with the body; this is useful when the object is not readable (e.g., audio data).

Any or all of these fields may appear in a normal RFC 5322 header. A compliant implementation must support the MIME-Version, Content-Type, and Content-Transfer-Encoding fields; the Content-ID and Content-Description fields are optional and may be ignored by the recipient implementation.

MIME Content Types The bulk of the MIME specification is concerned with the definition of a variety of content types. This reflects the need to provide standardized ways of dealing with a wide variety of information representations in a multimedia environment.

Table 18.3 lists the content types specified in RFC 2046. There are seven different major types of content and a total of 15 subtypes. In general, a content type declares the general type of data, and the subtype specifies a particular format for that type of data.

For the **text type** of body, no special software is required to get the full meaning of the text aside from support of the indicated character set. The primary subtype is *plain text*, which is simply a string of ASCII characters or ISO 8859 characters. The *enriched* subtype allows greater formatting flexibility.

The **multipart type** indicates that the body contains multiple, independent parts. The Content-Type header field includes a parameter (called a boundary) that defines the delimiter between body parts. This boundary should not appear in any parts of the message. Each boundary starts on a new line and consists of two hyphens followed by the boundary value. The final boundary, which indicates the end of the last part, also has a suffix of two hyphens. Within each part, there may be an optional ordinary MIME header.

Table 18.3 MIME Content Types

Type	Subtype	Description
Text	Plain	Unformatted text; may be ASCII or ISO 8859.
	Enriched	Provides greater format flexibility.
Multipart	Mixed	The different parts are independent but are to be transmitted together. They should be presented to the receiver in the order that they appear in the mail message.
	Parallel	Differs from Mixed only in that no order is defined for delivering the parts to the receiver.
	Alternative	The different parts are alternative versions of the same information. They are ordered in increasing faithfulness to the original, and the recipient's mail system should display the "best" version to the user.
	Digest	Similar to Mixed, but the default type/subtype of each part is message/rfc822.
Message	rfc822	The body is itself an encapsulated message that conforms to RFC 822.
	Partial	Used to allow fragmentation of large mail items, in a way that is transparent to the recipient.
	External-body	Contains a pointer to an object that exists elsewhere.
Image	jpeg	The image is in JPEG format, JFIF encoding.
	gif	The image is in GIF format.
Video	mpeg	MPEG format.
Audio	Basic	Single-channel 8-bit ISDN mu-law encoding at a sample rate of 8 kHz.
Application	PostScript	Adobe Postscript format.
	octet-stream	General binary data consisting of 8-bit bytes.

Here is a simple example of a multipart message containing two parts—both consisting of simple text (taken from RFC 2046).

```
From: Nathaniel Borenstein <nsb@bellcore.com>
To: Ned Freed <ned@innosoft.com>
Subject: Sample message
MIME-Version: 1.0
Content-type: multipart/mixed; boundary="simple
boundary"

This is the preamble. It is to be ignored, though it
is a handy place for mail composers to include an
explanatory note to non-MIME conformant readers.
—simple boundary

This is implicitly typed plain ASCII text. It does NOT
end with a linebreak.
—simple boundary
Content-type: text/plain; charset=us-ascii

This is explicitly typed plain ASCII text. It DOES end
with a linebreak.

—simple boundary—
This is the epilogue. It is also to be ignored.
```

There are four subtypes of the multipart type, all of which have the same overall syntax. The **multipart/mixed subtype** is used when there are multiple independent body parts that need to be bundled in a particular order. For the **multipart/parallel subtype,** the order of the parts is not significant. If the recipient's system is appropriate, the multiple parts can be presented in parallel. For example, a picture or text part could be accompanied by a voice commentary that is played while the picture or text is displayed.

For the **multipart/alternative subtype,** the various parts are different representations of the same information. The following is an example:

```
From: Nathaniel Borenstein <nsb@bellcore.com>
To: Ned Freed <ned@innosoft.com>
Subject: Formatted text mail
MIME-Version: 1.0
Content-Type: multipart/alternative;
boundary=boundary42

   —boundary42

Content-Type: text/plain; charset=us-ascii

   ...plain text version of message goes here....
```

```
—boundary42
Content-Type: text/enriched

   .... RFC 1896 text/enriched version of same message
goes here ...

—boundary42—
```

In this subtype, the body parts are ordered in terms of increasing preference. For this example, if the recipient system is capable of displaying the message in the text/enriched format, this is done; otherwise, the plain text format is used.

The **multipart/digest subtype** is used when each of the body parts is interpreted as an RFC 5322 message with headers. This subtype enables the construction of a message whose parts are individual messages. For example, the moderator of a group might collect e-mail messages from participants, bundle these messages, and send them out in one encapsulating MIME message.

The **message type** provides a number of important capabilities in MIME. The **message/rfc822 subtype** indicates that the body is an entire message, including header and body. Despite the name of this subtype, the encapsulated message may be not only a simple RFC 5322 message but also any MIME message.

The **message/partial subtype** enables fragmentation of a large message into a number of parts, which must be reassembled at the destination. For this subtype, three parameters are specified in the Content-Type: Message/Partial field: an *id* common to all fragments of the same message, a *sequence number* unique to each fragment, and the *total* number of fragments.

The **message/external-body subtype** indicates that the actual data to be conveyed in this message are not contained in the body. Instead, the body contains the information needed to access the data. As with the other message types, the message/external-body subtype has an outer header and an encapsulated message with its own header. The only necessary field in the outer header is the Content-Type field, which identifies this as a message/external-body subtype. The inner header is the message header for the encapsulated message. The Content-Type field in the outer header must include an access-type parameter, which indicates the method of access, such as FTP (file transfer protocol).

The **application type** refers to other kinds of data, typically either uninterpreted binary data or information to be processed by a mail-based application.

MIME Transfer Encodings The other major component of the MIME specification, in addition to content type specification, is a definition of transfer encodings for message bodies. The objective is to provide reliable delivery across the largest range of environments.

The MIME standard defines two methods of encoding data. The Content-Transfer-Encoding field can actually take on six values, as listed in Table 18.4. However, three of these values (7bit, 8bit, and binary) indicate that no encoding has been done but provide some information about the nature of the data. For SMTP transfer, it is safe to use the 7bit form. The 8bit and binary forms may be usable in other mail transport contexts. Another Content-Transfer-Encoding value is x-token,

Table 18.4 MIME Transfer Encodings

7bit	The data are all represented by short lines of ASCII characters.
8bit	The lines are short, but there may be non-ASCII characters (octets with the high-order bit set).
binary	Not only may non-ASCII characters be present, but the lines are not necessarily short enough for SMTP transport.
quoted-printable	Encodes the data in such a way that if the data being encoded are mostly ASCII text, the encoded form of the data remains largely recognizable by humans.
base64	Encodes data by mapping 6-bit blocks of input to 8-bit blocks of output, all of which are printable ASCII characters.
x-token	A named nonstandard encoding.

which indicates that some other encoding scheme is used for which a name is to be supplied. This could be a vendor-specific or application-specific scheme. The two actual encoding schemes defined are quoted-printable and base64. Two schemes are defined to provide a choice between a transfer technique that is essentially human readable and one that is safe for all types of data in a way that is reasonably compact.

The **quoted-printable** transfer encoding is useful when the data consists largely of octets that correspond to printable ASCII characters. In essence, it represents nonsafe characters by the hexadecimal representation of their code and introduces reversible (soft) line breaks to limit message lines to 76 characters.

The **base64 transfer encoding,** also known as radix-64 encoding, is a common one for encoding arbitrary binary data in such a way as to be invulnerable to the processing by mail-transport programs. It is also used in PGP and is described in Appendix 18A.

A Multipart Example Figure 18.8, taken from RFC 2045, is the outline of a complex multipart message. The message has five parts to be displayed serially: two introductory plain text parts, an embedded multipart message, a richtext part, and a closing encapsulated text message in a non-ASCII character set. The embedded multipart message has two parts to be displayed in parallel: a picture and an audio fragment.

Canonical Form An important concept in MIME and S/MIME is that of canonical form. Canonical form is a format, appropriate to the content type, that is standardized for use between systems. This is in contrast to native form, which is a format that may be peculiar to a particular system. Table 18.5, from RFC 2049, should help clarify this matter.

S/MIME Functionality

In terms of general functionality, S/MIME is very similar to PGP. Both offer the ability to sign and/or encrypt messages. In this subsection, we briefly summarize S/MIME capability. We then look in more detail at this capability by examining message formats and message preparation.

```
MIME-Version: 1.0
From: Nathaniel Borenstein <nsb@bellcore.com>
To: Ned Freed <ned@innosoft.com>
Subject: A multipart example
Content-Type: multipart/mixed;
     boundary=unique-boundary-1

This is the preamble area of a multipart message. Mail readers that understand multipart format should ignore
this preamble. If you are reading this text, you might want to consider changing to a mail reader that understands
how to properly display multipart messages.

--unique-boundary-1

    ...Some text appears here...
 [Note that the preceding blank line means no header fields were given and this is text, with charset US ASCII.
It could have been done with explicit typing as in the next part.]

--unique-boundary-1
Content-type: text/plain; charset=US-ASCII

This could have been part of the previous part, but illustrates explicit versus implicit typing of body parts.

--unique-boundary-1
Content-Type: multipart/parallel;   boundary=unique-boundary-2

--unique-boundary-2
Content-Type: audio/basic
Content-Transfer-Encoding: base64

    ... base64-encoded 8000 Hz single-channel mu-law-format audio data goes here....

--unique-boundary-2
Content-Type: image/jpeg
Content-Transfer-Encoding: base64

    ... base64-encoded image data goes here....

--unique-boundary-2--

--unique-boundary-1
Content-type: text/enriched

This is <bold><italic>richtext.</italic></bold> <smaller>as defined in RFC 1896</smaller>

Isn't it <bigger><bigger>cool?</bigger></bigger>

--unique-boundary-1
Content-Type: message/rfc822

From: (mailbox in US-ASCII)
To: (address in US-ASCII)
Subject: (subject in US-ASCII)
Content-Type: Text/plain; charset=ISO-8859-1
Content-Transfer-Encoding: Quoted-printable

    ... Additional text in ISO-8859-1 goes here ...

--unique-boundary-1--
```

Figure 18.8 Example MIME Message Structure