CHAPTER 16 – TRANSPORT-LEVEL SECURITY

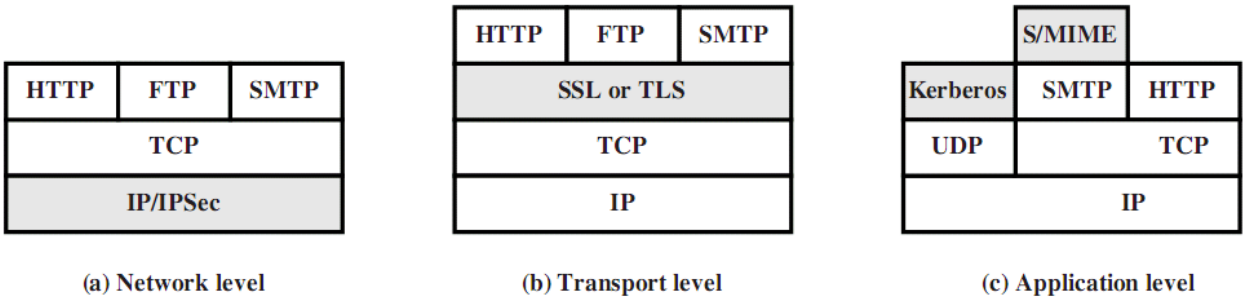
Web Security Considerations:

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 the Web presents new challenges not generally appreciated in the context of computer and network security.

- The Internet is two-way. 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.
- 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 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. Stallings Figure 16.1 illustrates this difference.



(a) Network level (b) Transport level (c) Application level

Figure 16.1 Relative Location of Security Facilities in the TCP/IP Protocol Stack

One way to provide Web security is to use IP Security (Figure 16.1a). The advantage of using IPSec is that it is transparent to end users and applications and provides a general-purpose solution. Further, IPSec includes a filtering capability so only selected traffic need incur the IPSec processing overhead.

Another relatively general-purpose solution is to implement security just above TCP (Figure 16.1b). The foremost 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, e.g. both the Netscape and Microsoft Explorer browsers come with SSL, & most Web servers have implemented it.

Application-specific security services are embedded within the particular application. Figure 16.1c 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.

SSL (Secure Socket Layer)

SSL probably most widely used Web security mechanism, and it is implemented at the Transport layer

SSL is designed to make use of TCP to provide a reliable end-to-end secure service. N etscape 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. SSL is not a single protocol but rather two layers of protocol, as shown next.

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 16.2.

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 also defined as part of SSL: the Handshake Protocol, Change Cipher Spec Protocol, and Alert Protocol. These SSL-specific protocols are used in the management of SSL exchanges.

Two important SSL concepts are the SSL connection and the SSL session:

• **Connection**: A connection is a network transport that provides a suitable type of service, such connections are transient, peer-to-peer relationships, associated with one session

• **Session:** An SSL session is an association between a client and a server, 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.

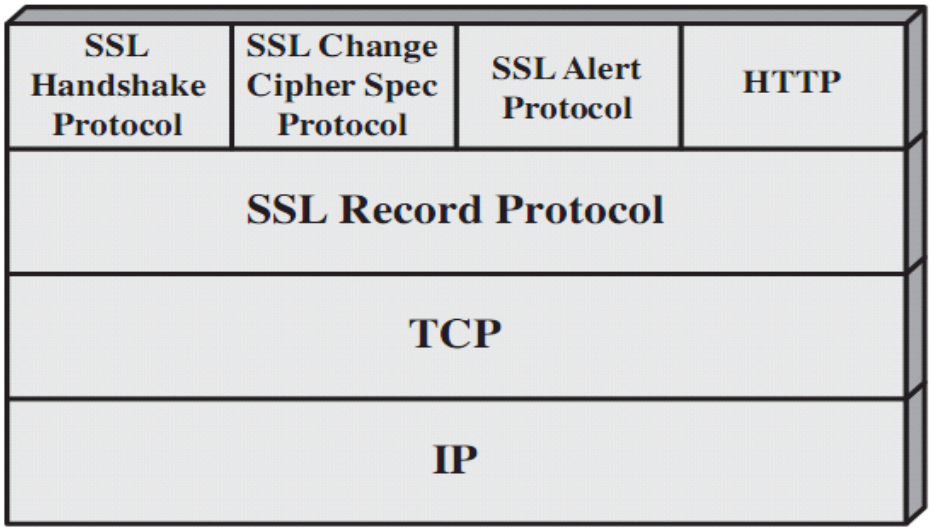


Figure 17.2 SSL Protocol Stack

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.

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 and a connection state are defined by sets of parameters, see textbook for details.

SSL Record Protocol Services

SSL Record Protocol defines two services for SSL connections:

• **Confidentiality:** The Handshake Protocol defines a shared secret key that is used for conventional encryption of SSL payloads. The message is compressed before being concatenated with the MAC and encrypted, with a range of ciphers being supported as shown.

• **Message Integrity:** The Handshake Protocol also defines a shared secret key that is used to form a message authentication code (MAC), which is similar to HMAC

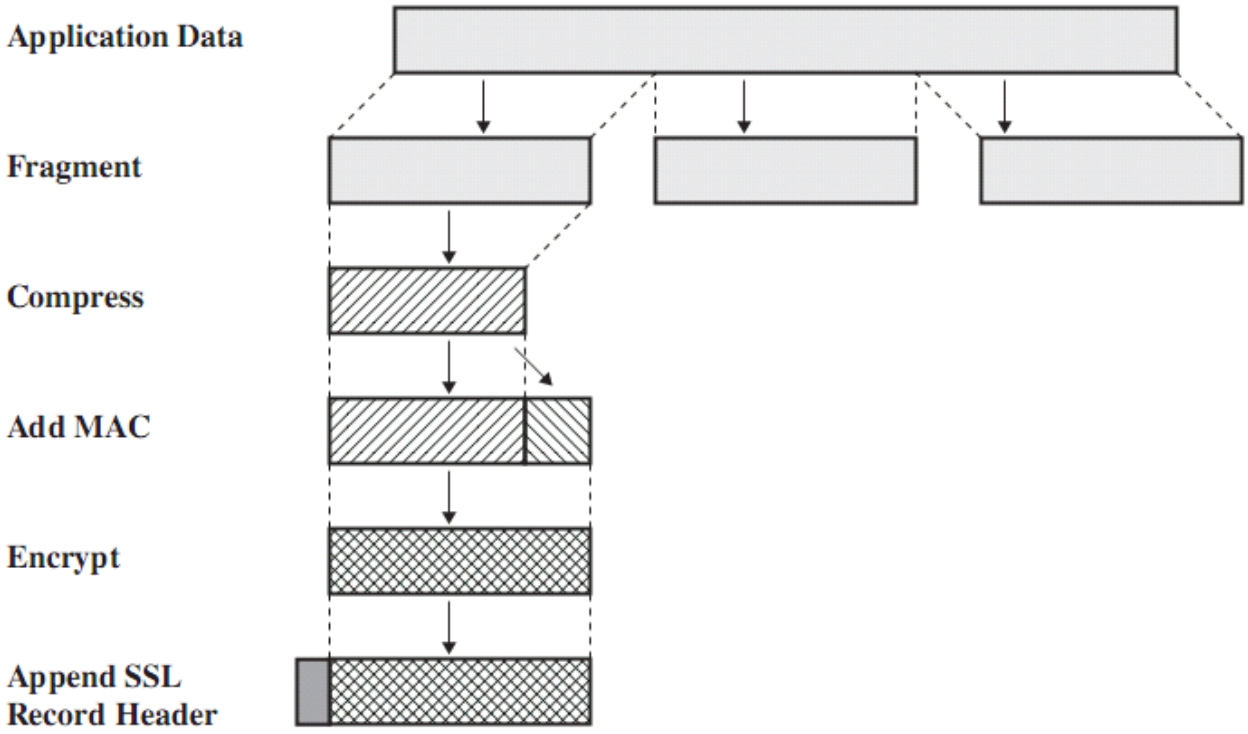


Fig: 17.3 SSL Record protocol Operation.

Figure 17.3 shows 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, computes and **appends a MAC** (using a hash very similar to HMAC), **encrypts** (using one of the symmetric algorithms listed on the previous slide), **adds a header** (with details of the SSL content type, major/minor version, and compressed length), and transmits the resulting unit in a TCP segment. Received data are decrypted, verified, decompressed, and reassembled and then delivered to higher-layer applications.

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.

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

```
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)

Next, the compressed message plus the MAC are **encrypted** using symmetric encryption.

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 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.

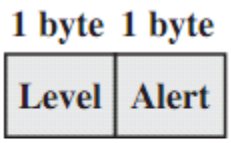
• **Minor Version (8 bits):** Indicates minor version in use. For SSLv3, the value is 0.

• **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.

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, consisting of a single message (shown in Stallings Figure 16.5a), 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.

(a) Change Cipher Spec Protocol



(b) Alert Protocol

SSL Alert Protocol

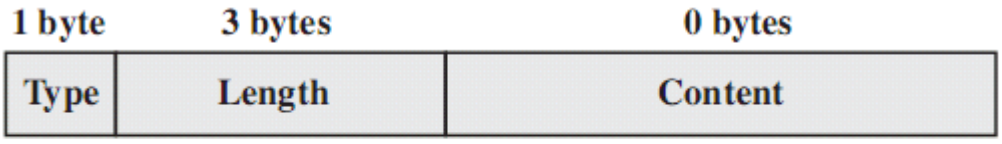
The Alert Protocol is used to convey SSL-related alerts to the peer entity. As with other applications that use SSL, alert messages are compressed and encrypted, as specified by the current state.

Each message in this protocol consists of two bytes (as shown in Stallings Figure 16.5b), the first takes the value warning(1) or fatal(2) to convey the severity of the message. The second byte contains a code that indicates the specific alert.

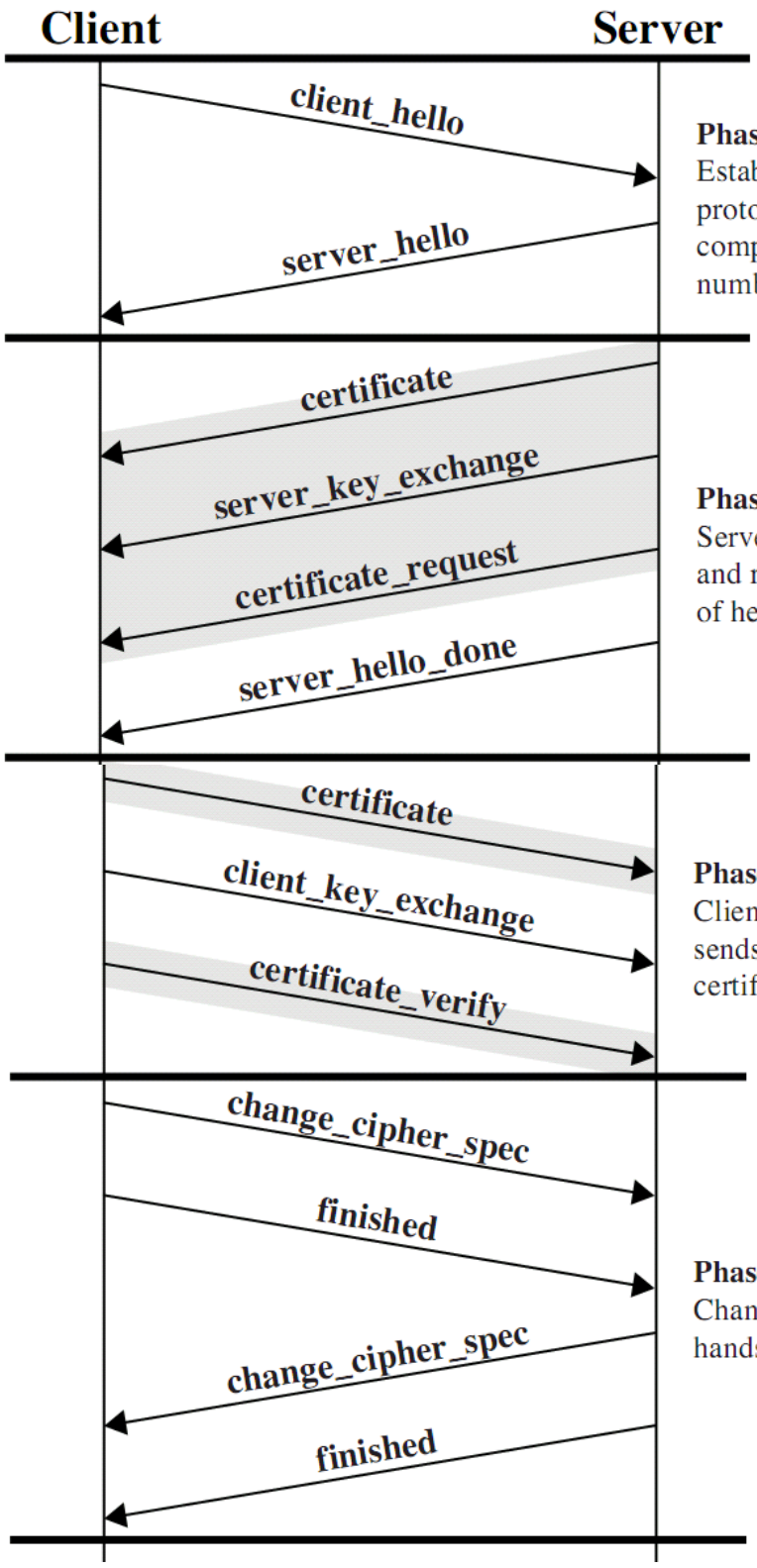
- **Fatal errors:** unexpected message, bad record MAC, decompression failure, handshake failure, illegal parameter
- **warning:** close notify, no certificate, bad certificate, unsupported certificate, certificate revoked, certificate expired, certificate unknown

SSL 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, which have the format shown in Stallings Figure 16.5c, and which can be viewed in 4 phases:



(c) Handshake Protocol



Phase 1
Establish security capabilities, including protocol version, session ID, cipher suite, compression method, and initial random numbers.

Phase 2
Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase.

Phase 3
Client sends certificate if requested. Client sends key exchange. Client may send certificate verification.

Phase 4
Change cipher suite and finish handshake protocol.

Phase 1. Establish Security Capabilities:

This phase is used by the client 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
- **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.

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 is supported.

• RSA: The secret key is encrypted with the receiver's RSA public key.

• 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.

• **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.

• **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.

Phase 2. Server Authentication and Key Exchange:

The server begins this phase by sending **its certificate** if it needs to be authenticated.

- Next, a *server_key_exchange* message may be sent if it is required.
- 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 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.

Phase 3. Client Authentication and Key Exchange:

The client should verify that the server provided a valid certificate if required and check that the *server_hello* parameters are acceptable.

- 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.
- 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).

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.
- 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.
- At this point the handshake is complete and the client and server may begin to exchange application layer data.

****