

# TRANSPORT-LEVEL SECURITY

## 5.1 Web Security Considerations

- Web Security Threats
- Web Traffic Security Approaches

## 5.2 Secure Socket Layer and Transport Layer Security

- SSL Architecture
- SSL Record Protocol
- Change Cipher Spec Protocol
- Alert Protocol
- Handshake Protocol
- Cryptographic Computations

## 5.3 Transport Layer Security

- Version Number
- Message Authentication Code
- Pseudorandom Function
- Alert Codes
- Cipher Suites
- Client Certificate Types
- Certificate\_Verify and Finished Messages
- Cryptographic Computations
- Padding

## 5.4 HTTPS

- Connection Initiation
- Connection Closure

## 5.5 Secure Shell (SSH)

- Transport Layer Protocol
- User Authentication Protocol
- Connection Protocol

## 5.6 Recommended Reading and Web Sites

## 5.7 Key Terms, Review Questions, and Problems

*Wake your mentality  
Wake up to reality*

—From the song, “I’ve Got You Under My Skin” by Cole Porter

### KEY POINTS

- ◆ Secure Socket Layer (SSL) provides security services between TCP and applications that use TCP. The Internet standard version is called Transport Layer Service (TLS).
- ◆ SSL/TLS provides confidentiality using symmetric encryption and message integrity using a message authentication code.
- ◆ SSL/TLS includes protocol mechanisms to enable two TCP users to determine the security mechanisms and services they will use.
- ◆ HTTPS (HTTP over SSL) refers to the combination of HTTP and SSL to implement secure communication between a Web browser and a Web server.
- ◆ Secure Shell (SSH) provides secure remote logon and other secure client/server facilities.

Virtually all businesses, most government agencies, and many individuals now have Web sites. The number of individuals and companies with Internet access is expanding rapidly and all of these have graphical Web browsers. As a result, businesses are enthusiastic about setting up facilities on the Web for electronic commerce. But the reality is that the Internet and the Web are extremely vulnerable to compromises of various sorts. As businesses wake up to this reality, the demand for secure Web services grows.

The topic of Web security is a broad one and can easily fill a book. In this chapter, we begin with a discussion of the general requirements for Web security and then focus on three standardized schemes that are becoming increasingly important as part of Web commerce and that focus on security at the transport layer: SSL/TLS, HTTPS, and SSH.

## 5.1 WEB SECURITY CONSIDERATIONS

The World Wide Web is fundamentally a client/server application running over the Internet and TCP/IP intranets. As such, the security tools and approaches discussed so far in this book are relevant to the issue of Web security. But, as pointed out in [GARF02], the Web presents new challenges not generally appreciated in the context of computer and network security.

- The Internet is two-way. Unlike traditional publishing environments—even electronic publishing systems involving teletext, voice response, or fax-back—the Web is vulnerable to attacks on the Web servers over the Internet.

- The Web is increasingly serving as a highly visible outlet for corporate and product information and as the platform for business transactions. Reputations can be damaged and money can be lost if the Web servers are subverted.
- Although Web browsers are very easy to use, Web servers are relatively easy to configure and manage, and Web content is increasingly easy to develop, the underlying software is extraordinarily complex. This complex software may hide many potential security flaws. The short history of the Web is filled with examples of new and upgraded systems, properly installed, that are vulnerable to a variety of security attacks.
- A Web server can be exploited as a launching pad into the corporation's or agency's entire computer complex. Once the Web server is subverted, an attacker may be able to gain access to data and systems not part of the Web itself but connected to the server at the local site.
- Casual and untrained (in security matters) users are common clients for Web-based services. Such users are not necessarily aware of the security risks that exist and do not have the tools or knowledge to take effective countermeasures.

### Web Security Threats

Table 5.1 provides a summary of the types of security threats faced when using the Web. One way to group these threats is in terms of passive and active attacks. Passive attacks include eavesdropping on network traffic between browser and server and gaining access to information on a Web site that is supposed to be restricted. Active attacks include impersonating another user, altering messages in transit between client and server, and altering information on a Web site.

Another way to classify Web security threats is in terms of the location of the threat: Web server, Web browser, and network traffic between browser and server. Issues of server and browser security fall into the category of computer system security; Part Four of this book addresses the issue of system security in general but is also applicable to Web system security. Issues of traffic security fall into the category of network security and are addressed in this chapter.

### Web Traffic Security Approaches

A number of approaches to providing Web security are possible. The various approaches that have been considered are similar in the services they provide and, to some extent, in the mechanisms that they use, but they differ with respect to their scope of applicability and their relative location within the TCP/IP protocol stack.

Figure 5.1 illustrates this difference. One way to provide Web security is to use IP security (IPsec) (Figure 5.1a). The advantage of using IPsec is that it is transparent to end users and applications and provides a general-purpose solution. Furthermore, IPsec includes a filtering capability so that only selected traffic need incur the overhead of IPsec processing.

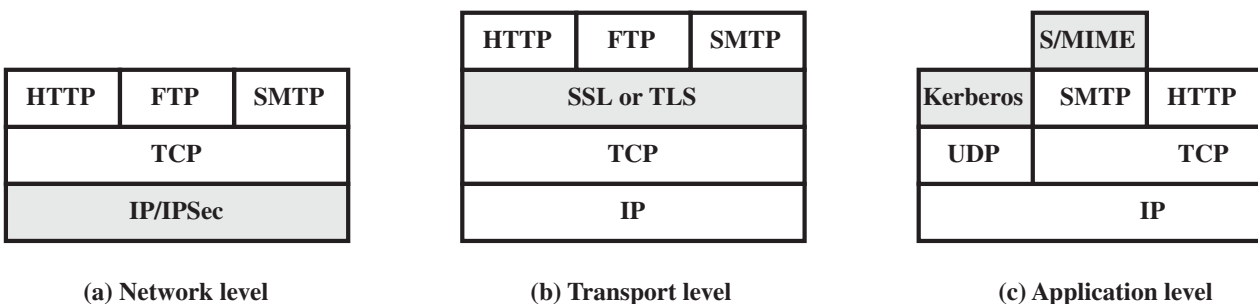
Another relatively general-purpose solution is to implement security just above TCP (Figure 5.1b). The foremost example of this approach is the Secure

**Table 5.1** A Comparison of Threats on the Web

	Threats	Consequences	Countermeasures
<b>Integrity</b>	<ul style="list-style-type: none"> <li>• Modification of user data</li> <li>• Trojan horse browser</li> <li>• Modification of memory</li> <li>• Modification of message traffic in transit</li> </ul>	<ul style="list-style-type: none"> <li>• Loss of information</li> <li>• Compromise of machine</li> <li>• Vulnerability to all other threats</li> </ul>	Cryptographic checksums
<b>Confidentiality</b>	<ul style="list-style-type: none"> <li>• Eavesdropping on the net</li> <li>• Theft of info from server</li> <li>• Theft of data from client</li> <li>• Info about network configuration</li> <li>• Info about which client talks to server</li> </ul>	<ul style="list-style-type: none"> <li>• Loss of information</li> <li>• Loss of privacy</li> </ul>	Encryption, Web proxies
<b>Denial of Service</b>	<ul style="list-style-type: none"> <li>• Killing of user threads</li> <li>• Flooding machine with bogus requests</li> <li>• Filling up disk or memory</li> <li>• Isolating machine by DNS attacks</li> </ul>	<ul style="list-style-type: none"> <li>• Disruptive</li> <li>• Annoying</li> <li>• Prevent user from getting work done</li> </ul>	Difficult to prevent
<b>Authentication</b>	<ul style="list-style-type: none"> <li>• Impersonation of legitimate users</li> <li>• Data forgery</li> </ul>	<ul style="list-style-type: none"> <li>• Misrepresentation of user</li> <li>• Belief that false information is valid</li> </ul>	Cryptographic techniques

Sockets Layer (SSL) and the follow-on Internet standard known as Transport Layer Security (TLS). At this level, there are two implementation choices. For full generality, SSL (or TLS) could be provided as part of the underlying protocol suite and therefore be transparent to applications. Alternatively, SSL can be embedded in specific packages. For example, Netscape and Microsoft Explorer browsers come equipped with SSL, and most Web servers have implemented the protocol.

Application-specific security services are embedded within the particular application. Figure 5.1c shows examples of this architecture. The advantage of this approach is that the service can be tailored to the specific needs of a given application.

**Figure 5.1** Relative Location of Security Facilities in the TCP/IP Protocol Stack

## 5.2 SECURE SOCKET LAYER AND TRANSPORT LAYER SECURITY

Netscape originated SSL. Version 3 of the protocol was designed with public review and input from industry and was published as an Internet draft document. Subsequently, when a consensus was reached to submit the protocol for Internet standardization, the TLS working group was formed within IETF to develop a common standard. This first published version of TLS can be viewed as essentially an SSLv3.1 and is very close to and backward compatible with SSLv3.

This section is devoted to a discussion of SSLv3. In the next section, the principal differences between SSLv3 and TLS are described.

### SSL Architecture

SSL is designed to make use of TCP to provide a reliable end-to-end secure service. SSL is not a single protocol but rather two layers of protocols, as illustrated in Figure 5.2.

The SSL Record Protocol provides basic security services to various higher-layer protocols. In particular, the Hypertext Transfer Protocol (HTTP), which provides the transfer service for Web client/server interaction, can operate on top of SSL. Three higher-layer protocols are defined as part of SSL: the Handshake Protocol, The Change Cipher Spec Protocol, and the Alert Protocol. These SSL-specific protocols are used in the management of SSL exchanges and are examined later in this section.

Two important SSL concepts are the SSL session and the SSL connection, which are defined in the specification as follows.

- **Connection:** A connection is a transport (in the OSI layering model definition) that provides a suitable type of service. For SSL, such connections are peer-to-peer relationships. The connections are transient. Every connection is associated with one session.
- **Session:** An SSL session is an association between a client and a server. Sessions are created by the Handshake Protocol. Sessions define a set of cryptographic

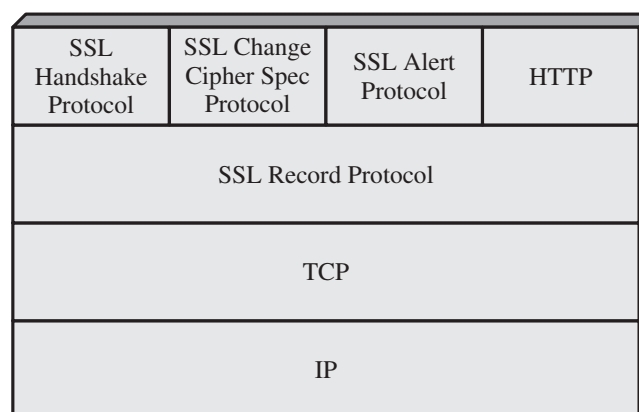


Figure 5.2 SSL Protocol Stack

security parameters which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.

Between any pair of parties (applications such as HTTP on client and server), there may be multiple secure connections. In theory, there may also be multiple simultaneous sessions between parties, but this feature is not used in practice.

There are a number of states associated with each session. Once a session is established, there is a current operating state for both read and write (i.e., receive and send). In addition, during the Handshake Protocol, pending read and write states are created. Upon successful conclusion of the Handshake Protocol, the pending states become the current states.

A session state is defined by the following parameters.

- **Session identifier:** An arbitrary byte sequence chosen by the server to identify an active or resumable session state.
- **Peer certificate:** An X509.v3 certificate of the peer. This element of the state may be null.
- **Compression method:** The algorithm used to compress data prior to encryption.
- **Cipher spec:** Specifies the bulk data encryption algorithm (such as null, AES, etc.) and a hash algorithm (such as MD5 or SHA-1) used for MAC calculation. It also defines cryptographic attributes such as the `hash_size`.
- **Master secret:** 48-byte secret shared between the client and server.
- **Is resumable:** A flag indicating whether the session can be used to initiate new connections.

A connection state is defined by the following parameters.

- **Server and client random:** Byte sequences that are chosen by the server and client for each connection.
- **Server write MAC secret:** The secret key used in MAC operations on data sent by the server.
- **Client write MAC secret:** The secret key used in MAC operations on data sent by the client.
- **Server write key:** The secret encryption key for data encrypted by the server and decrypted by the client.
- **Client write key:** The symmetric encryption key for data encrypted by the client and decrypted by the server.
- **Initialization vectors:** When a block cipher in CBC mode is used, an initialization vector (IV) is maintained for each key. This field is first initialized by the SSL Handshake Protocol. Thereafter, the final ciphertext block from each record is preserved for use as the IV with the following record.
- **Sequence numbers:** Each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a change cipher spec message, the appropriate sequence number is set to zero. Sequence numbers may not exceed  $2^{64} - 1$ .

## SSL Record Protocol

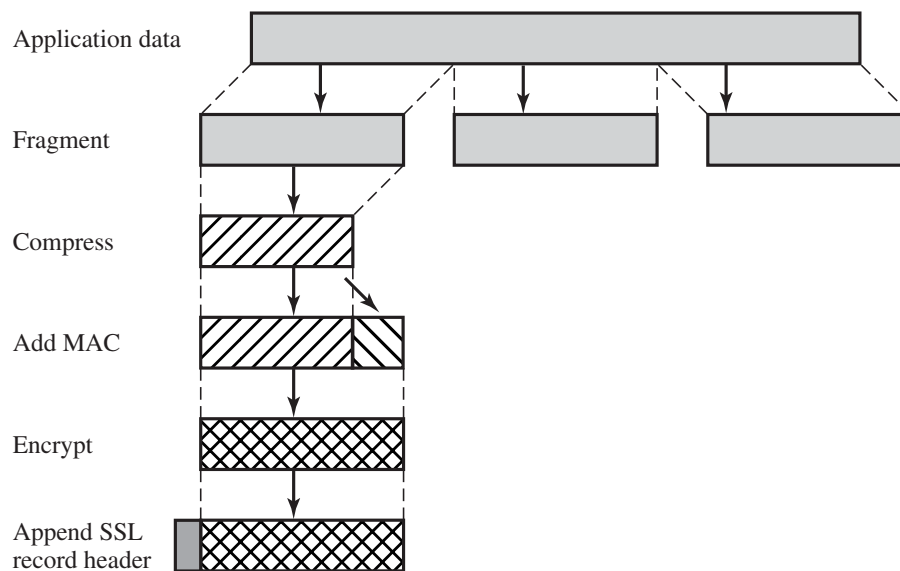
The SSL Record Protocol provides two services for SSL connections:

- **Confidentiality:** The Handshake Protocol defines a shared secret key that is used for conventional encryption of SSL payloads.
- **Message Integrity:** The Handshake Protocol also defines a shared secret key that is used to form a message authentication code (MAC).

Figure 5.3 indicates the overall operation of the SSL Record Protocol. The Record Protocol takes an application message to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, adds a header, and transmits the resulting unit in a TCP segment. Received data are decrypted, verified, decompressed, and reassembled before being delivered to higher-level users.

The first step is **fragmentation**. Each upper-layer message is fragmented into blocks of  $2^{14}$  bytes (16384 bytes) or less. Next, **compression** is optionally applied. Compression must be lossless and may not increase the content length by more than 1024 bytes.<sup>1</sup> In SSLv3 (as well as the current version of TLS), no compression algorithm is specified, so the default compression algorithm is null.

The next step in processing is to compute a **message authentication code** over the compressed data. For this purpose, a shared secret key is used. The calculation is defined as



**Figure 5.3** SSL Record Protocol Operation

<sup>1</sup>Of course, one hopes that compression shrinks rather than expands the data. However, for very short blocks, it is possible, because of formatting conventions, that the compression algorithm will actually provide output that is longer than the input.



```

hash(MAC_write_secret || pad_2 ||
     hash(MAC_write_secret || pad_1 || seq_num ||
          SSLCompressed.type || SSLCompressed.length ||
          SSLCompressed.fragment))

```

where

	= concatenation
MAC_write_secret	= shared secret key
hash	= cryptographic hash algorithm; either MD5 or SHA-1
pad_1	= the byte 0x36 (0011 0110) repeated 48 times (384 bits) for MD5 and 40 times (320 bits) for SHA-1
pad_2	= the byte 0x5C (0101 1100) repeated 48 times for MD5 and 40 times for SHA-1
seq_num	= the sequence number for this message
SSLCompressed.type	= the higher-level protocol used to process this fragment
SSLCompressed.length	= the length of the compressed fragment
SSLCompressed.fragment	= the compressed fragment (if compression is not used, this is the plaintext fragment)

Note that this is very similar to the HMAC algorithm defined in Chapter 3. The difference is that the two pads are concatenated in SSLv3 and are XORed in HMAC. The SSLv3 MAC algorithm is based on the original Internet draft for HMAC, which used concatenation. The final version of HMAC (defined in RFC 2104) uses the XOR.

Next, the compressed message plus the MAC are **encrypted** using symmetric encryption. Encryption may not increase the content length by more than 1024 bytes, so that the total length may not exceed  $2^{14} + 2048$ . The following encryption algorithms are permitted:

Block Cipher		Stream Cipher	
Algorithm	Key Size	Algorithm	Key Size
AES	128, 256	RC4-40	40
IDEA	128	RC4-128	128
RC2-40	40		
DES-40	40		
DES	56		
3DES	168		
Fortezza	80		

Fortezza can be used in a smart card encryption scheme.

For stream encryption, the compressed message plus the MAC are encrypted. Note that the MAC is computed before encryption takes place and that the MAC is then encrypted along with the plaintext or compressed plaintext.

For block encryption, padding may be added after the MAC prior to encryption. The padding is in the form of a number of padding bytes followed by a one-byte



indication of the length of the padding. The total amount of padding is the smallest amount such that the total size of the data to be encrypted (plaintext plus MAC plus padding) is a multiple of the cipher's block length. An example is a plaintext (or compressed text if compression is used) of 58 bytes, with a MAC of 20 bytes (using SHA-1), that is encrypted using a block length of 8 bytes (e.g., DES). With the padding-length byte, this yields a total of 79 bytes. To make the total an integer multiple of 8, one byte of padding is added.

The final step of SSL Record Protocol processing is to prepare a header consisting of the following fields:

- **Content Type (8 bits):** The higher-layer protocol used to process the enclosed fragment.
- **Major Version (8 bits):** Indicates major version of SSL in use. For SSLv3, the value is 3.
- **Minor Version (8 bits):** Indicates minor version in use. For SSLv3, the value is 0.
- **Compressed Length (16 bits):** The length in bytes of the plaintext fragment (or compressed fragment if compression is used). The maximum value is  $2^{14} + 2048$ .

The content types that have been defined are `change_cipher_spec`, `alert`, `handshake`, and `application_data`. The first three are the SSL-specific protocols, discussed next. Note that no distinction is made among the various applications (e.g., HTTP) that might use SSL; the content of the data created by such applications is opaque to SSL.

Figure 5.4 illustrates the SSL record format.

### Change Cipher Spec Protocol

The Change Cipher Spec Protocol is one of the three SSL-specific protocols that use the SSL Record Protocol, and it is the simplest. This protocol consists of a single message (Figure 5.5a), which consists of a single byte with the value 1. The sole purpose of this message is to cause the pending state to be copied into the current state, which updates the cipher suite to be used on this connection.

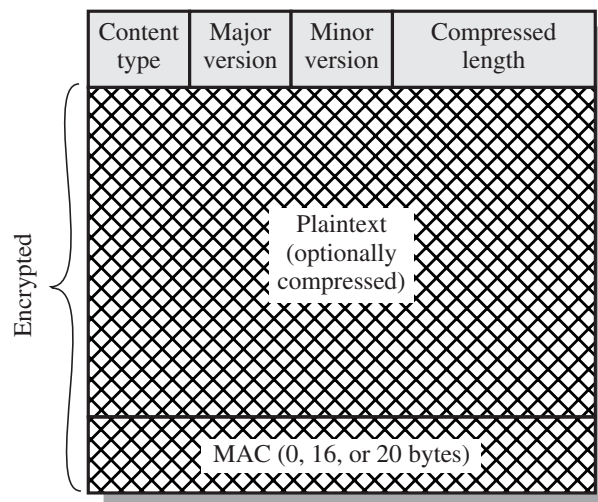
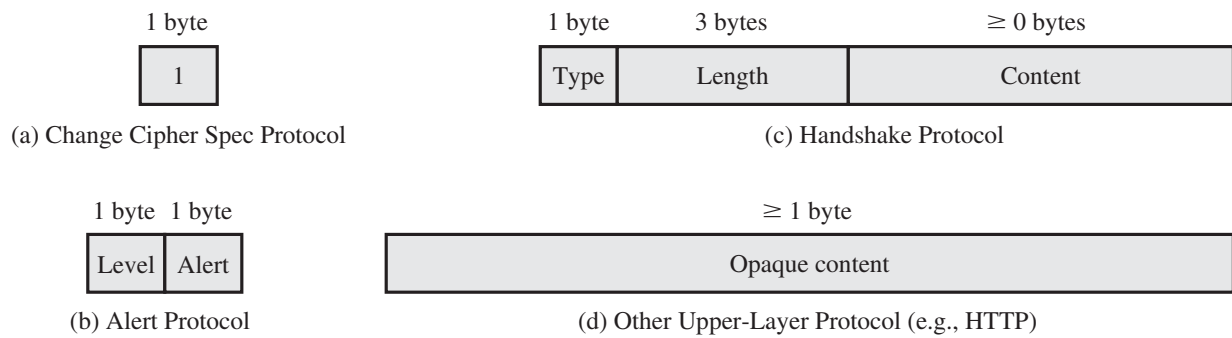


Figure 5.4 SSL Record Format



**Figure 5.5** SSL Record Protocol Payload

## Alert Protocol

The Alert Protocol is used to convey SSL-related alerts to the peer entity. As with other applications that use SSL, alert messages are compressed and encrypted, as specified by the current state.

Each message in this protocol consists of two bytes (Figure 5.5b). The first byte takes the value warning (1) or fatal (2) to convey the severity of the message. If the level is fatal, SSL immediately terminates the connection. Other connections on the same session may continue, but no new connections on this session may be established. The second byte contains a code that indicates the specific alert. First, we list those alerts that are always fatal (definitions from the SSL specification):

- **unexpected\_message:** An inappropriate message was received.
- **bad\_record\_mac:** An incorrect MAC was received.
- **decompression\_failure:** The decompression function received improper input (e.g., unable to decompress or decompress to greater than maximum allowable length).
- **handshake\_failure:** Sender was unable to negotiate an acceptable set of security parameters given the options available.
- **illegal\_parameter:** A field in a handshake message was out of range or inconsistent with other fields.

The remaining alerts are the following.

- **close\_notify:** Notifies the recipient that the sender will not send any more messages on this connection. Each party is required to send a **close\_notify** alert before closing the write side of a connection.
- **no\_certificate:** May be sent in response to a certificate request if no appropriate certificate is available.
- **bad\_certificate:** A received certificate was corrupt (e.g., contained a signature that did not verify).
- **unsupported\_certificate:** The type of the received certificate is not supported.
- **certificate\_revoked:** A certificate has been revoked by its signer.
- **certificate\_expired:** A certificate has expired.

- **certificate\_unknown:** Some other unspecified issue arose in processing the certificate, rendering it unacceptable.

## Handshake Protocol

The most complex part of SSL is the Handshake Protocol. This protocol allows the server and client to authenticate each other and to negotiate an encryption and MAC algorithm and cryptographic keys to be used to protect data sent in an SSL record. The Handshake Protocol is used before any application data is transmitted.

The Handshake Protocol consists of a series of messages exchanged by client and server. All of these have the format shown in Figure 5.5c. Each message has three fields:

- **Type (1 byte):** Indicates one of 10 messages. Table 5.2 lists the defined message types.
- **Length (3 bytes):** The length of the message in bytes.
- **Content ( $\geq 0$  bytes):** The parameters associated with this message; these are listed in Table 5.2.

Figure 5.6 shows the initial exchange needed to establish a logical connection between client and server. The exchange can be viewed as having four phases.

**PHASE 1. ESTABLISH SECURITY CAPABILITIES** This phase is used to initiate a logical connection and to establish the security capabilities that will be associated with it. The exchange is initiated by the client, which sends a **client\_hello** message with the following parameters:

- **Version:** The highest SSL version understood by the client.
- **Random:** A client-generated random structure consisting of a 32-bit timestamp and 28 bytes generated by a secure random number generator. These values serve as nonces and are used during key exchange to prevent replay attacks.

**Table 5.2** SSL Handshake Protocol Message Types

Message Type	Parameters
<b>hello_request</b>	null
<b>client_hello</b>	version, random, session id, cipher suite, compression method
<b>server_hello</b>	version, random, session id, cipher suite, compression method
<b>certificate</b>	chain of X.509v3 certificates
<b>server_key_exchange</b>	parameters, signature
<b>certificate_request</b>	type, authorities
<b>server_done</b>	null
<b>certificate_verify</b>	signature
<b>client_key_exchange</b>	parameters, signature
<b>finished</b>	hash value

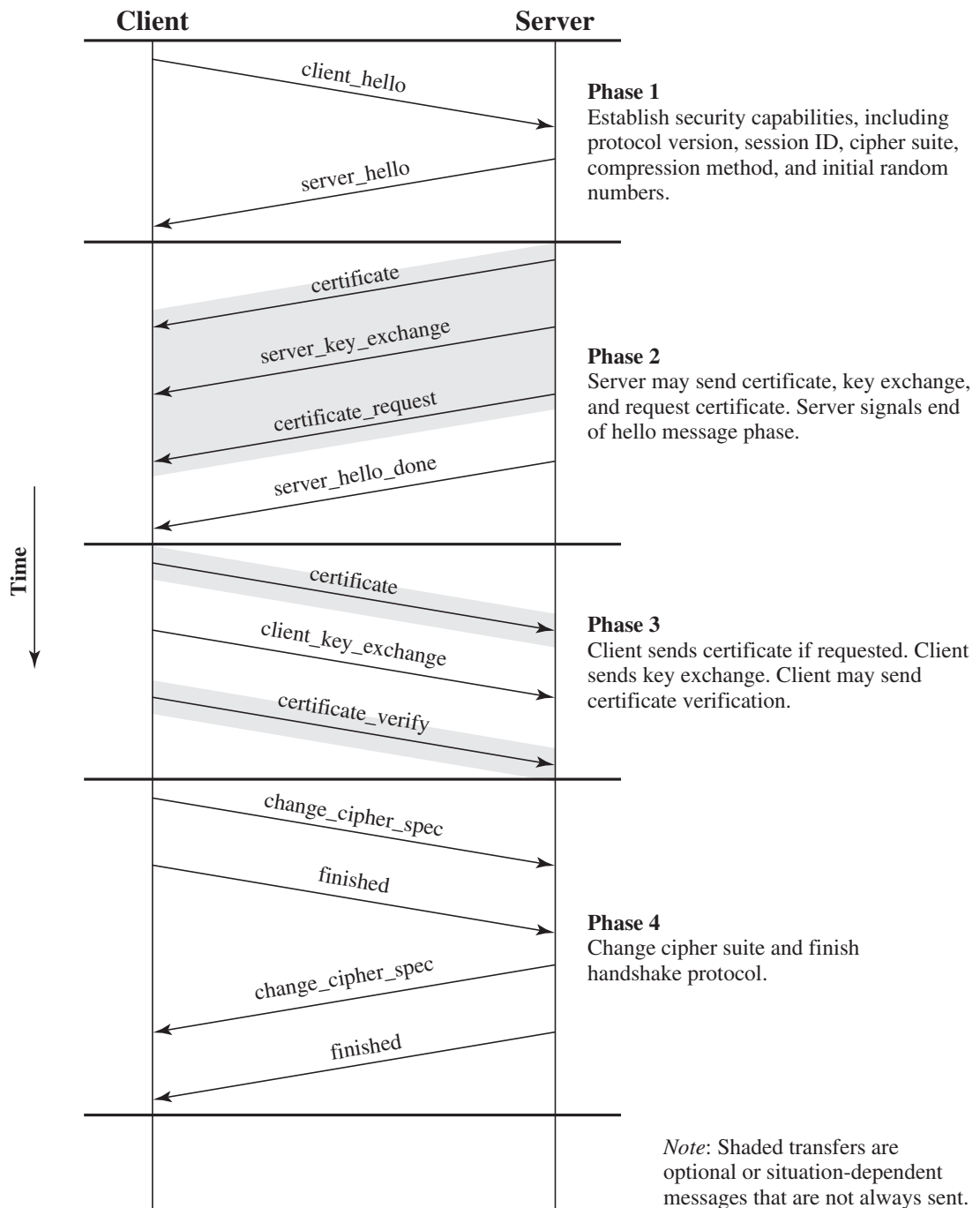


Figure 5.6 Handshake Protocol Action

- **Session ID:** A variable-length session identifier. A nonzero value indicates that the client wishes to update the parameters of an existing connection or to create a new connection on this session. A zero value indicates that the client wishes to establish a new connection on a new session.
- **CipherSuite:** This is a list that contains the combinations of cryptographic algorithms supported by the client, in decreasing order of preference. Each element of the list (each cipher suite) defines both a key exchange algorithm and a CipherSpec; these are discussed subsequently.

- **Compression Method:** This is a list of the compression methods the client supports.

After sending the `client_hello` message, the client waits for the `server_hello` message, which contains the same parameters as the `client_hello` message. For the `server_hello` message, the following conventions apply. The Version field contains the lower of the versions suggested by the client and the highest supported by the server. The Random field is generated by the server and is independent of the client's Random field. If the SessionID field of the client was nonzero, the same value is used by the server; otherwise the server's SessionID field contains the value for a new session. The CipherSuite field contains the single cipher suite selected by the server from those proposed by the client. The Compression field contains the compression method selected by the server from those proposed by the client.

The first element of the CipherSuite parameter is the key exchange method (i.e., the means by which the cryptographic keys for conventional encryption and MAC are exchanged). The following key exchange methods are supported.

- **RSA:** The secret key is encrypted with the receiver's RSA public key. A public-key certificate for the receiver's key must be made available.
- **Fixed Diffie-Hellman:** This is a Diffie-Hellman key exchange in which the server's certificate contains the Diffie-Hellman public parameters signed by the certificate authority (CA). That is, the public-key certificate contains the Diffie-Hellman public-key parameters. The client provides its Diffie-Hellman public-key parameters either in a certificate, if client authentication is required, or in a key exchange message. This method results in a fixed secret key between two peers based on the Diffie-Hellman calculation using the fixed public keys.
- **Ephemeral Diffie-Hellman:** This technique is used to create ephemeral (temporary, one-time) secret keys. In this case, the Diffie-Hellman public keys are exchanged, signed using the sender's private RSA or DSS key. The receiver can use the corresponding public key to verify the signature. Certificates are used to authenticate the public keys. This would appear to be the most secure of the three Diffie-Hellman options, because it results in a temporary, authenticated key.
- **Anonymous Diffie-Hellman:** The base Diffie-Hellman algorithm is used with no authentication. That is, each side sends its public Diffie-Hellman parameters to the other with no authentication. This approach is vulnerable to man-in-the-middle attacks, in which the attacker conducts anonymous Diffie-Hellman with both parties.
- **Fortezza:** The technique defined for the Fortezza scheme.

Following the definition of a key exchange method is the CipherSpec, which includes the following fields.

- **CipherAlgorithm:** Any of the algorithms mentioned earlier: RC4, RC2, DES, 3DES, DES40, IDEA, or Fortezza

- **MACAlgorithm:** MD5 or SHA-1
- **CipherType:** Stream or Block
- **IsExportable:** True or False
- **HashSize:** 0, 16 (for MD5), or 20 (for SHA-1) bytes
- **Key Material:** A sequence of bytes that contain data used in generating the write keys
- **IV Size:** The size of the Initialization Value for Cipher Block Chaining (CBC) encryption

*PHASE 2. SERVER AUTHENTICATION AND KEY EXCHANGE* The server begins this phase by sending its certificate if it needs to be authenticated; the message contains one or a chain of X.509 certificates. The **certificate message** is required for any agreed-on key exchange method except anonymous Diffie-Hellman. Note that if fixed Diffie-Hellman is used, this certificate message functions as the server's key exchange message because it contains the server's public Diffie-Hellman parameters.

Next, a **server\_key\_exchange message** may be sent if it is required. It is not required in two instances: (1) The server has sent a certificate with fixed Diffie-Hellman parameters or (2) a RSA key exchange is to be used. The `server_key_exchange` message is needed for the following:

- **Anonymous Diffie-Hellman:** The message content consists of the two global Diffie-Hellman values (a prime number and a primitive root of that number) plus the server's public Diffie-Hellman key (see Figure 3.12).
- **Ephemeral Diffie-Hellman:** The message content includes the three Diffie-Hellman parameters provided for anonymous Diffie-Hellman plus a signature of those parameters.
- **RSA key exchange (in which the server is using RSA but has a signature-only RSA key):** Accordingly, the client cannot simply send a secret key encrypted with the server's public key. Instead, the server must create a temporary RSA public/private key pair and use the `server_key_exchange` message to send the public key. The message content includes the two parameters of the temporary RSA public key (exponent and modulus; see Figure 3.10) plus a signature of those parameters.
- **Fortezza**

Some further details about the signatures are warranted. As usual, a signature is created by taking the hash of a message and encrypting it with the sender's private key. In this case, the hash is defined as

```
hash(ClientHello.random || ServerHello.random ||
      ServerParams)
```

So the hash covers not only the Diffie-Hellman or RSA parameters but also the two nonces from the initial hello messages. This ensures against replay attacks and misrepresentation. In the case of a DSS signature, the hash is performed using the

SHA-1 algorithm. In the case of an RSA signature, both an MD5 and an SHA-1 hash are calculated, and the concatenation of the two hashes (36 bytes) is encrypted with the server's private key.

Next, a nonanonymous server (server not using anonymous Diffie-Hellman) can request a certificate from the client. The **certificate\_request message** includes two parameters: `certificate_type` and `certificate_authorities`. The certificate type indicates the public-key algorithm and its use:

- RSA, signature only
- DSS, signature only
- RSA for fixed Diffie-Hellman; in this case the signature is used only for authentication, by sending a certificate signed with RSA
- DSS for fixed Diffie-Hellman; again, used only for authentication
- RSA for ephemeral Diffie-Hellman
- DSS for ephemeral Diffie-Hellman
- Fortezza

The second parameter in the `certificate_request` message is a list of the distinguished names of acceptable certificate authorities.

The final message in phase 2, and one that is always required, is the `server_done` message, which is sent by the server to indicate the end of the server hello and associated messages. After sending this message, the server will wait for a client response. This message has no parameters.

**PHASE 3. CLIENT AUTHENTICATION AND KEY EXCHANGE** Upon receipt of the `server_done` message, the client should verify that the server provided a valid certificate (if required) and check that the `server_hello` parameters are acceptable. If all is satisfactory, the client sends one or more messages back to the server.

If the server has requested a certificate, the client begins this phase by sending a **certificate message**. If no suitable certificate is available, the client sends a `no_certificate` alert instead.

Next is the **client\_key\_exchange message**, which must be sent in this phase. The content of the message depends on the type of key exchange, as follows.

- **RSA:** The client generates a 48-byte *pre-master secret* and encrypts with the public key from the server's certificate or temporary RSA key from a `server_key_exchange` message. Its use to compute a *master secret* is explained later.
- **Ephemeral or Anonymous Diffie-Hellman:** The client's public Diffie-Hellman parameters are sent.
- **Fixed Diffie-Hellman:** The client's public Diffie-Hellman parameters were sent in a certificate message, so the content of this message is null.
- **Fortezza:** The client's Fortezza parameters are sent.

Finally, in this phase, the client may send a **certificate\_verify message** to provide explicit verification of a client certificate. This message is only sent following any client certificate that has signing capability (i.e., all certificates except



those containing fixed Diffie-Hellman parameters). This message signs a hash code based on the preceding messages, defined as

```
CertificateVerify.signature.md5_hash=
    MD5(master_secret || pad_2 || MD5(handshake_messages ||
        master_secret || pad_1));
CertificateVerify.signature.sha_hash=
    SHA(master_secret || pad_2 || SHA(handshake_messages ||
        master_secret || pad_1));
```

where `pad_1` and `pad_2` are the values defined earlier for the MAC, **handshake\_messages** refers to all Handshake Protocol messages sent or received starting at `client_hello` but not including this message, and `master_secret` is the calculated secret whose construction is explained later in this section. If the user's private key is DSS, then it is used to encrypt the SHA-1 hash. If the user's private key is RSA, it is used to encrypt the concatenation of the MD5 and SHA-1 hashes. In either case, the purpose is to verify the client's ownership of the private key for the client certificate. Even if someone is misusing the client's certificate, he or she would be unable to send this message.

**PHASE 4. FINISH** This phase completes the setting up of a secure connection. The client sends a `change_cipher_spec` message and copies the pending CipherSpec into the current CipherSpec. Note that this message is not considered part of the Handshake Protocol but is sent using the Change Cipher Spec Protocol. The client then immediately sends the **finished message** under the new algorithms, keys, and secrets. The finished message verifies that the key exchange and authentication processes were successful. The content of the finished message is the concatenation of two hash values:

```
MD5(master_secret || pad2 || MD5(handshake_messages ||
    Sender || master_secret || pad1))
SHA(master_secret || pad2 || SHA(handshake_messages ||
    Sender || master_secret || pad1))
```

where `Sender` is a code that identifies that the sender is the client and `handshake_messages` is all of the data from all handshake messages up to but not including this message.

In response to these two messages, the server sends its own `change_cipher_spec` message, transfers the pending to the current CipherSpec, and sends its finished message. At this point, the handshake is complete and the client and server may begin to exchange application-layer data.

## Cryptographic Computations

Two further items are of interest: (1) the creation of a shared master secret by means of the key exchange and (2) the generation of cryptographic parameters from the master secret.

**MASTER SECRET CREATION** The shared master secret is a one-time 48-byte value (384 bits) generated for this session by means of secure key exchange. The creation is in two stages. First, a `pre_master_secret` is exchanged. Second, the `master_secret` is calculated by both parties. For `pre_master_secret` exchange, there are two possibilities.

- **RSA:** A 48-byte `pre_master_secret` is generated by the client, encrypted with the server's public RSA key, and sent to the server. The server decrypts the ciphertext using its private key to recover the `pre_master_secret`.
- **Diffie-Hellman:** Both client and server generate a Diffie-Hellman public key. After these are exchanged, each side performs the Diffie-Hellman calculation to create the shared `pre_master_secret`.

Both sides now compute the `master_secret` as

```
master_secret = MD5(pre_master_secret || SHA('A' ||
    pre_master_secret || ClientHello.random ||
    ServerHello.random)) ||
    MD5(pre_master_secret || SHA('BB' ||
    pre_master_secret || ClientHello.random ||
    ServerHello.random)) ||
    MD5(pre_master_secret || SHA('CCC' ||
    pre_master_secret || ClientHello.random ||
    ServerHello.random))
```

where `ClientHello.random` and `ServerHello.random` are the two nonce values exchanged in the initial hello messages.

**GENERATION OF CRYPTOGRAPHIC PARAMETERS** CipherSpecs require a client write MAC secret, a server write MAC secret, a client write key, a server write key, a client write IV, and a server write IV, which are generated from the master secret in that order. These parameters are generated from the master secret by hashing the master secret into a sequence of secure bytes of sufficient length for all needed parameters.

The generation of the key material from the master secret uses the same format for generation of the master secret from the pre-master secret as

```
key_block = MD5(master_secret || SHA('A' || master_secret ||
    ServerHello.random || ClientHello.random)) ||
    MD5(master_secret || SHA('BB' || master_secret ||
    ServerHello.random || ClientHello.random)) ||
    MD5(master_secret || SHA('CCC' || master_secret ||
    ServerHello.random || ClientHello.random)) || ...
```

until enough output has been generated. The result of this algorithmic structure is a pseudorandom function. We can view the `master_secret` as the pseudorandom seed value to the function. The client and server random numbers can be viewed as salt values to complicate cryptanalysis (see Chapter 9 for a discussion of the use of salt values).

### 5.3 TRANSPORT LAYER SECURITY

TLS is an IETF standardization initiative whose goal is to produce an Internet standard version of SSL. TLS is defined as a Proposed Internet Standard in RFC 5246. RFC 5246 is very similar to SSLv3. In this section, we highlight the differences.

#### Version Number

The TLS Record Format is the same as that of the SSL Record Format (Figure 5.4), and the fields in the header have the same meanings. The one difference is in version values. For the current version of TLS, the major version is 3 and the minor version is 3.

#### Message Authentication Code

There are two differences between the SSLv3 and TLS MAC schemes: the actual algorithm and the scope of the MAC calculation. TLS makes use of the HMAC algorithm defined in RFC 2104. Recall from Chapter 3 that HMAC is defined as

$$\text{HMAC}_K(M) = H[(K^+ \oplus \text{opad}) \parallel H[(K^+ \oplus \text{ipad}) \parallel M]]$$

where

- $H$  = embedded hash function (for TLS, either MD5 or SHA-1)
- $M$  = message input to HMAC
- $K^+$  = secret key padded with zeros on the left so that the result is equal to the block length of the hash code (for MD5 and SHA-1, block length = 512 bits)
- $\text{ipad}$  = 00110110 (36 in hexadecimal) repeated 64 times (512 bits)
- $\text{opad}$  = 01011100 (5C in hexadecimal) repeated 64 times (512 bits)

SSLv3 uses the same algorithm, except that the padding bytes are concatenated with the secret key rather than being XORed with the secret key padded to the block length. The level of security should be about the same in both cases.

For TLS, the MAC calculation encompasses the fields indicated in the following expression:

```
MAC(MAC_write_secret, seq_num || TLSCompressed.type ||
    TLSCompressed.version || TLSCompressed.length ||
    TLSCompressed.fragment)
```

The MAC calculation covers all of the fields covered by the SSLv3 calculation, plus the field `TLSCompressed.version`, which is the version of the protocol being employed.

## Pseudorandom Function

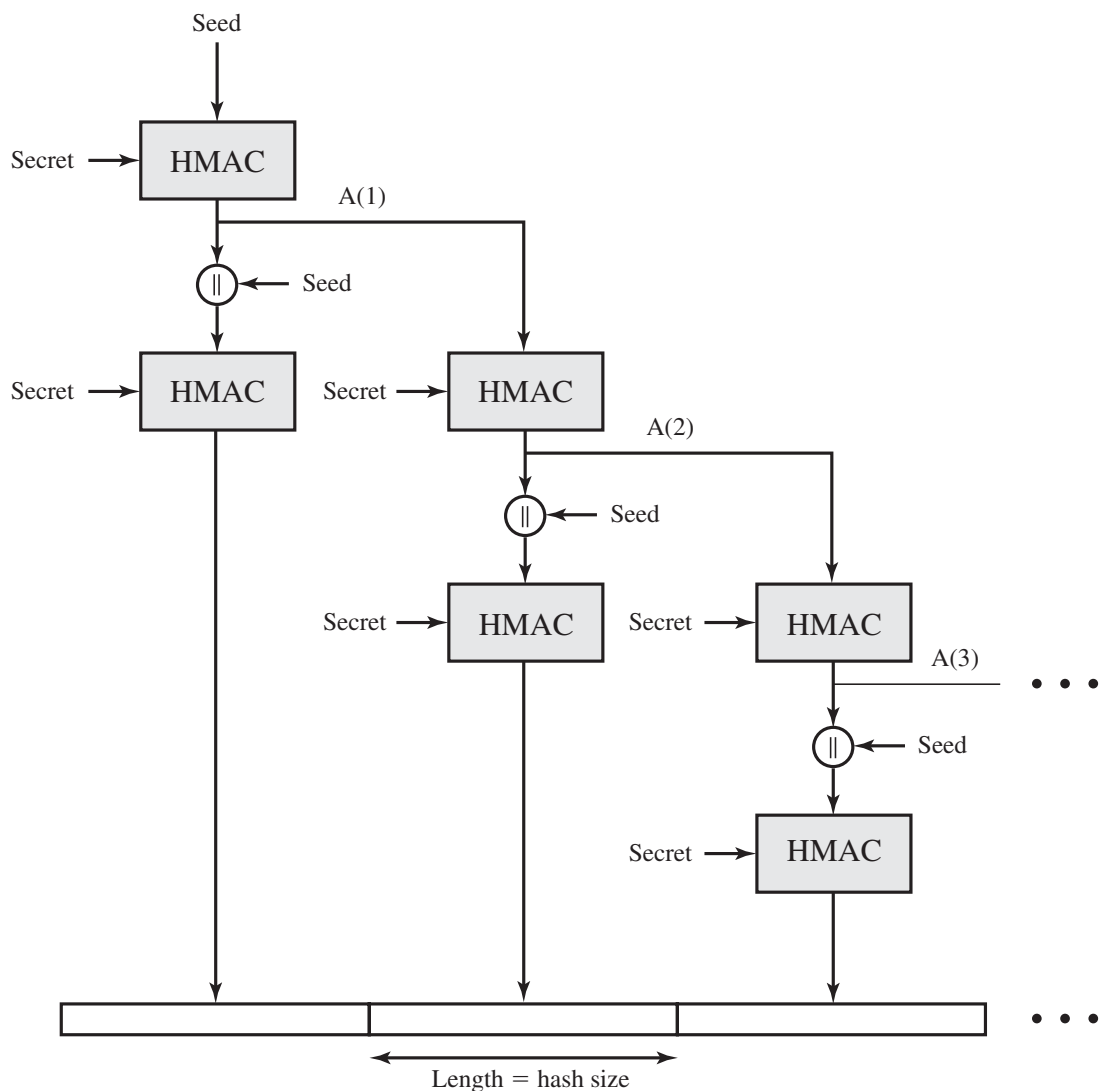
TLS makes use of a pseudorandom function referred to as PRF to expand secrets into blocks of data for purposes of key generation or validation. The objective is to make use of a relatively small shared secret value but to generate longer blocks of data in a way that is secure from the kinds of attacks made on hash functions and MACs. The PRF is based on the data expansion function (Figure 5.7) given as

$$\begin{aligned} P\_hash(secret, seed) = & HMAC\_hash(secret, A(1) \parallel seed) \parallel \\ & HMAC\_hash(secret, A(2) \parallel seed) \parallel \\ & HMAC\_hash(secret, A(3) \parallel seed) \parallel \dots \end{aligned}$$

where  $A()$  is defined as

$$A(0) = seed$$

$$A(i) = HMAC\_hash(secret, A(i-1))$$



**Figure 5.7** TLS Function  $P\_hash(secret, seed)$

The data expansion function makes use of the HMAC algorithm with either MD5 or SHA-1 as the underlying hash function. As can be seen, `P_hash` can be iterated as many times as necessary to produce the required quantity of data. For example, if `P_SHA-1` was used to generate 64 bytes of data, it would have to be iterated four times, producing 80 bytes of data of which the last 16 would be discarded. In this case, `P_MD5` would also have to be iterated four times, producing exactly 64 bytes of data. Note that each iteration involves two executions of HMAC—each of which in turn involves two executions of the underlying hash algorithm.

To make PRF as secure as possible, it uses two hash algorithms in a way that should guarantee its security if either algorithm remains secure. PRF is defined as

$$\text{PRF}(\text{secret}, \text{label}, \text{seed}) = \text{P\_hash}(\text{S1}, \text{label} \parallel \text{seed})$$

PRF takes as input a secret value, an identifying label, and a seed value and produces an output of arbitrary length.

### Alert Codes

TLS supports all of the alert codes defined in SSLv3 with the exception of `no_certificate`. A number of additional codes are defined in TLS; of these, the following are always fatal.

- **record\_overflow:** A TLS record was received with a payload (ciphertext) whose length exceeds  $2^{14}+2048$  bytes, or the ciphertext decrypted to a length of greater than  $2^{14}+1024$  bytes.
- **unknown\_ca:** A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known, trusted CA.
- **access\_denied:** A valid certificate was received, but when access control was applied, the sender decided not to proceed with the negotiation.
- **decode\_error:** A message could not be decoded, because either a field was out of its specified range or the length of the message was incorrect.
- **protocol\_version:** The protocol version the client attempted to negotiate is recognized but not supported.
- **insufficient\_security:** Returned instead of `handshake_failure` when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client.
- **unsupported\_extension:** Sent by clients that receive an extended server hello containing an extension not in the corresponding client hello.
- **internal\_error:** An internal error unrelated to the peer or the correctness of the protocol makes it impossible to continue.
- **decrypt\_error:** A handshake cryptographic operation failed, including being unable to verify a signature, decrypt a key exchange, or validate a finished message.

The remaining alerts include the following.

- **user\_canceled:** This handshake is being canceled for some reason unrelated to a protocol failure.
- **no\_renegotiation:** Sent by a client in response to a hello request or by the server in response to a client hello after initial handshaking. Either of these messages would normally result in renegotiation, but this alert indicates that the sender is not able to renegotiate. This message is always a warning.

## Cipher Suites

There are several small differences between the cipher suites available under SSLv3 and under TLS:

- **Key Exchange:** TLS supports all of the key exchange techniques of SSLv3 with the exception of Fortezza.
- **Symmetric Encryption Algorithms:** TLS includes all of the symmetric encryption algorithms found in SSLv3, with the exception of Fortezza.

## Client Certificate Types

TLS defines the following certificate types to be requested in a `certificate_request` message: `rsa_sign`, `dss_sign`, `rsa_fixed_dh`, and `dss_fixed_dh`. These are all defined in SSLv3. In addition, SSLv3 includes `rsa_ephemeral_dh`, `dss_ephemeral_dh`, and `fortezza_ke`. Ephemeral Diffie-Hellman involves signing the Diffie-Hellman parameters with either RSA or DSS. For TLS, the `rsa_sign` and `dss_sign` types are used for that function; a separate signing type is not needed to sign Diffie-Hellman parameters. TLS does not include the Fortezza scheme.

## Certificate\_Verify and Finished Messages

In the TLS `certificate_verify` message, the MD5 and SHA-1 hashes are calculated only over `handshake_messages`. Recall that for SSLv3, the hash calculation also included the master secret and pads. These extra fields were felt to add no additional security.

As with the finished message in SSLv3, the finished message in TLS is a hash based on the shared `master_secret`, the previous handshake messages, and a label that identifies client or server. The calculation is somewhat different. For TLS, we have

$$\text{PRF}(\text{master\_secret}, \text{finished\_label}, \text{MD5}(\text{handshake\_messages}) || \text{SHA-1}(\text{handshake\_messages}))$$

where `finished_label` is the string “client finished” for the client and “server finished” for the server.

### Cryptographic Computations

The `pre_master_secret` for TLS is calculated in the same way as in SSLv3. As in SSLv3, the `master_secret` in TLS is calculated as a hash function of the `pre_master_secret` and the two hello random numbers. The form of the TLS calculation is different from that of SSLv3 and is defined as

```
master_secret= PRF(pre_master_secret, "master secret",
                   ClientHello.random||ServerHello.random)
```

The algorithm is performed until 48 bytes of pseudorandom output are produced. The calculation of the key block material (MAC secret keys, session encryption keys, and IVs) is defined as

```
key_block = PRF(master_secret, "key expansion",
                 SecurityParameters.server_random||
                 SecurityParameters.client_random)
```

until enough output has been generated. As with SSLv3, the `key_block` is a function of the `master_secret` and the client and server random numbers, but for TLS, the actual algorithm is different.

### Padding

In SSL, the padding added prior to encryption of user data is the minimum amount required so that the total size of the data to be encrypted is a multiple of the cipher's block length. In TLS, the padding can be any amount that results in a total that is a multiple of the cipher's block length, up to a maximum of 255 bytes. For example, if the plaintext (or compressed text if compression is used) plus MAC plus padding.length byte is 79 bytes long, then the padding length (in bytes) can be 1, 9, 17, and so on, up to 249. A variable padding length may be used to frustrate attacks based on an analysis of the lengths of exchanged messages.

## 5.4 HTTPS

HTTPS (HTTP over SSL) refers to the combination of HTTP and SSL to implement secure communication between a Web browser and a Web server. The HTTPS capability is built into all modern Web browsers. Its use depends on the Web server supporting HTTPS communication. For example, search engines do not support HTTPS.

The principal difference seen by a user of a Web browser is that URL (uniform resource locator) addresses begin with `https://` rather than `http://`. A normal HTTP connection uses port 80. If HTTPS is specified, port 443 is used, which invokes SSL.



When HTTPS is used, the following elements of the communication are encrypted:

- URL of the requested document
- Contents of the document
- Contents of browser forms (filled in by browser user)
- Cookies sent from browser to server and from server to browser
- Contents of HTTP header

HTTPS is documented in RFC 2818, *HTTP Over TLS*. There is no fundamental change in using HTTP over either SSL or TLS, and both implementations are referred to as HTTPS.

### Connection Initiation

For HTTPS, the agent acting as the HTTP client also acts as the TLS client. The client initiates a connection to the server on the appropriate port and then sends the TLS ClientHello to begin the TLS handshake. When the TLS handshake has finished, the client may then initiate the first HTTP request. All HTTP data is to be sent as TLS application data. Normal HTTP behavior, including retained connections, should be followed.

We need to be clear that there are three levels of awareness of a connection in HTTPS. At the HTTP level, an HTTP client requests a connection to an HTTP server by sending a connection request to the next lowest layer. Typically, the next lowest layer is TCP, but it also may be TLS/SSL. At the level of TLS, a session is established between a TLS client and a TLS server. This session can support one or more connections at any time. As we have seen, a TLS request to establish a connection begins with the establishment of a TCP connection between the TCP entity on the client side and the TCP entity on the server side.

### Connection Closure

An HTTP client or server can indicate the closing of a connection by including the following line in an HTTP record: `Connection: close`. This indicates that the connection will be closed after this record is delivered.

The closure of an HTTPS connection requires that TLS close the connection with the peer TLS entity on the remote side, which will involve closing the underlying TCP connection. At the TLS level, the proper way to close a connection is for each side to use the TLS alert protocol to send a `close_notify` alert. TLS implementations must initiate an exchange of closure alerts before closing a connection. A TLS implementation may, after sending a closure alert, close the connection without waiting for the peer to send its closure alert, generating an “incomplete close”. Note that an implementation that does this may choose to reuse the session. This should only be done when the application knows (typically through detecting HTTP message boundaries) that it has received all the message data that it cares about.

HTTP clients also must be able to cope with a situation in which the underlying TCP connection is terminated without a prior `close_notify` alert and without a `Connection: close` indicator. Such a situation could be due to a programming

error on the server or a communication error that causes the TCP connection to drop. However, the unannounced TCP closure could be evidence of some sort of attack. So the HTTPS client should issue some sort of security warning when this occurs.

## 5.5 SECURE SHELL (SSH)

Secure Shell (SSH) is a protocol for secure network communications designed to be relatively simple and inexpensive to implement. The initial version, SSH1 was focused on providing a secure remote logon facility to replace TELNET and other remote logon schemes that provided no security. SSH also provides a more general client/server capability and can be used for such network functions as file transfer and e-mail. A new version, SSH2, fixes a number of security flaws in the original scheme. SSH2 is documented as a proposed standard in IETF RFCs 4250 through 4256.

SSH client and server applications are widely available for most operating systems. It has become the method of choice for remote login and X tunneling and is rapidly becoming one of the most pervasive applications for encryption technology outside of embedded systems.

SSH is organized as three protocols that typically run on top of TCP (Figure 5.8):

- **Transport Layer Protocol:** Provides server authentication, data confidentiality, and data integrity with forward secrecy (i.e., if a key is compromised during one session, the knowledge does not affect the security of earlier sessions). The transport layer may optionally provide compression.

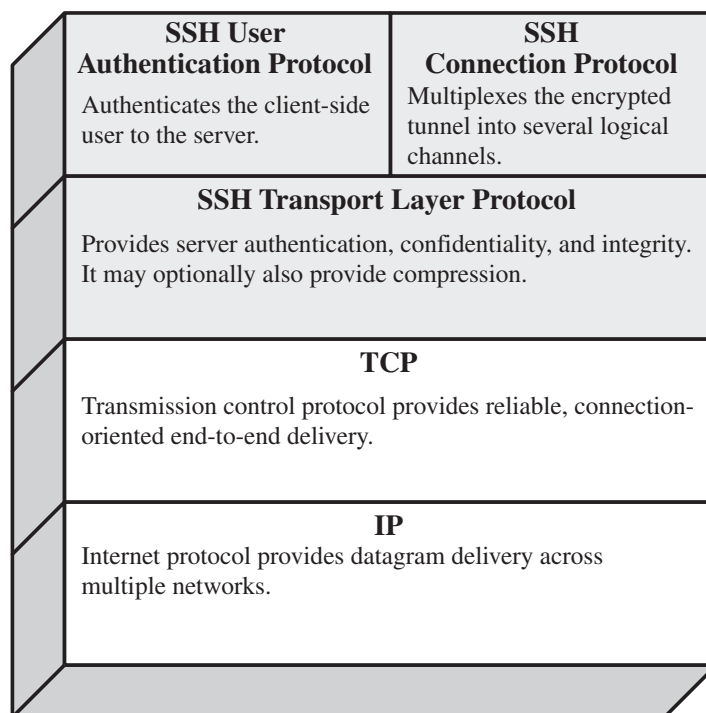


Figure 5.8 SSH Protocol Stack

- **User Authentication Protocol:** Authenticates the user to the server.
- **Connection Protocol:** Multiplexes multiple logical communications channels over a single, underlying SSH connection.

### Transport Layer Protocol

**HOST KEYS** Server authentication occurs at the transport layer, based on the server possessing a public/private key pair. A server may have multiple host keys using multiple different asymmetric encryption algorithms. Multiple hosts may share the same host key. In any case, the server host key is used during key exchange to authenticate the identity of the host. For this to be possible, the client must have a priori knowledge of the server's public host key. RFC 4251 dictates two alternative trust models that can be used:

1. The client has a local database that associates each host name (as typed by the user) with the corresponding public host key. This method requires no centrally administered infrastructure and no third-party coordination. The downside is that the database of name-to-key associations may become burdensome to maintain.
2. The host name-to-key association is certified by a trusted certification authority (CA). The client only knows the CA root key and can verify the validity of all host keys certified by accepted CAs. This alternative eases the maintenance problem, since ideally, only a single CA key needs to be securely stored on the client. On the other hand, each host key must be appropriately certified by a central authority before authorization is possible.

**PACKET EXCHANGE** Figure 5.9 illustrates the sequence of events in the SSH Transport Layer Protocol. First, the client establishes a TCP connection to the server. This is done via the TCP protocol and is not part of the Transport Layer Protocol. Once the connection is established, the client and server exchange data, referred to as packets, in the data field of a TCP segment. Each packet is in the following format (Figure 5.10).

- **Packet length:** Length of the packet in bytes, not including the packet length and MAC fields.
- **Padding length:** Length of the random padding field.
- **Payload:** Useful contents of the packet. Prior to algorithm negotiation, this field is uncompressed. If compression is negotiated, then in subsequent packets, this field is compressed.
- **Random padding:** Once an encryption algorithm has been negotiated, this field is added. It contains random bytes of padding so that total length of the packet (excluding the MAC field) is a multiple of the cipher block size, or 8 bytes for a stream cipher.
- **Message authentication code (MAC):** If message authentication has been negotiated, this field contains the MAC value. The MAC value is computed over the entire packet plus a sequence number, excluding the MAC field. The sequence number is an implicit 32-bit packet sequence that is initialized to

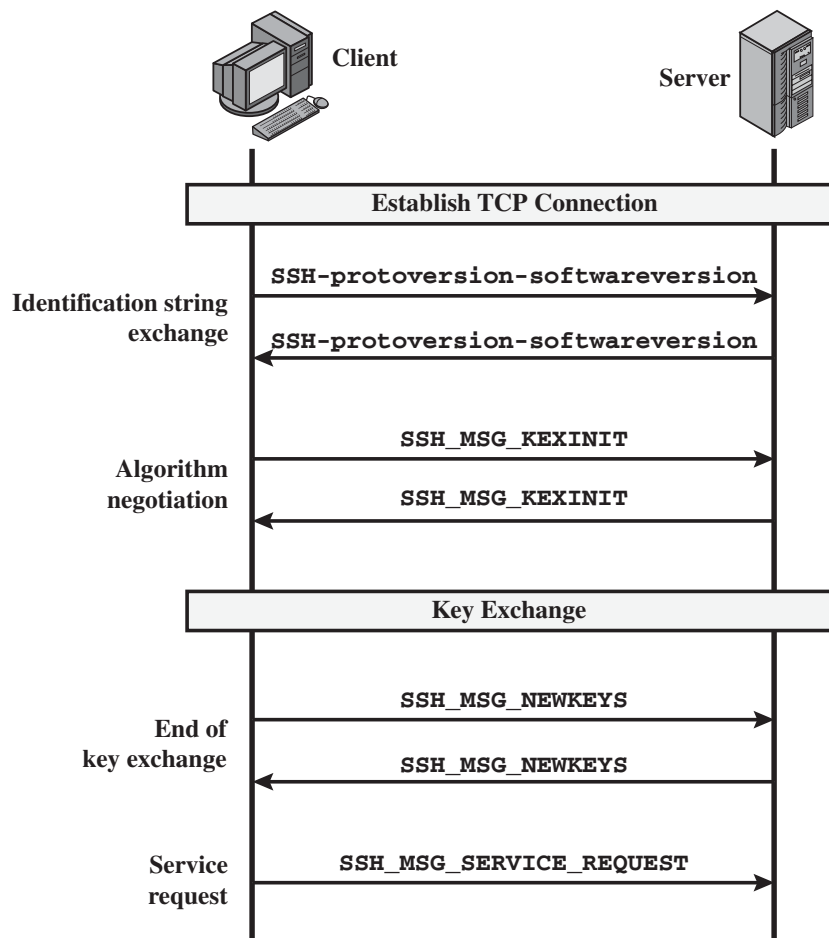


Figure 5.9 SSH Transport Layer Protocol Packet Exchanges

zero for the first packet and incremented for every packet. The sequence number is not included in the packet sent over the TCP connection.

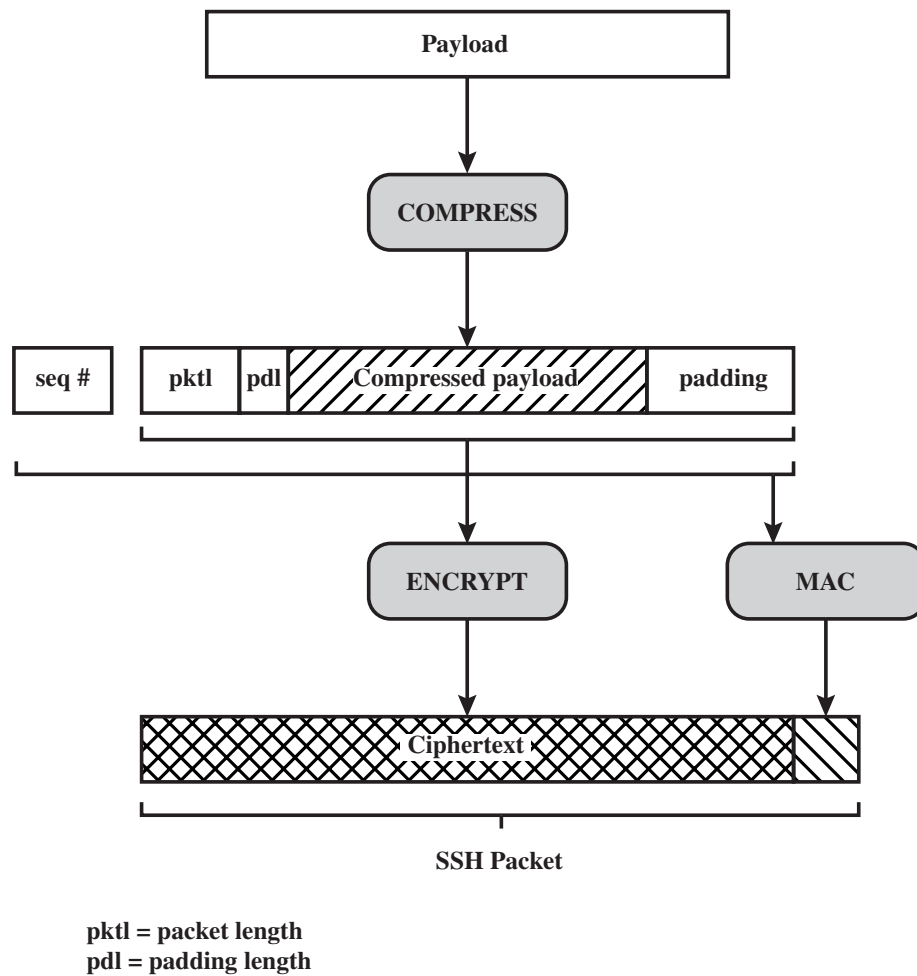
Once an encryption algorithm has been negotiated, the entire packet (excluding the MAC field) is encrypted after the MAC value is calculated.

The SSH Transport Layer packet exchange consists of a sequence of steps (Figure 5.9). The first step, the **identification string exchange**, begins with the client sending a packet with an identification string of the form:

```
SSH-protoversion-softwareversion SP comments CR LF
```

where SP, CR, and LF are space character, carriage return, and line feed, respectively. An example of a valid string is `SSH-2.0-billsSSH_3.6.3q3`<CR><LF>. The server responds with its own identification string. These strings are used in the Diffie-Hellman key exchange.

Next comes **algorithm negotiation**. Each side sends an `SSH_MSG_KEXINIT` containing lists of supported algorithms in the order of preference to the sender. There is one list for each type of cryptographic algorithm. The algorithms include key exchange, encryption, MAC algorithm, and compression algorithm. Table 5.3 shows the allowable options for encryption, MAC, and compression. For each category, the algorithm chosen is the first algorithm on the client's list that is also supported by the server.



**Figure 5.10** SSH Transport Layer Protocol Packet Formation

The next step is **key exchange**. The specification allows for alternative methods of key exchange, but at present, only two versions of Diffie-Hellman key exchange are specified. Both versions are defined in RFC 2409 and require only one packet in each direction. The following steps are involved in the exchange. In this, C is the client; S is the server;  $p$  is a large safe prime;  $g$  is a generator for a subgroup of  $\text{GF}(p)$ ;  $q$  is the order of the subgroup;  $V_S$  is S's identification string;  $V_C$  is C's identification string;  $K_S$  is S's public host key;  $I_C$  is C's `SSH_MSG_KEXINIT` message and  $I_S$  is S's `SSH_MSG_KEXINIT` message that have been exchanged before this part begins. The values of  $p$ ,  $g$ , and  $q$  are known to both client and server as a result of the algorithm selection negotiation. The hash function `hash()` is also decided during algorithm negotiation.

1. C generates a random number  $x$  ( $1 < x < q$ ) and computes  $e = g^x \bmod p$ . C sends  $e$  to S.
2. S generates a random number  $y$  ( $0 < y < q$ ) and computes  $f = g^y \bmod p$ . S receives  $e$ . It computes  $K = e^y \bmod p$ ,  $H = \text{hash}(V_C \parallel V_S \parallel I_C \parallel I_S \parallel K_S \parallel e \parallel f \parallel K)$ , and signature  $s$  on  $H$  with its private host key. S sends  $(K_S \parallel f \parallel s)$  to C. The signing operation may involve a second hashing operation.

**Table 5.3** SSH Transport Layer Cryptographic Algorithms

Cipher		MAC algorithm	
<b>3des-cbc*</b>	Three-key 3DES in CBC mode	<b>hmac-sha1*</b>	HMAC-SHA1; digest length = key length = 20
<b>blowfish-cbc</b>	Blowfish in CBC mode	<b>hmac-sha1-96**</b>	First 96 bits of HMAC-SHA1; digest length = 12; key length = 20
<b>twofish256-cbc</b>	Twofish in CBC mode with a 256-bit key	<b>hmac-md5</b>	HMAC-SHA1; digest length = key length = 16
<b>twofish192-cbc</b>	Twofish with a 192-bit key	<b>hmac-md5-96</b>	First 96 bits of HMAC-SHA1; digest length = 12; key length = 16
<b>twofish128-cbc</b>	Twofish with a 128-bit key	<b>Compression algorithm</b>	
<b>aes256-cbc</b>	AES in CBC mode with a 256-bit key		
<b>aes192-cbc</b>	AES with a 192-bit key	<b>none*</b>	No compression
<b>aes128-cbc**</b>	AES with a 128-bit key	<b>zlib</b>	Defined in RFC 1950 and RFC 1951
<b>Serpent256-cbc</b>	Serpent in CBC mode with a 256-bit key		
<b>Serpent192-cbc</b>	Serpent with a 192-bit key		
<b>Serpent128-cbc</b>	Serpent with a 128-bit key		
<b>arcfour</b>	RC4 with a 128-bit key		
<b>cast128-cbc</b>	CAST-128 in CBC mode		

\* = Required

\*\* = Recommended

3. C verifies that  $K_S$  really is the host key for S (e.g., using certificates or a local database). C is also allowed to accept the key without verification; however, doing so will render the protocol insecure against active attacks (but may be desirable for practical reasons in the short term in many environments). C then computes  $K = f^x \bmod p$ ,  $H = \text{hash}(V_C \parallel V_S \parallel I_C \parallel I_S \parallel K_S \parallel e \parallel f \parallel K)$ , and verifies the signature  $s$  on  $H$ .

As a result of these steps, the two sides now share a master key  $K$ . In addition, the server has been authenticated to the client, because the server has used its private key to sign its half of the Diffie-Hellman exchange. Finally, the hash value  $H$  serves as a session identifier for this connection. Once computed, the session identifier is not changed, even if the key exchange is performed again for this connection to obtain fresh keys.

The **end of key exchange** is signaled by the exchange of `SSH_MSG_NEWKEYS` packets. At this point, both sides may start using the keys generated from  $K$ , as discussed subsequently.

The final step is **service request**. The client sends an `SSH_MSG_SERVICE_REQUEST` packet to request either the User Authentication or the Connection Protocol. Subsequent to this, all data is exchanged as the payload of an SSH Transport Layer packet, protected by encryption and MAC.

**KEY GENERATION** The keys used for encryption and MAC (and any needed IVs) are generated from the shared secret key  $K$ , the hash value from the key exchange  $H$ , and the session identifier, which is equal to  $H$  unless there has been a subsequent key exchange after the initial key exchange. The values are computed as follows.

- Initial IV client to server:  $\text{HASH}(K \parallel H \parallel \text{"A"} \parallel \text{session\_id})$
- Initial IV server to client:  $\text{HASH}(K \parallel H \parallel \text{"B"} \parallel \text{session\_id})$
- Encryption key client to server:  $\text{HASH}(K \parallel H \parallel \text{"C"} \parallel \text{session\_id})$
- Encryption key server to client:  $\text{HASH}(K \parallel H \parallel \text{"D"} \parallel \text{session\_id})$
- Integrity key client to server:  $\text{HASH}(K \parallel H \parallel \text{"E"} \parallel \text{session\_id})$
- Integrity key server to client:  $\text{HASH}(K \parallel H \parallel \text{"F"} \parallel \text{session\_id})$

where  $\text{HASH}()$  is the hash function determined during algorithm negotiation.

## User Authentication Protocol

The User Authentication Protocol provides the means by which the client is authenticated to the server.

**MESSAGE TYPES AND FORMATS** Three types of messages are always used in the User Authentication Protocol. Authentication requests from the client have the format:

byte	<code>SSH_MSG_USERAUTH_REQUEST</code> (50)
string	user name
string	service name
string	method name
...	method specific fields

where user name is the authorization identity the client is claiming, service name is the facility to which the client is requesting access (typically the SSH Connection Protocol), and method name is the authentication method being used in this request. The first byte has decimal value 50, which is interpreted as `SSH_MSG_USERAUTH_REQUEST`.

If the server either (1) rejects the authentication request or (2) accepts the request but requires one or more additional authentication methods, the server sends a message with the format:

byte	<code>SSH_MSG_USERAUTH_FAILURE</code> (51)
name-list	authentications that can continue
boolean	partial success

where the name-list is a list of methods that may productively continue the dialog. If the server accepts authentication, it sends a single byte message: `SSH_MSG_USERAUTH_SUCCESS` (52).



**MESSAGE EXCHANGE** The message exchange involves the following steps.

1. The client sends a `SSH_MSG_USERAUTH_REQUEST` with a requested method of none.
2. The server checks to determine if the user name is valid. If not, the server returns `SSH_MSG_USERAUTH_FAILURE` with the partial success value of false. If the user name is valid, the server proceeds to step 3.
3. The server returns `SSH_MSG_USERAUTH_FAILURE` with a list of one or more authentication methods to be used.
4. The client selects one of the acceptable authentication methods and sends a `SSH_MSG_USERAUTH_REQUEST` with that method name and the required method-specific fields. At this point, there may be a sequence of exchanges to perform the method.
5. If the authentication succeeds and more authentication methods are required, the server proceeds to step 3, using a partial success value of true. If the authentication fails, the server proceeds to step 3, using a partial success value of false.
6. When all required authentication methods succeed, the server sends a `SSH_MSG_USERAUTH_SUCCESS` message, and the Authentication Protocol is over.

**AUTHENTICATION METHODS** The server may require one or more of the following authentication methods.

- **publickey:** The details of this method depend on the public-key algorithm chosen. In essence, the client sends a message to the server that contains the client's public key, with the message signed by the client's private key. When the server receives this message, it checks whether the supplied key is acceptable for authentication and, if so, it checks whether the signature is correct.
- **password:** The client sends a message containing a plaintext password, which is protected by encryption by the Transport Layer Protocol.
- **hostbased:** Authentication is performed on the client's host rather than the client itself. Thus, a host that supports multiple clients would provide authentication for all its clients. This method works by having the client send a signature created with the private key of the client host. Thus, rather than directly verifying the user's identity, the SSH server verifies the identity of the client host—and then believes the host when it says the user has already authenticated on the client side.

## Connection Protocol

The SSH Connection Protocol runs on top of the SSH Transport Layer Protocol and assumes that a secure authentication connection is in use.<sup>2</sup> That secure authentication

---

<sup>2</sup>RFC 4254, *The Secure Shell (SSH) Connection Protocol*, states that the Connection Protocol runs on top of the Transport Layer Protocol and the User Authentication Protocol. RFC 4251, *SSH Protocol Architecture*, states that the Connection Protocol runs over the User Authentication Protocol. In fact, the Connection Protocol runs over the Transport Layer Protocol, but assumes that the User Authentication Protocol has been previously invoked.

connection, referred to as a **tunnel**, is used by the Connection Protocol to multiplex a number of logical channels.

*CHANNEL MECHANISM* All types of communication using SSH, such as a terminal session, are supported using separate channels. Either side may open a channel. For each channel, each side associates a unique channel number, which need not be the same on both ends. Channels are flow controlled using a window mechanism. No data may be sent to a channel until a message is received to indicate that window space is available.

The life of a channel progresses through three stages: opening a channel, data transfer, and closing a channel.

When either side wishes to **open a new channel**, it allocates a local number for the channel and then sends a message of the form:

byte	SSH_MSG_CHANNEL_OPEN
string	channel type
uint32	sender channel
uint32	initial window size
uint32	maximum packet size
....	channel type specific data follows

where uint32 means unsigned 32-bit integer. The channel type identifies the application for this channel, as described subsequently. The sender channel is the local channel number. The initial window size specifies how many bytes of channel data can be sent to the sender of this message without adjusting the window. The maximum packet size specifies the maximum size of an individual data packet that can be sent to the sender. For example, one might want to use smaller packets for interactive connections to get better interactive response on slow links.

If the remote side is able to open the channel, it returns a SSH\_MSG\_CHANNEL\_OPEN\_CONFIRMATION message, which includes the sender channel number, the recipient channel number, and window and packet size values for incoming traffic. Otherwise, the remote side returns a SSH\_MSG\_CHANNEL\_OPEN\_FAILURE message with a reason code indicating the reason for failure.

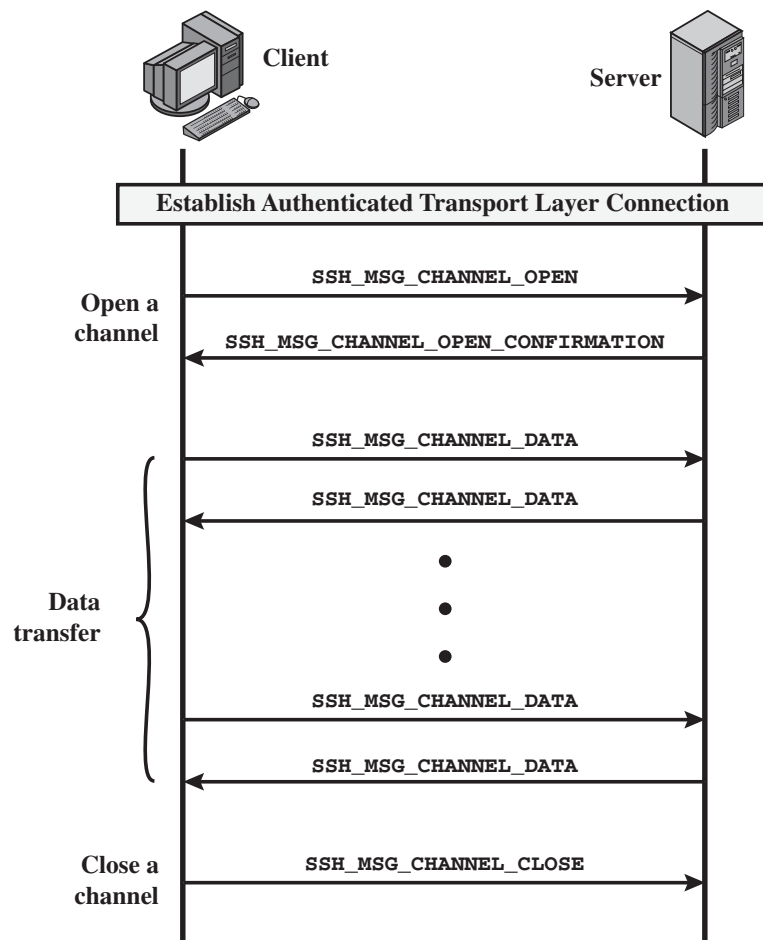
Once a channel is open, **data transfer** is performed using a SSH\_MSG\_CHANNEL\_DATA message, which includes the recipient channel number and a block of data. These messages, in both directions, may continue as long as the channel is open.

When either side wishes to **close a channel**, it sends a SSH\_MSG\_CHANNEL\_CLOSE message, which includes the recipient channel number.

Figure 5.11 provides an example of Connection Protocol Message Exchange.

*CHANNEL TYPES* Four channel types are recognized in the SSH Connection Protocol specification.

- **session:** The remote execution of a program. The program may be a shell, an application such as file transfer or e-mail, a system command, or some built-in subsystem. Once a session channel is opened, subsequent requests are used to start the remote program.



**Figure 5.11** Example SSH Connection Protocol Message Exchange

- **x11:** This refers to the X Window System, a computer software system and network protocol that provides a graphical user interface (GUI) for networked computers. X allows applications to run on a network server but to be displayed on a desktop machine.
- **forwarded-tcpip:** This is remote port forwarding, as explained in the next subsection.
- **direct-tcpip:** This is local port forwarding, as explained in the next subsection.

**PORT FORWARDING** One of the most useful features of SSH is port forwarding. In essence, port forwarding provides the ability to convert any insecure TCP connection into a secure SSH connection. This is also referred to as SSH tunneling. We need to know what a port is in this context. A **port** is an identifier of a user of TCP. So, any application that runs on top of TCP has a port number. Incoming TCP traffic is delivered to the appropriate application on the basis of the port number. An application may employ multiple port numbers. For example, for the Simple Mail Transfer Protocol (SMTP), the server side generally listens on port 25, so an incoming SMTP request uses TCP and addresses the data to destination port 25. TCP recognizes that this is the SMTP server address and routes the data to the SMTP server application.

Figure 5.12 illustrates the basic concept behind port forwarding. We have a client application that is identified by port number  $x$  and a server application identified by port number  $y$ . At some point, the client application invokes the local TCP entity and requests a connection to the remote server on port  $y$ . The local TCP entity negotiates a TCP connection with the remote TCP entity, such that the connection links local port  $x$  to remote port  $y$ .

To secure this connection, SSH is configured so that the SSH Transport Layer Protocol establishes a TCP connection between the SSH client and server entities with TCP port numbers  $a$  and  $b$ , respectively. A secure SSH tunnel is established over this TCP connection. Traffic from the client at port  $x$  is redirected to the local

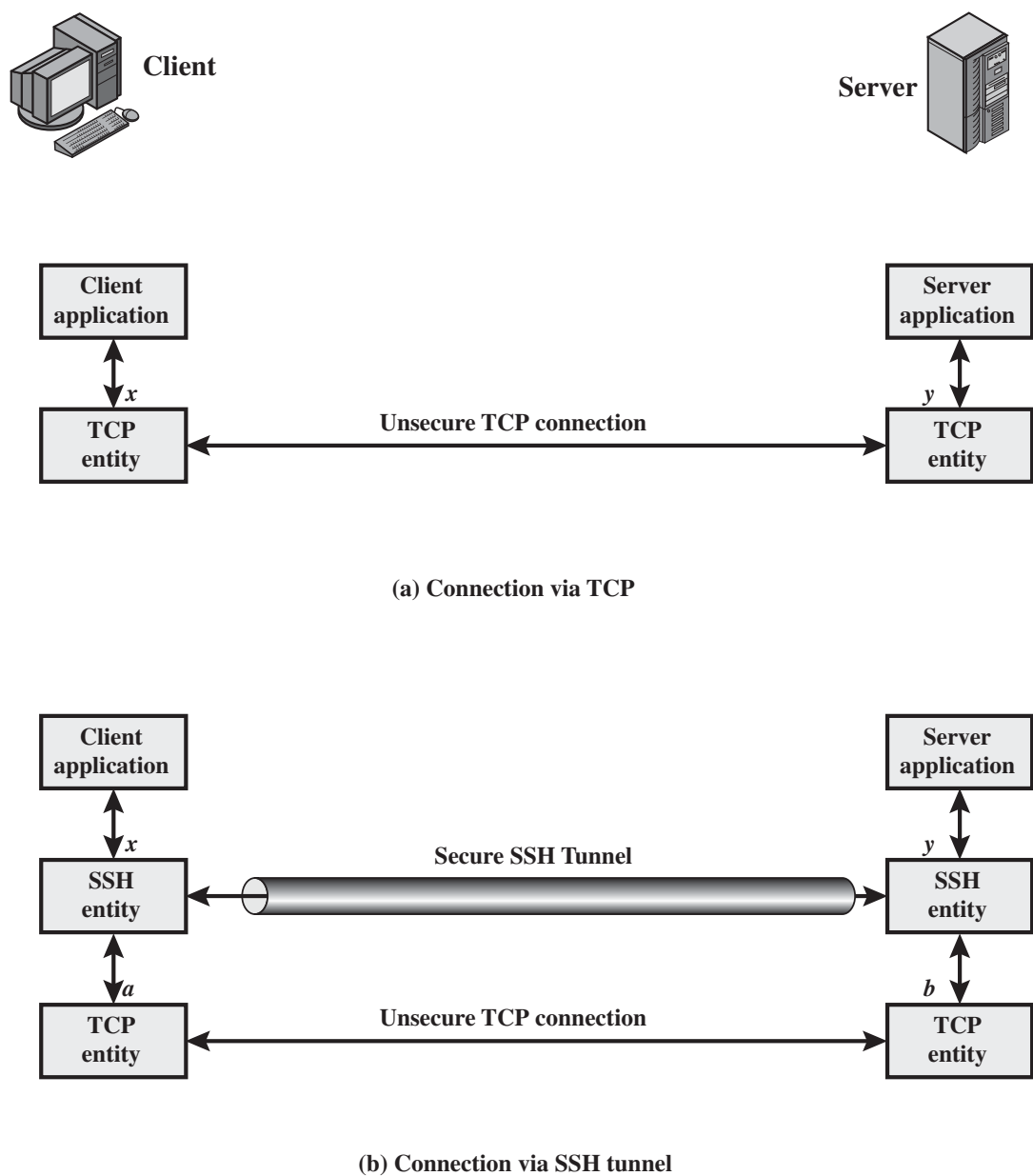


Figure 5.12 SSH Transport Layer Packet Exchanges

SSH entity and travels through the tunnel where the remote SSH entity delivers the data to the server application on port *y*. Traffic in the other direction is similarly redirected.

SSH supports two types of port forwarding: local forwarding and remote forwarding. **Local forwarding** allows the client to set up a “hijacker” process. This will intercept selected application-level traffic and redirect it from an unsecured TCP connection to a secure SSH tunnel. SSH is configured to listen on selected ports. SSH grabs all traffic using a selected port and sends it through an SSH tunnel. On the other end, the SSH server sends the incoming traffic to the destination port dictated by the client application.

The following example should help clarify local forwarding. Suppose you have an e-mail client on your desktop and use it to get e-mail from your mail server via the Post Office Protocol (POP). The assigned port number for POP3 is port 110. We can secure this traffic in the following way:

1. The SSH client sets up a connection to the remote server.
2. Select an unused local port number, say 9999, and configure SSH to accept traffic from this port destined for port 110 on the server.
3. The SSH client informs the SSH server to create a connection to the destination, in this case mailserver port 110.
4. The client takes any bits sent to local port 9999 and sends them to the server inside the encrypted SSH session. The SSH server decrypts the incoming bits and sends the plaintext to port 110.
5. In the other direction, the SSH server takes any bits received on port 110 and sends them inside the SSH session back to the client, who decrypts and sends them to the process connected to port 9999.

With **remote forwarding**, the user’s SSH client acts on the server’s behalf. The client receives traffic with a given destination port number, places the traffic on the correct port and sends it to the destination the user chooses. A typical example of remote forwarding is the following. You wish to access a server at work from your home computer. Because the work server is behind a firewall, it will not accept an SSH request from your home computer. However, from work you can set up an SSH tunnel using remote forwarding. This involves the following steps.

1. From the work computer, set up an SSH connection to your home computer. The firewall will allow this, because it is a protected outgoing connection.
2. Configure the SSH server to listen on a local port, say 22, and to deliver data across the SSH connection addressed to remote port, say 2222.
3. You can now go to your home computer, and configure SSH to accept traffic on port 2222.
4. You now have an SSH tunnel that can be used for remote login to the work server.

## 5.6 RECOMMENDED READING AND WEB SITES

[RESC01] is a good detailed treatment of SSL and TLS. [BARR05] provides a thorough treatment of SSH. The original version (SSH-1) of SSH was introduced in [YLON96].

**BARR05** Barrett, D.; Silverman, R.; and Byrnes, R. *SSH The Secure Shell: The Definitive Guide*. Sebastopol, CA: O'Reilly, 2005.

**RESC01** Rescorla, E. *SSL and TLS: Designing and Building Secure Systems*. Reading, MA: Addison-Wesley, 2001.

**YLON96** Ylonen, T. "SSH - Secure Login Connections over the Internet." *Proceedings, Sixth USENIX Security Symposium*, July 1996.



### Recommended Web Sites:

- **Transport Layer Security Charter:** Latest RFCs and Internet drafts for TLS.
- **OpenSSL Project:** Project to develop open-source SSL and TLS software. Site includes documents and links.

## 5.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

### Key Terms

Alert protocol Change Cipher Spec protocol Handshake protocol	HTTPS (HTTP over SSL) Master Secret Secure Shell (SSH)	Secure Socket Layer (SSL) Transport Layer Security (TLS)
---	--	---

### Review Questions

- 5.1 What are the advantages of each of the three approaches shown in Figure 5.1?
- 5.2 What protocols comprise SSL?
- 5.3 What is the difference between an SSL connection and an SSL session?
- 5.4 List and briefly define the parameters that define an SSL session state.
- 5.5 List and briefly define the parameters that define an SSL session connection.
- 5.6 What services are provided by the SSL Record Protocol?
- 5.7 What steps are involved in the SSL Record Protocol transmission?
- 5.8 What is the purpose of HTTPS?
- 5.9 For what applications is SSH useful?
- 5.10 List and briefly define the SSH protocols.

## Problems

- 5.1 In SSL and TLS, why is there a separate Change Cipher Spec Protocol rather than including a `change_cipher_spec` message in the Handshake Protocol?
- 5.2 What purpose does the MAC serve during the change cipher spec SSL exchange?
- 5.3 Consider the following threats to Web security and describe how each is countered by a particular feature of SSL.
  - a. Brute-Force Cryptanalytic Attack: An exhaustive search of the key space for a conventional encryption algorithm.
  - b. Known Plaintext Dictionary Attack: Many messages will contain predictable plaintext, such as the HTTP GET command. An attacker constructs a dictionary containing every possible encryption of the known-plaintext message. When an encrypted message is intercepted, the attacker takes the portion containing the encrypted known plaintext and looks up the ciphertext in the dictionary. The ciphertext should match against an entry that was encrypted with the same secret key. If there are several matches, each of these can be tried against the full ciphertext to determine the right one. This attack is especially effective against small key sizes (e.g., 40-bit keys).
  - c. Replay Attack: Earlier SSL handshake messages are replayed.
  - d. Man-in-the-Middle Attack: An attacker interposes during key exchange, acting as the client to the server and as the server to the client.
  - e. Password Sniffing: Passwords in HTTP or other application traffic are eavesdropped.
  - f. IP Spoofing: Uses forged IP addresses to fool a host into accepting bogus data.
  - g. IP Hijacking: An active, authenticated connection between two hosts is disrupted and the attacker takes the place of one of the hosts.
  - h. SYN Flooding: An attacker sends TCP SYN messages to request a connection but does not respond to the final message to establish the connection fully. The attacked TCP module typically leaves the “half-open connection” around for a few minutes. Repeated SYN messages can clog the TCP module.
- 5.4 Based on what you have learned in this chapter, is it possible in SSL for the receiver to reorder SSL record blocks that arrive out of order? If so, explain how it can be done. If not, why not?
- 5.5 For SSH packets, what is the advantage, if any, of not including the MAC in the scope of the packet encryption?