Module 2

Transport Level Security
Web security considerations, Secure Sockets Layer, Transport Layer Security, HTTPS, Secure Shell (SSH)

2.1 WEB SECURITY CONSIDERATIONS

The World Wide Web is a client/server application running over the Internet and TCP/IP intranets.

- 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 easy to develop, the underlying software is extraordinarily complex. This complex software may hide many potential security flaws.
- 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

- 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. The various types of security threats faced when using the Web are listed below.

Table 1 : Web Security threats

	Threats	Consequences	Countermeasures
Integrity	• Modification of user data • Trojan horse browser • Modification of memory • Modification of message traffic in transit	• Loss of information • Compromise of machine • Vulnerability to all other threats	Cryptographic checksums
Confidentiality	• Eavesdropping on the net • Theft of info from server • Theft of data from client • Info about network configuration • Info about which client talks to server	• Loss of information • Loss of privacy	Encryption, Web proxies
Denial of Service	• Killing of user threads • Flooding machine with bogus requests • Filling up disk or memory • Isolating machine by DNS attacks	• Disruptive • Annoying • Prevent user from getting work done	Difficult to prevent
Authentication	• Impersonation of legitimate users • Data forgery	• Misrepresentation of user • Belief that false information is valid	Cryptographic techniques

WEB TRAFFIC SECURITY APPROACHES

- 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 2.1 illustrates this difference.

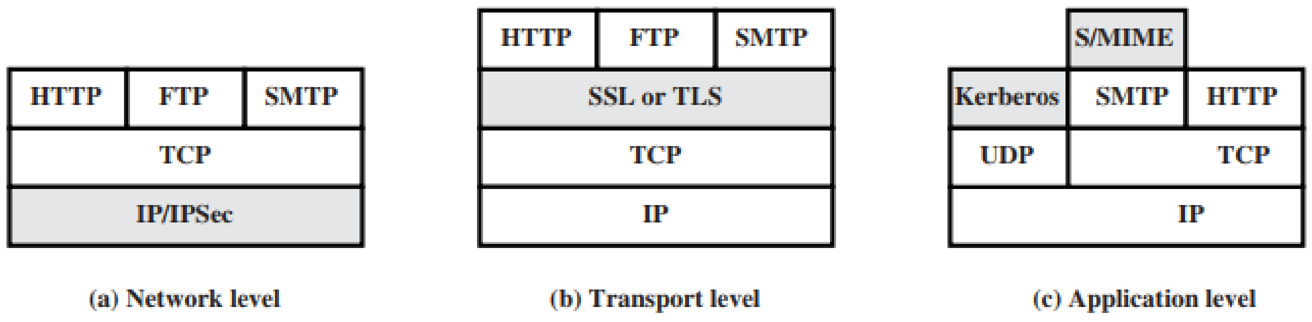


Figure 2.1 Relative Location of Security Facilities in the TCP/IP Protocol Stack

- One way to provide Web security is to use IP security (IPsec) (Figure 2.1a). The advantage of using IPsec is that it is transparent to end users and applications and provides a general-purpose solution. IPsec includes a filtering capability so that only selected traffic need to incur the overhead of IPsec processing.

- Another general-purpose solution is to implement security just above TCP (Figure 2.1b). The example of this approach is the Secure 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.

- Application-specific security services are embedded within the particular application. Figure 2.1c shows examples of this architecture. The advantage of this approach is that the service can be provided to the specific needs of a given application.

2.2 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.
- This first published version of TLS can be viewed as essentially an SSLv3.1 and is very close to and backward compatible with SSLv3.
- **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 2.2

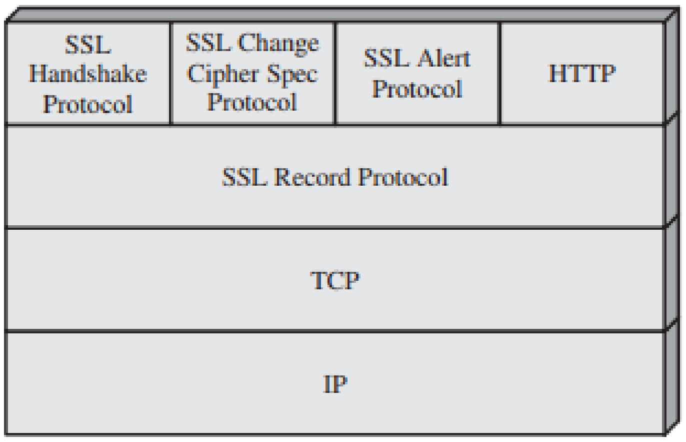


Figure 2.2 SSL Protocol Stack

- 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.
- Two important SSL concepts are the SSL session and the SSL connection, which are defined in the specification as follows.
  - ✓ **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 security parameters which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.
  - ✓ **Connection**: A connection is a transport 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.

A **session state** is defined by the following parameters.

- **Session identifier**: An arbitrary byte (8 bits) sequence chosen by the server to identify an active or resumable session state.
- **Peer certificate**: An X509.v3 certificate of the peer. This parameter may be empty(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.
- **Master secret**: 48-byte secret shared between the client and server.
- **Is resumable**: A Yes- No flag indicating whether the session can be used to initiate new connections in an old session.

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.
- **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 2.3 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.

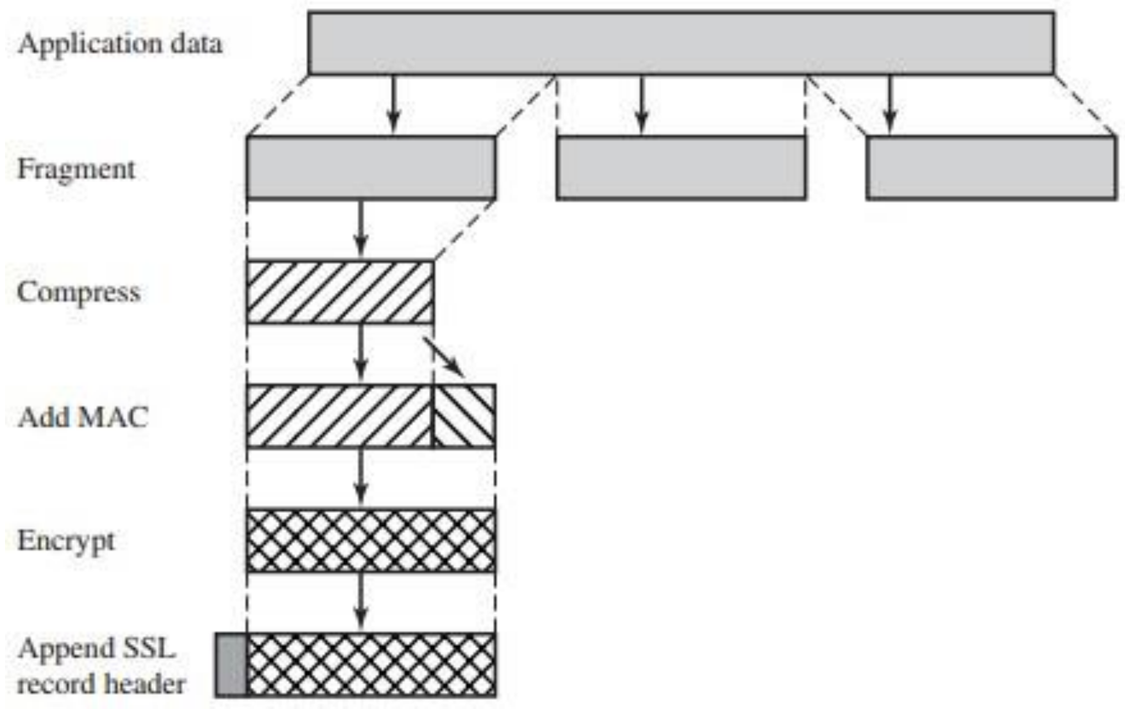


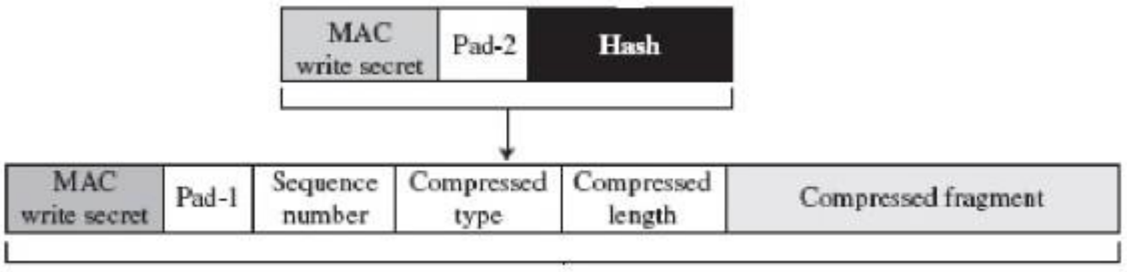
Figure 2.3 SSL Record Protocol operation

- 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. 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 (MAC)** over the compressed data. For this purpose, a shared secret key is used. The calculation is defined as

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

`||`
`MAC_write_secret`
`hash`
	`=concatenation`
	`=shared secret key`
	`=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)`



- 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 IDEA RC2-40 DES-40 DES 3DES Fortezza	128, 256 128 40 40 56 168 80	RC4-40 RC4-128	40 128

- For stream encryption, the compressed message plus the MAC are encrypted.
- For block encryption, padding may be added after the MAC prior to encryption.

- The final step of SSL Record Protocol processing is to prepare a header consisting of the following fields as shown in figure 2.4

✓ **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.

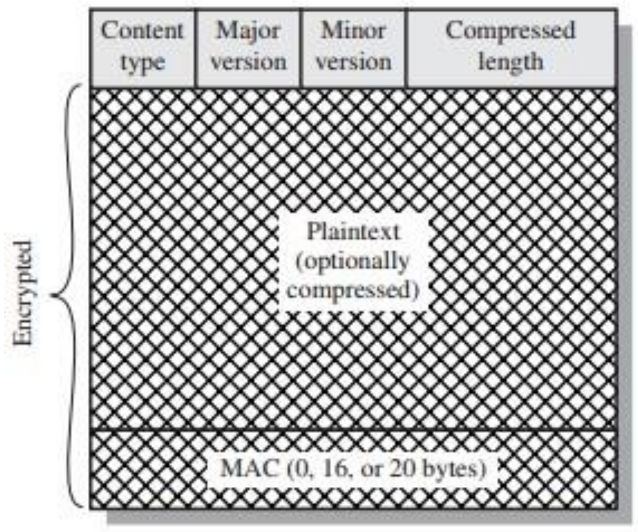


Figure 2.4 SSL record format.

- **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 2.5a), 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.

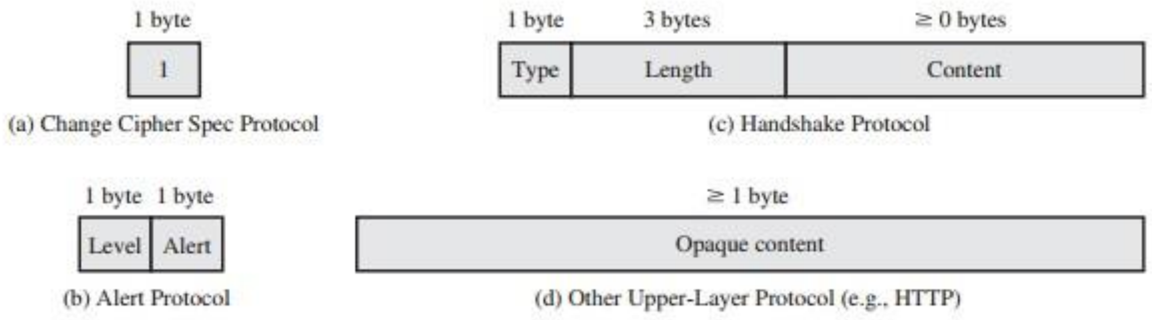
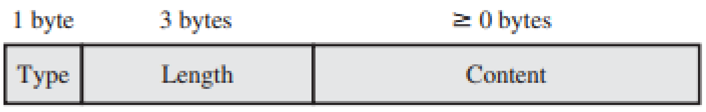


Figure 2.5 SSL record Protocol Payload.

- **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 2.5b). 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. The list of 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 Handshake 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 2.5c. Each message has three fields:



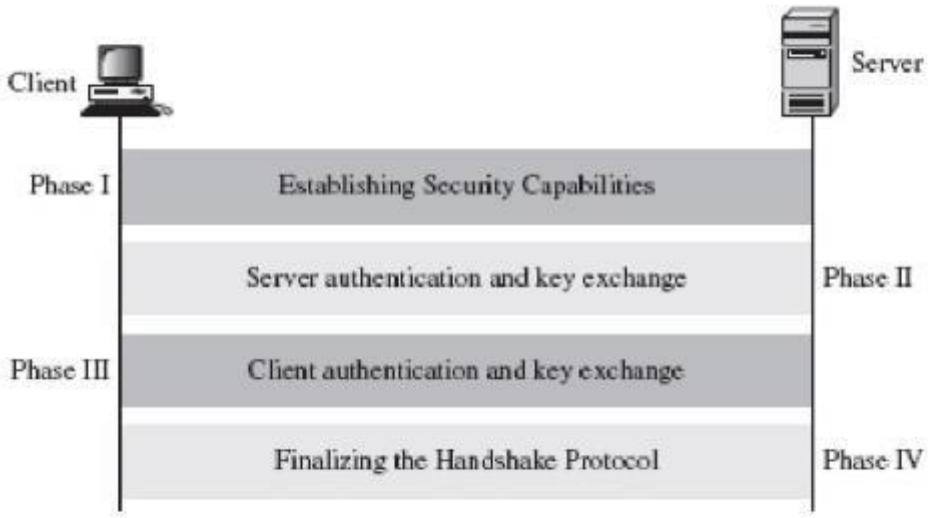
(c) Handshake Protocol

- ✓ **Type** (1 byte): Indicates one of 10 messages. Table 2.2 lists the defined message types.
- ✓ **Length** (3 bytes): The length of the message in bytes.
- ✓ **Content** (bytes): The parameters associated with this message; these are listed in Table 2.2.

Table 2.2 SSL Handshake Protocol Message Types

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

- Figure 2.6 shows the initial exchange needed to establish a logical connection between client and server. The exchange can be viewed as having four phases as shown in figure
  - ✓ Phase 1. Establish Security Capabilities
  - ✓ Phase 2. Server Authentication and Key Exchange
  - ✓ Phase 3. Client Authentication and Key Exchange
  - ✓ Phase 4. Finish 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

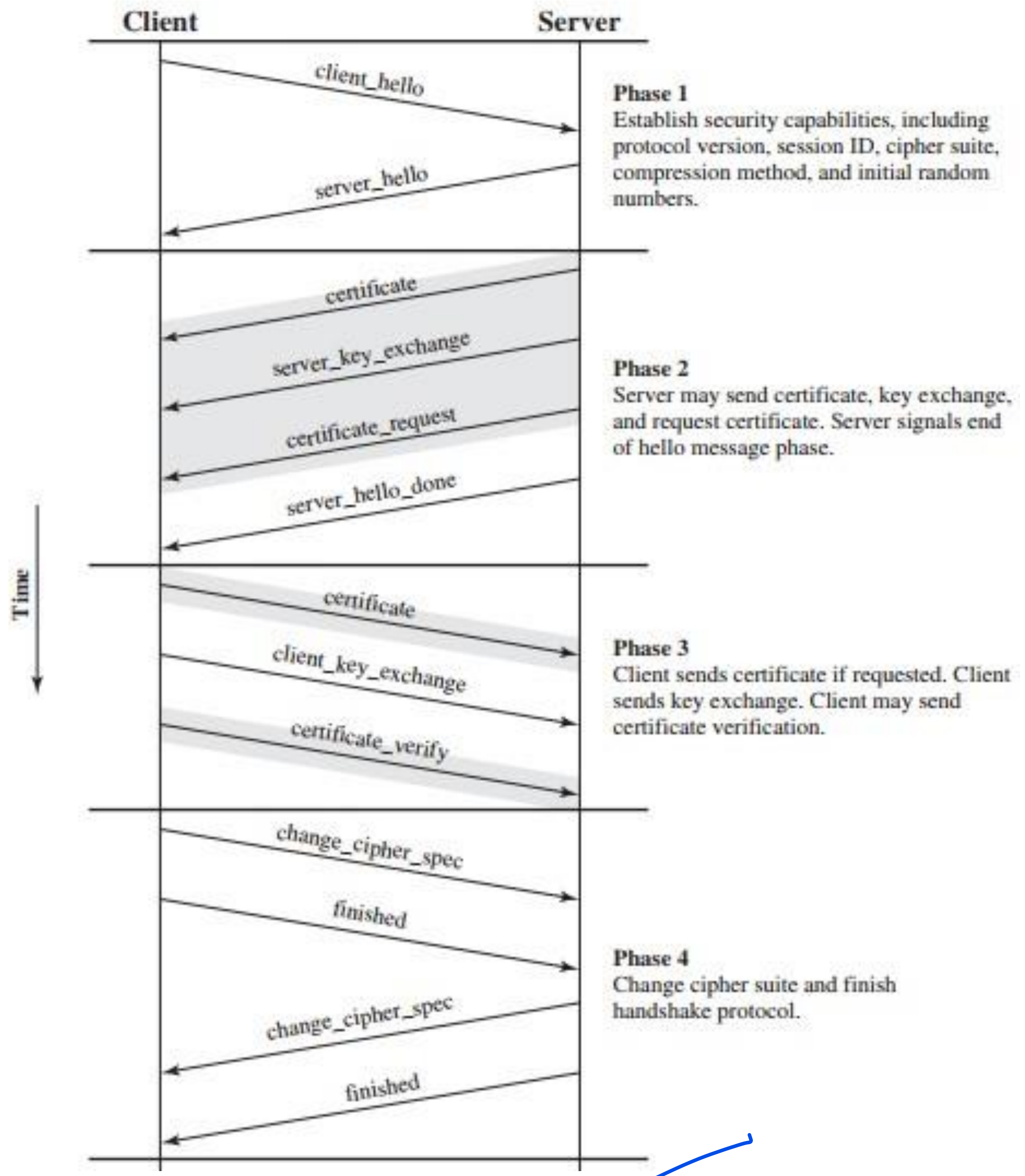

Client
Server
client_hello
server_hello
Phase 1
Establish security capabilities, including protocol version, session ID, cipher suite, compression method, and initial random numbers.
certificate
server_key_exchange
certificate_request
server_hello_done
Phase 2
Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase.
Time
certificate
client_key_exchange
certificate_verify
Phase 3
Client sends certificate if requested. Client sends key exchange. Client may send certificate verification.
change_cipher_spec
finished
change_cipher_spec
finished
Phase 4
Change cipher suite and finish handshake protocol.

Figure 2.6 Handshake protocol action.

- **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 the client can support.
  - ✓ **Random**: A client-generated random 32-byte random number that will be used for master secret generation. These values are used during key exchange to prevent replay attacks.
  - ✓ **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 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;
  - ✓ **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.
  - ✓ **Version:** The Version field contains the lower of the two versions ; the highest supported by the client and the highest supported by the server.
  - ✓ **Random**: A 32-byte random number that will be used for master secret generation. The Random field is generated by the server and is independent of the client's Random field.
  - ✓ **Session:** 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.
  - ✓ **CipherSuite**: The CipherSuite field contains the single cipher suite selected by the server from those proposed by the client.
  - ✓ **Compression Method** : The Compression field contains the compression method selected by the server from those proposed by the client.

    NOTE:

    - The first element of the CipherSuite parameter is the key exchange method (i.e., the means by which the cryptographic keys for 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). 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. 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 (security protocols defined for defence department).

- 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 key

✓ **IV Size**: The size of the Initialization Value for Cipher Block Chaining (CBC) encryption

- **PHASE 2. SERVER AUTHENTICATION AND KEY EXCHANGE**
  - In phase II, the server authenticates itself if needed. The server may send its certificate, its public key and may also request certificates from the client. At the end the server announces that the server_hello process is done.
  - **Certificate**: 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 is not needed if the key-exchange algorithm is anonymous Diffie-Hellman.

- **Server Key Exchange**: Next, a server_key_exchange message that includes its contribution to the pre-master secret may be sent if it is required. It is not required in two instances if the key exchange method is RSA or fixed Diffie Hellman.
- The server_key_exchange message is needed for the following:
  - ✓ <u>Anonymous Diffie-Hellman:</u> In this method, there is no certificate message. An anonymous entity does not have a certificate. In the Server key exchange, the server sends the Diffie Hellman parameters and its half-key.
  - ✓ <u>Ephemeral Diffie-Hellman</u>: 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.
  - ✓ <u>RSA</u> exchange (in which the server is using RSA but has a signature-only RSA key): 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.
- **Certificate Request:** Next, a non-anonymous server (server not using anonymous Diffie-Hellman) can request a certificate from the client to authenticate itself. The certificate_request message includes two parameters: certificate_type and certificate_authorities.
- The certificate type indicates the public-key algorithm and its use:
- The second parameter in the certificate_request message is a list of the distinguished names of acceptable certificate authorities.
- **Server Hello Done** The final message in phase 2, , 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**
  - Phase 3 is designed to authenticate the client.
  - 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.
  - **Certificate:** 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.
  - **Client_Key_Exchange**. The client sends a client key exchange message, which includes its contribution to the pre-master secret. The content of this message is based on the

type of key exchange, as follows.

- ✓ RSA: The client generates the entire (48-byte) pre-master secret and encrypts with the RSA public key of the server.
- ✓ 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.

- **Certificate verify**: 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

- **PHASE 4. FINISH & FINALIZE**
  - This phase completes the setting up of a secure connection.
  - **Change_Cipher_Spec** : The client sends a change_cipher_spec message and copies the pending CipherSpec into the current(active) CipherSpec.
  - **Finished**: 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.
  - **Change_Cipher_Spec**: 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.
  - **Finished:** Finally the server sends a finished message to show the handshaking is totally completed

2.3 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 the differences are listed below:

- **Version Number**

The TLS Record Format is the same as that of the SSL Record Format 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. HMAC is defined as

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

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)
ipad = 00110110 (36 in hexadecimal) repeated 64 times (512 bits)
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 2.7) given as

P_hash(secret, seed)= HMAC_hash(secret,A(1) || seed) ||
HMAC_hash(secret, A(2) || seed) ||
HMAC_hash(secret, A(3) || seed) || . . .

where A() is defined as

A(0) = seed

A(i) = HMAC_hash(secret, A(i − 1))

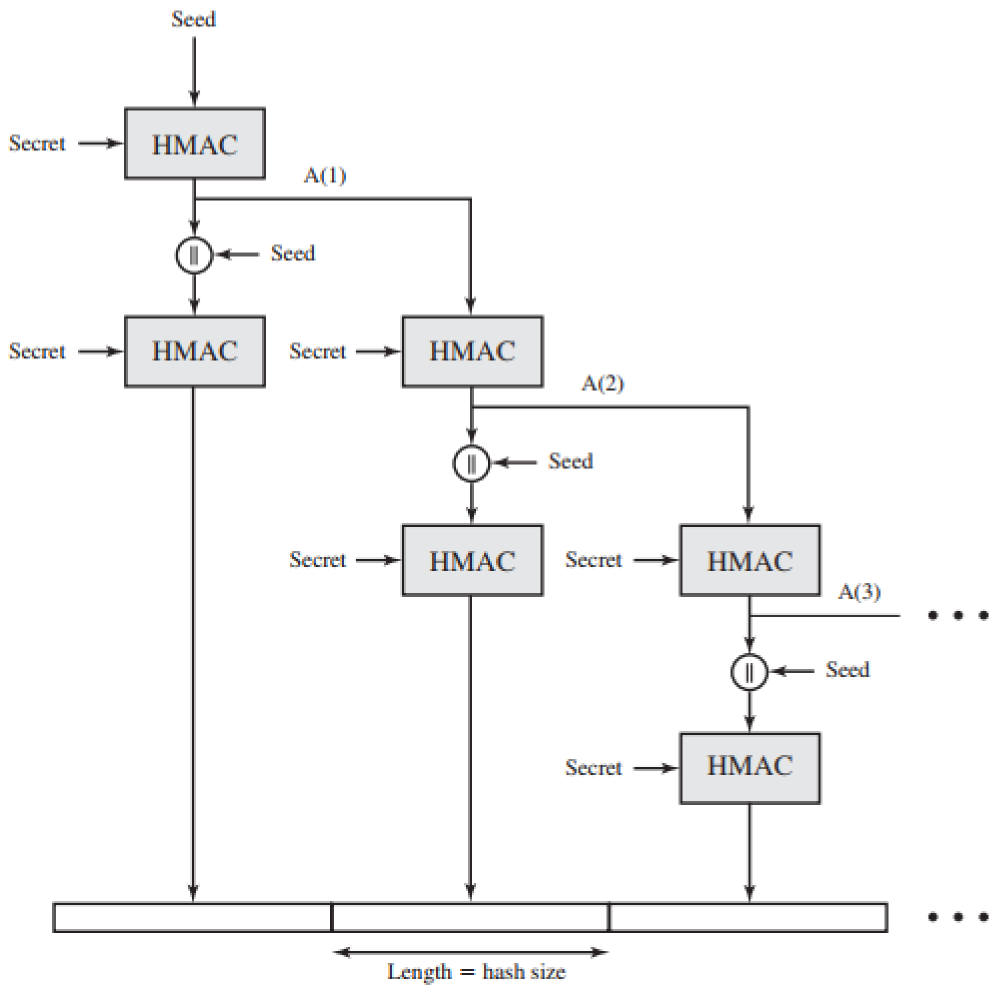


Figure 2.7 TLS Function P_hash (secret, seed)

- The data expansion function makes use of the HMAC algorithm with either MD5 or SHA-1 as the hash function.

- P_hash can be iterated as many times as necessary to produce the required quantity of data.
- Each iteration involves two executions of HMAC—each of which in turn involves two executions of the 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
  PRF(secret, label, seed) = P_hash(S1,label || 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 bytes, or the ciphertext decrypted to a length of greater than 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.

- **certificate_verify and Finished Messages**

  In the TLS certificate_verify message, the MD5 and SHA-1 hashes are calculated only over handshake_messages. 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.

- **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.

2.4 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.
- 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

- 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.
- At the HTTP level, an HTTP client requests a connection to an HTTP server by sending a connection request to the next lowest layer.
- 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.

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".

- 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. 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.

2.5 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.
- 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 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 2.8):
  - **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.

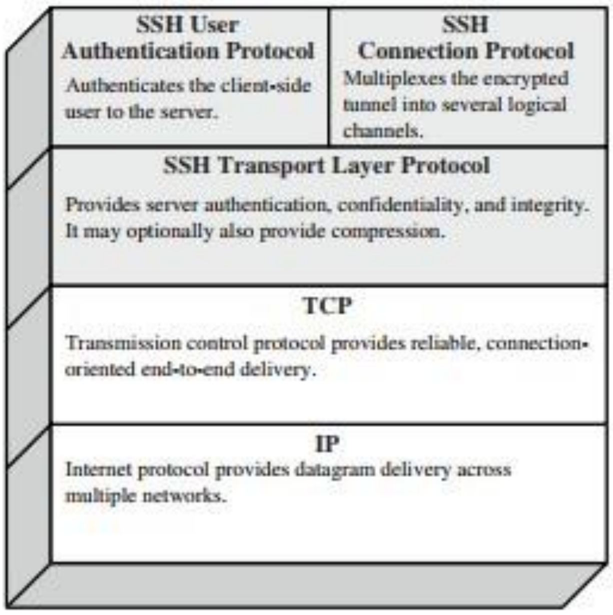

SSH User Authentication Protocol
Authenticates the client-side user to the server.
SSH Connection Protocol
Multiplexes the encrypted tunnel into several logical channels.
SSH Transport Layer Protocol
Provides server authentication, confidentiality, and integrity. It may optionally also provide compression.
TCP
Transmission control protocol provides reliable, connection-oriented end-to-end delivery.
IP
Internet protocol provides datagram delivery across multiple networks.

Figure 2.8 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:

- The server host key is used during key exchange to authenticate the identity of the host. 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. There are two alternative trust models that can be used:

  1. The client has a local database that associates each host name with the corresponding public host key. This method requires no centrally administered infrastructure and no third-party coordination. The drawback 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.

❖ PACKET EXCHANGE

- Figure 2.10 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 (Fig 2.9).

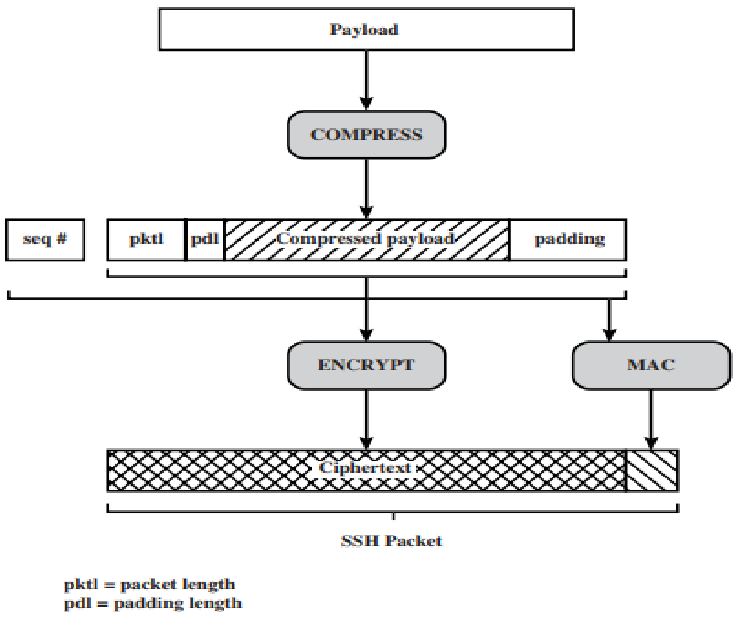


Figure 2.9 SSH Transport Layer Protocol Packet Formation

• **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 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 zero for the first packet and incremented for every packet.

- 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 2.10).

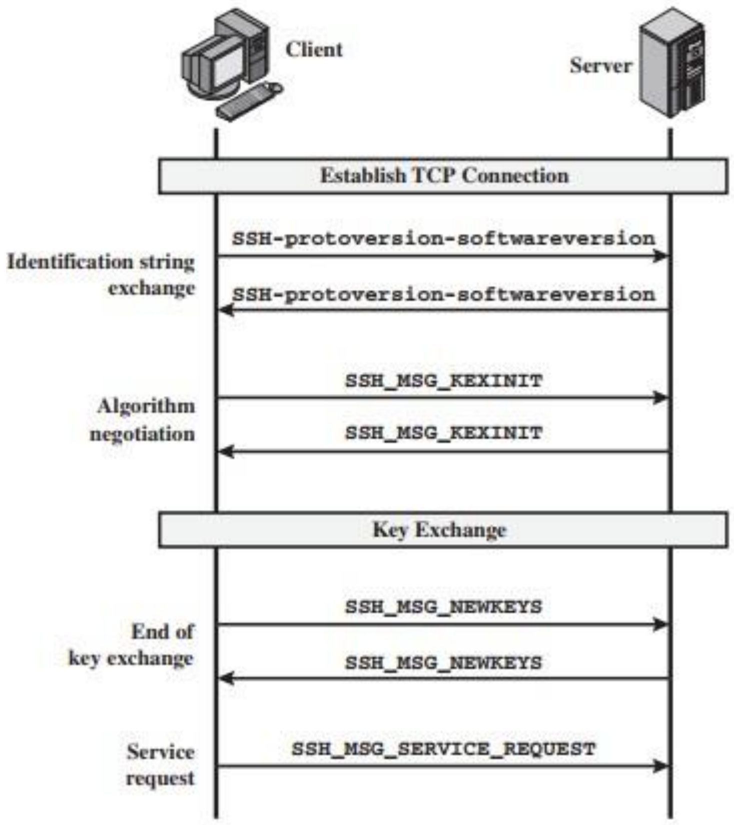


Figure 2.10 SSH Transport Layer Protocol Packet Exchanges

- The first step, the **identification string exchange,** begins with the client sending a packet with an identification string.

- 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. For each category, the algorithm chosen is the first algorithm on the client's list that is also supported by the server.
- 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 **end of key exchange** is signalled by the exchange of SSH_MSG_NEWKEYS packets.
- 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.

➢ 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 SSH_MSG_USERAUTH_REQUEST (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 rejects the authentication request or accepts the request but requires one or more additional authentication methods, the server sends a message with the format:

byte	**SSH_MSG_USERAUTH_FAILURE** (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. The secure authentication 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. 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.
  - 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 2.11 provides an example of Connection Protocol Message Exchange.

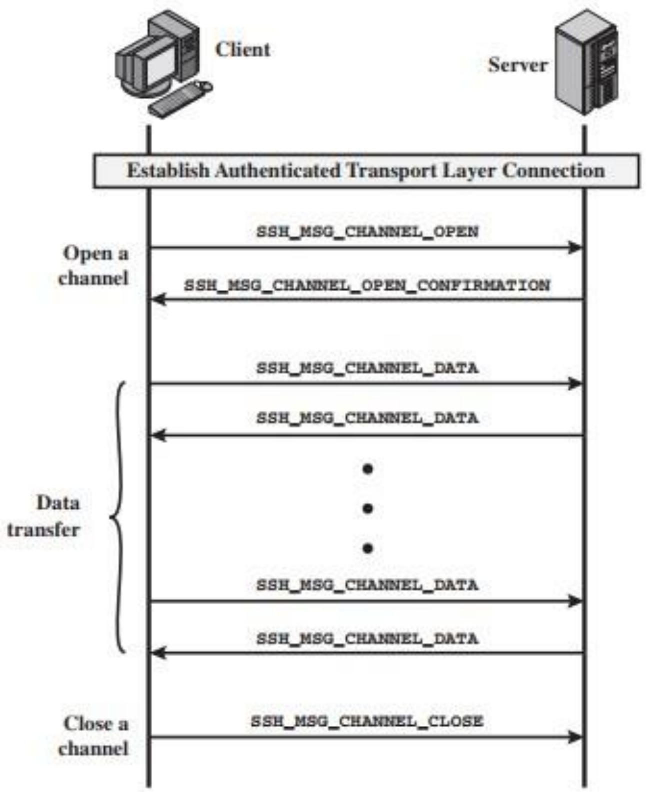


Figure 2.11 Example SSH Connection Protocol Message Exchange

❖ **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.
- **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
- **direct-tcpip:** This is local port forwarding,

PORT FORWARDING

- One of the most useful features of SSH is port forwarding. Port forwarding provides the ability to convert any insecure TCP connection into a secure SSH connection. This is also referred to as **SSH tunnelling.**
- 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 2.12 illustrates the basic concept behind port forwarding.

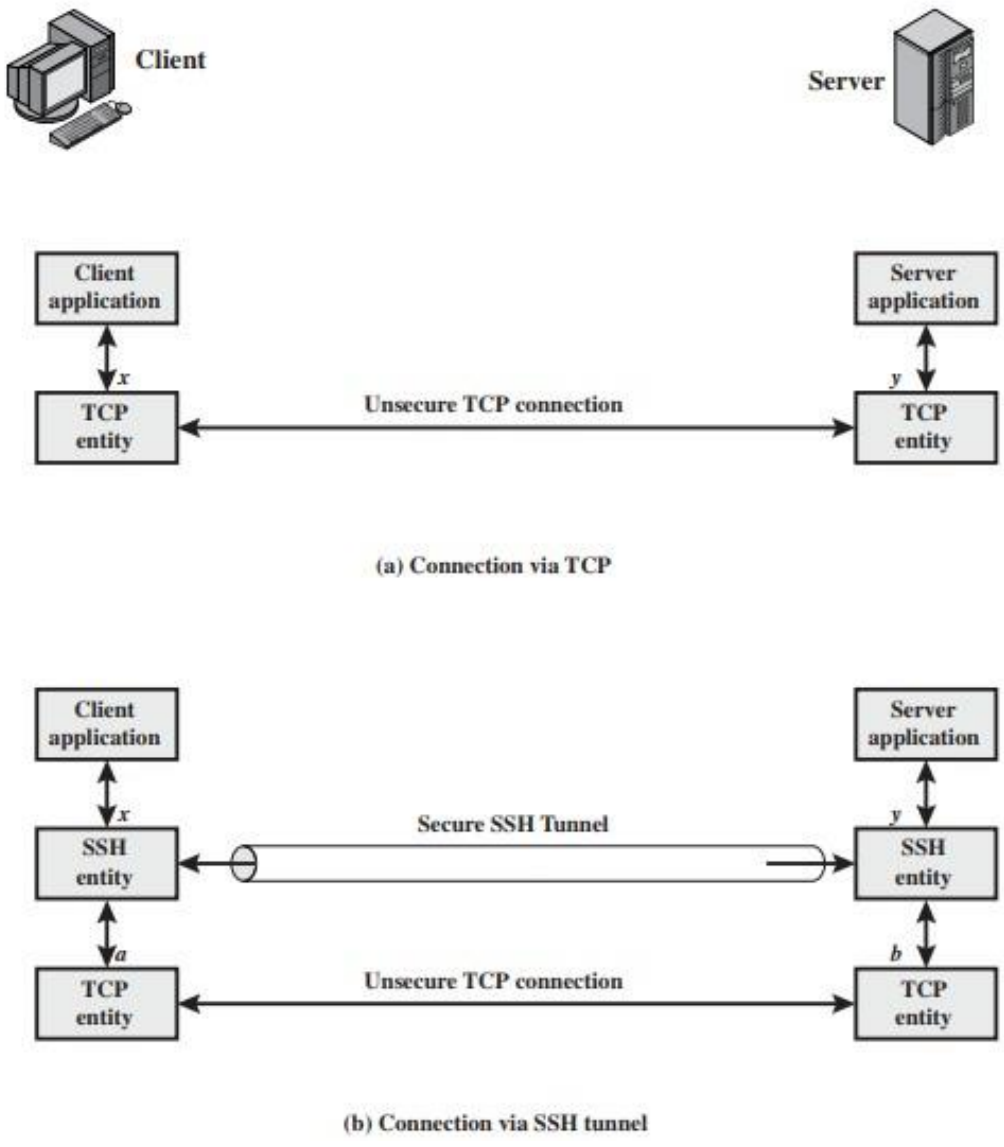


Figure 2.12 SSH Transport Layer Packet Exchanges

- There is 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 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 illustrates 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 mail server 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.

- **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 example of remote forwarding is illustrated below. 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 logon to the work server.