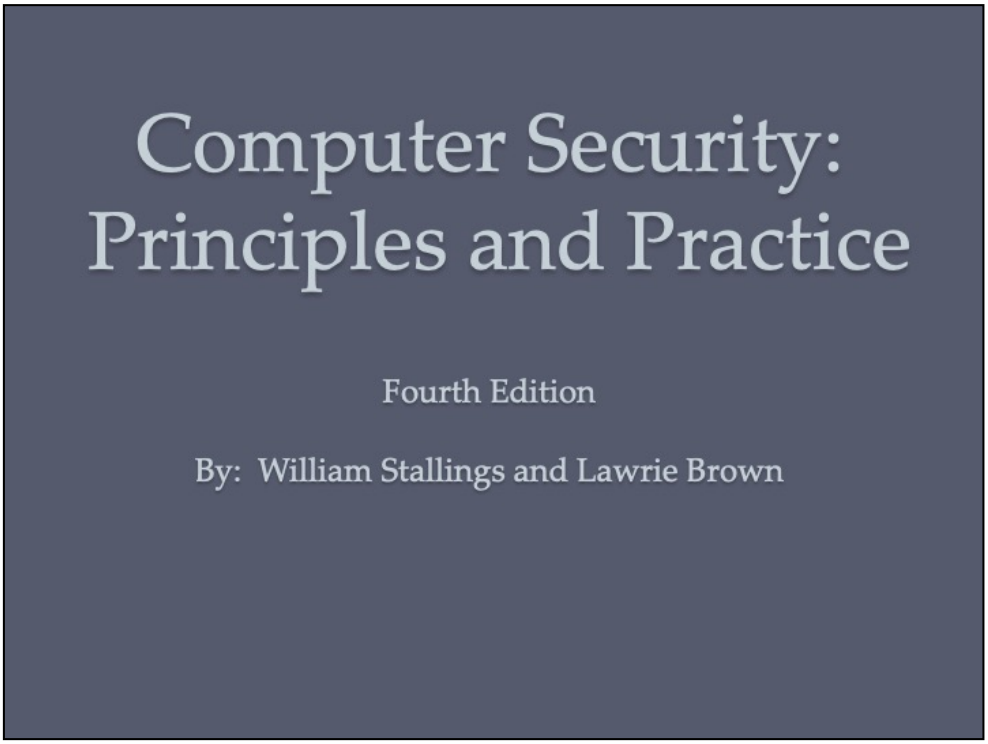
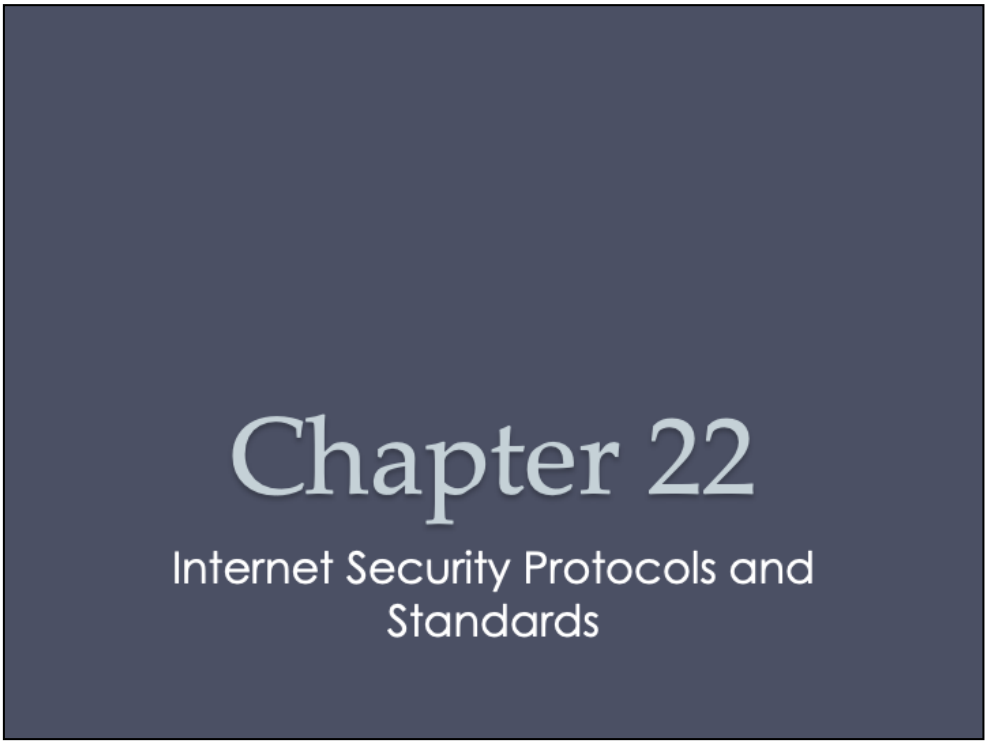
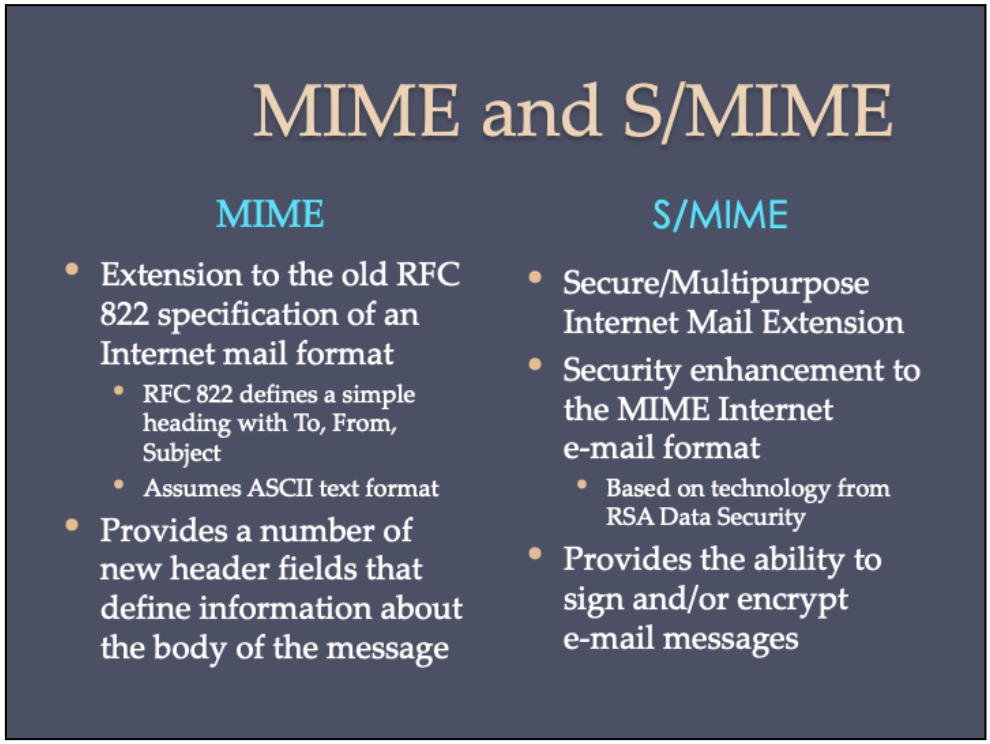


Lecture slides prepared for "Computer Security: Principles and Practice", 4/e, by William Stallings and Lawrie Brown, Chapter 22 "Internet Security Protocols and Standards".



This chapter looks at some of the most widely used and important Internet security protocols and standards.



S/MIME (Secure/Multipurpose Internet Mail Extension) is a security enhancement to the MIME Internet e-mail format standard.

MIME is an extension to the old RFC 822 (*Standard For The Format of ARPA Internet Text Messages,* 1982:

specification of an Internet mail format. RFC 822 defines a simple header with To, From, Subject, and other fields that can

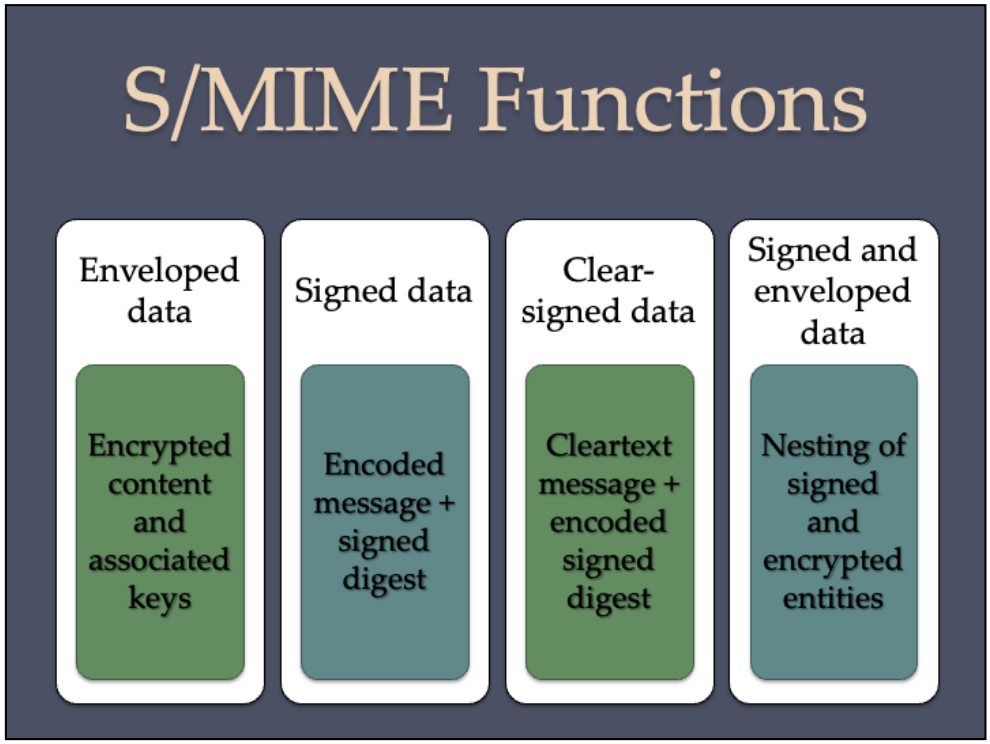
be used to route an e-mail message through the Internet and that provides basic information about the e-mail content. RFC 822 assumes a simple ASCII text format for the content.

MIME provides a number of new header fields that define information about the body of the message, including the format of the body and any encoding that is done to facilitate transfer. Most important, MIME defines a number of content formats, which standardize representations for the support of multimedia e-mail. Examples include text, image, audio, and video.

Table 22.1
S/MIME Content Types

Type	Subtype	smime Parameter	Description
Multipart	Signed		A clear-signed message in two parts: one is the message and the other is the signature.
Application	pkcs7-mime	signedData	A signed S/MIME entity.
	pkcs7-mime	envelopedData	An encrypted S/MIME entity.
	pkcs7-mime	degenerate signedData	An entity containing only public-key certificates.
	pkcs7-mime	CompressedData	A compressed S/MIME entity.
	pkcs7-signature	signedData	The content type of the signature subpart of a multipart/signed message.

S/MIME functionality is built into the majority of modern e-mail software and interoperates between them. S/MIME is defined as a set of additional MIME content types (see Table 22.1) and provides the ability to sign and/or encrypt e-mail messages.



In essence, these content types support four new functions:

• **Enveloped data:** This function consists of encrypted content of any type and encrypted-content encryption keys for one or more recipients.

• **Signed data:** A digital signature is formed by taking the message digest of the content to be signed and then encrypting that with the private key of the signer. The content plus signature are then encoded using base64 encoding. A signed data message can only be viewed by a recipient with S/MIME capability.

• **Clear-signed data:** As with signed data, a digital signature of the content is formed. However, in this case, only the digital signature is encoded using base64. As a result, recipients without S/MIME capability can view the message content, although they cannot verify the signature.

• **Signed and enveloped data:** Signed-only and encrypted-only entities may be nested, so that encrypted data may be signed and signed data or clear-signed

data may be encrypted.

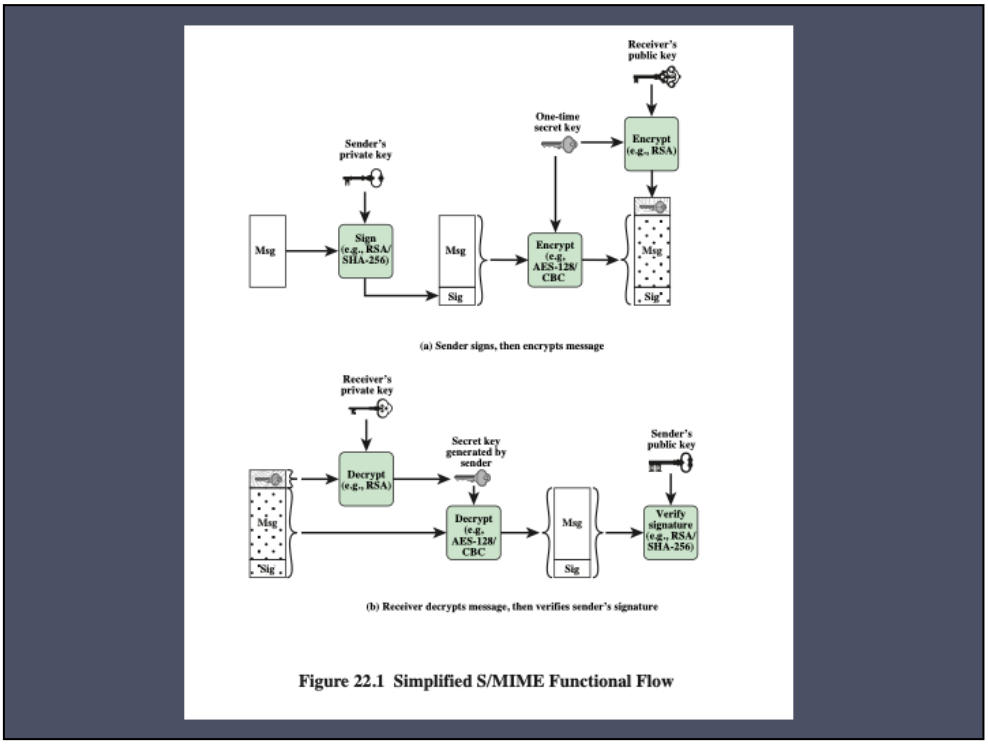
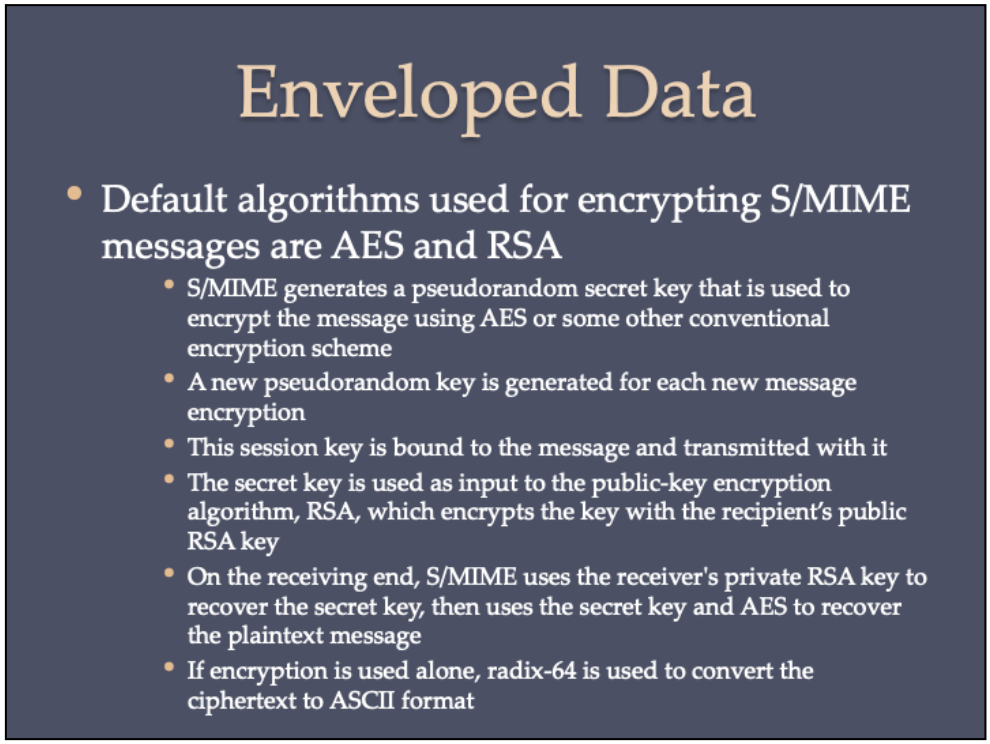


Figure 22.1 Simplified S/MIME Functional Flow

Figure 22.1 provides a general overview of S/MIME functional flow.

The preferred algorithms used for signing S/MIME messages use either an RSA or a Digital Signature Algorithm (DSA) signature of an SHA-256 message hash. The process works as follows. Take the message you want to send and map it into a fixed-length code of 256 bits, using SHA-256. The 256-bit message digest is, for all practical purposes, unique for this message. It would be virtually impossible for someone to alter this message or substitute another message and still come up with the same digest. Then, S/MIME encrypts the digest using RSA and the sender's private RSA key. The result is the digital signature, which is attached to the message, as we discuss in Chapter 2. Now, anyone who gets this message can recompute the message digest then decrypt the signature using RSA and the sender's public RSA key. If the message digest in the signature matches the message digest that was calculated, then the signature is valid. Since this operation only involves encrypting and decrypting a 256-bit block, it takes up little time. The DSA can be used instead of RSA as the signature algorithm.

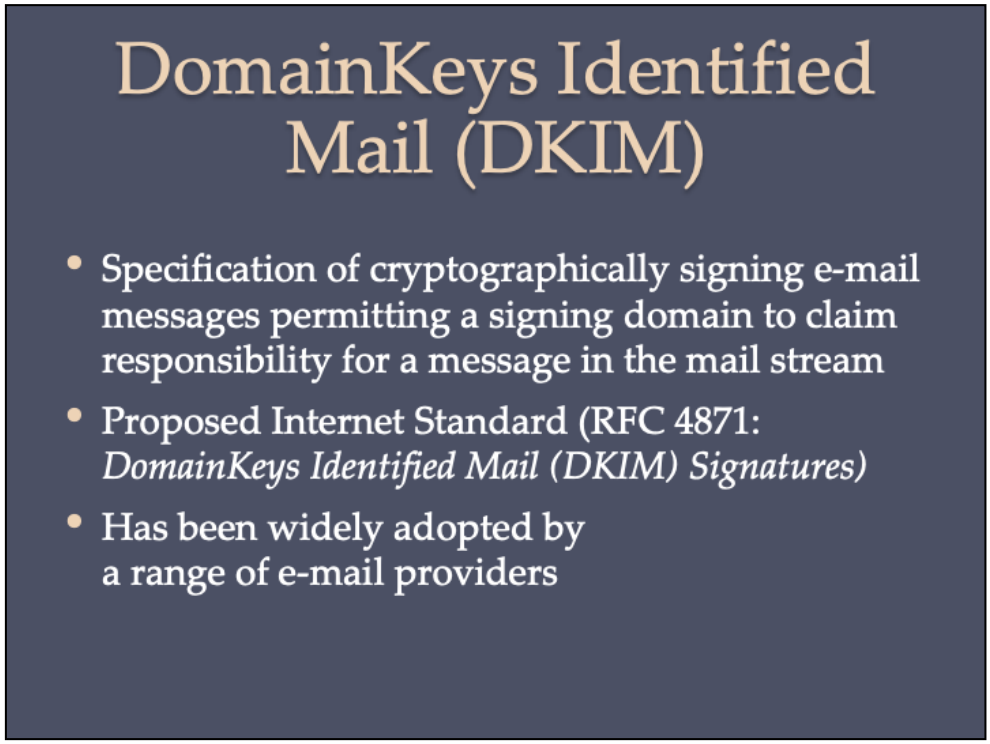
The signature is a binary string, and sending it in that form through the Internet e-mail system could result in unintended alteration of the contents, because some e-mail software will attempt to interpret the message content looking for control characters such as line feeds. To protect the data, either the signature alone or the signature plus the message are mapped into printable ASCII characters using a scheme known as radix-64 or base64 mapping. Radix-64 maps each input group of three octets of binary data into four ASCII characters (see Appendix G).



The default algorithms used for encrypting S/MIME messages are AES and RSA. To begin, S/MIME generates a pseudorandom secret key; this is used to encrypt the message using AES or some other conventional encryption scheme, such as 3DES. In any conventional encryption application, the problem of key distribution must be addressed. In S/MIME, each conventional key is used only once. That is, a new pseudorandom key is generated for each new message encryption. This session key is bound to the message and transmitted with it. The secret key is used as input to the public-key encryption algorithm, RSA, which encrypts the key with the recipient's public RSA key.

On the receiving end, S/MIME uses the receiver's private RSA key to recover the secret key, then uses the secret key and AES to recover the plaintext message.

If encryption is used alone, radix-64 is used to convert the ciphertext to ASCII format.



DomainKeys Identified Mail (DKIM) is a specification for cryptographically signing e-mail messages, permitting a signing domain to claim responsibility for a message in the mail stream. Message recipients (or agents acting in their behalf) can verify the signature by querying the signer's domain directly to retrieve the appropriate public key and thereby can confirm that the message was attested to by a party in possession of the private key for the signing domain. DKIM is specified in Internet Standard RFC 4871: (*DomainKeys Identified Mail (DKIM) Signatures). DKIM has* been widely adopted by a range of e-mail providers, including corporations, government agencies, Gmail, yahoo, and many Internet service providers (ISPs).

To understand the operation of DKIM, it is useful to have a basic grasp of the Internet mail architecture, which is currently defined in RFC 5598 (Internet Mail Architecture , 2009). This subsection provides an overview of the basic concepts.

At its most fundamental level, the Internet mail architecture consists of a user world in the form of Message User Agents (MUA), and the transfer world, in the form of the Message Handling Service (MHS), which is composed of Message Transfer Agents (MTA). The MHS accepts a message from one user and delivers it to one or more other users, creating a virtual MUA-to-MUA exchange environment. This architecture involves three types of interoperability. One is directly between users: messages must be formatted by the MUA on behalf of the message author so that the message can be displayed to the message recipient by the destination MUA. There are also interoperability requirements between the MUA and the MHS—first when a message is posted from an MUA to the MHS and later when it is delivered from the MHS to the destination MUA. Interoperability is required among the MTA components along the transfer path through the MHS.

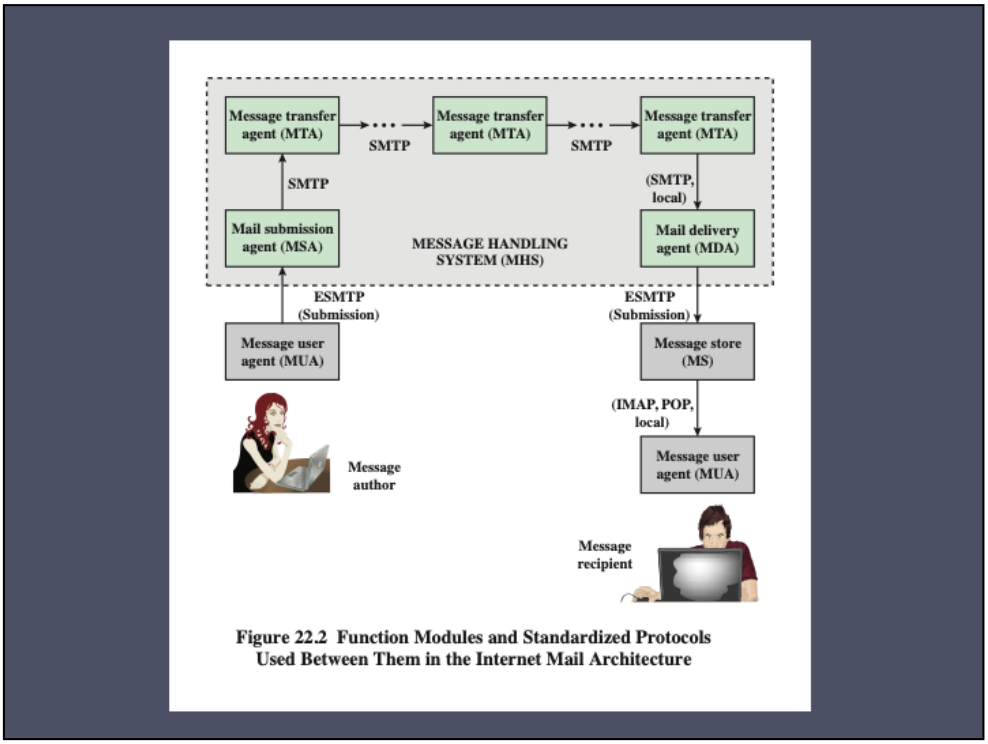


Figure 22.2 Function Modules and Standardized Protocols Used Between Them in the Internet Mail Architecture

Figure 22.2 illustrates the key components of the Internet mail architecture, which include the following.

• **Message user agent (MUA):** Works on behalf of user actors and user applications. It is their representative within the e-mail service. Typically, this function is housed in the user's computer and is referred to as a client e-mail program or a local network e-mail server. The author MUA formats a message and performs initial submission into the MHS via a MSA. The recipient MUA processes received mail for storage and/or display to the recipient user.

• **Mail submission agent (MSA):** Accepts the message submitted by an MUA and enforces the policies of the hosting domain and the requirements of Internet standards. This function may be located together with the MUA or as a separate functional model. In the latter case, the Simple Mail Transfer Protocol (SMTP) is used between the MUA and the MSA.

• **Message transfer agent (MTA):** Relays mail for one application-level hop. It is like a packet switch or IP router in that its job is to make routing assessments and to move the message closer to the recipients. Relaying is performed by a sequence of MTAs until the message reaches a destination MDA. An MTA also adds trace information to the message header. SMTP is used between MTAs and between an MTA and an MSA or MDA.

• **Mail delivery agent (MDA):** Responsible for transferring the message from the MHS to the MS.

• **Message store (MS):** An MUA can employ a long-term MS. An MS can be located on a remote server or on the same machine as the MUA. Typically, an MUA retrieves messages from a remote server using POP (Post Office Protocol) or IMAP (Internet Message Access Protocol).

Two other concepts need to be defined. An **administrative management domain (ADMD)** is an Internet e-mail provider. Examples include a department that operates a local mail relay (MTA), an IT department that operates an enterprise mail relay, and an ISP that operates a public shared e-mail service. Each ADMD can have different operating policies and trust-based decision making. One obvious example is the distinction between mail that is exchanged within an organization and mail that is exchanged between independent organizations. The rules for handling the two types of traffic tend to be quite different.

The Domain name system (DNS) is a directory lookup service that provides a mapping between the name of a host on the Internet and its numerical address.

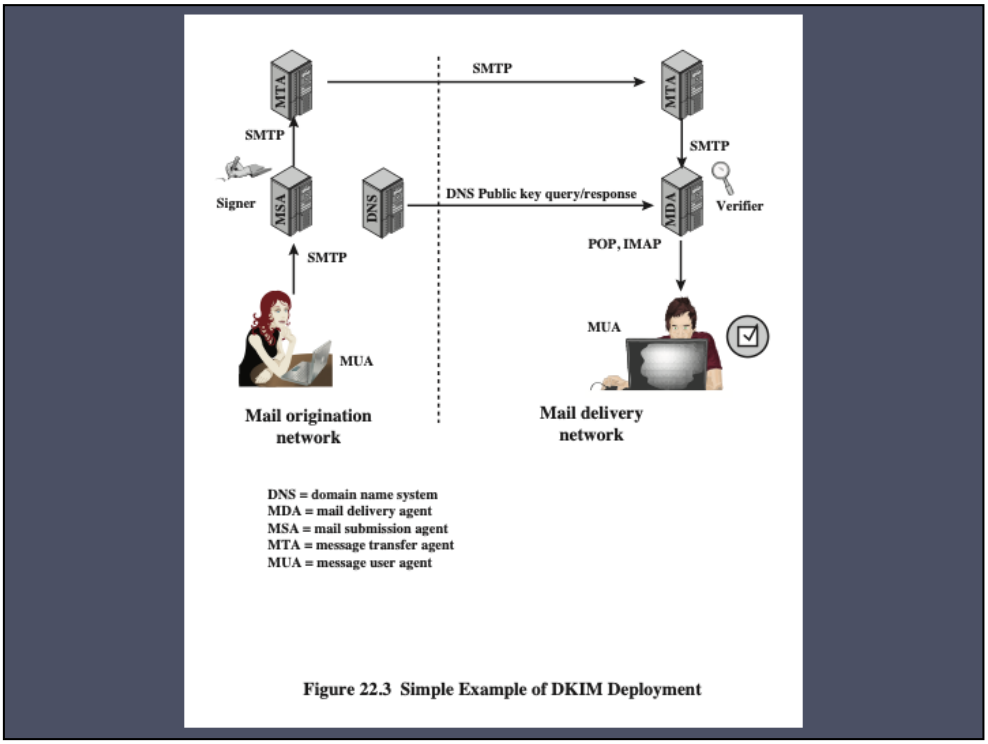
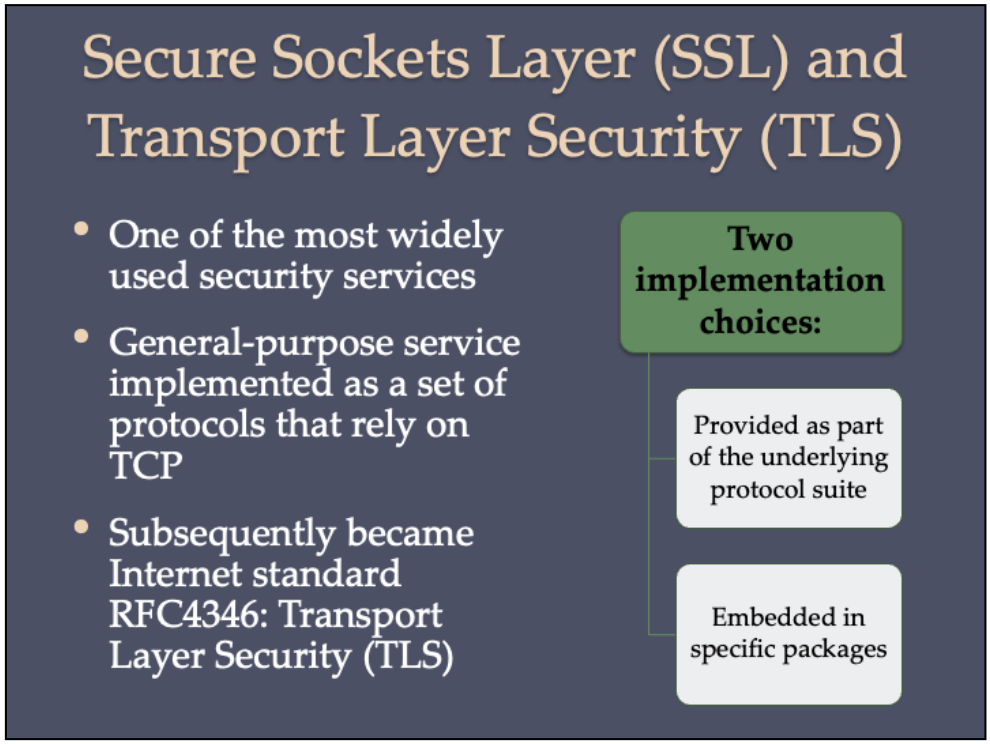


Figure 22.3 Simple Example of DKIM Deployment

Figure 22.3 is a simple example of the operation of DKIM. We begin with a message generated by a user and transmitted into the MHS to an MSA that is within the user's administrative domain. An e-mail message is generated by an e-mail client program. The content of the message, plus selected RFC 5322 headers, is signed by the e-mail provider using the provider's private key. The signer is associated with a domain, which could be a corporate local network, an ISP, or a public e-mail facility such as Gmail. The signed message then passes through the Internet via a sequence of MTAs. At the destination, the MDA retrieves the public key for the incoming signature and verifies the signature before passing the message on to the destination e-mail client. The default signing algorithm is RSA with SHA-256. RSA with SHA-1 also may be used.



One of the most widely used security services is the Secure Sockets Layer (SSL) and the follow-on Internet standard RFC 4346 (The Transport Layer Security (TLS) Protocol

Version 1.1 , 2006). TLS has largely supplanted earlier SSL implementations.

TLS is a general-purpose service implemented as a set of protocols that rely on TCP. At this level, there are two implementation choices. For full generality, TLS could be provided as part of the underlying protocol suite and therefore be transparent to applications. Alternatively, TLS can be embedded in specific packages.

For example, most browsers come equipped with SSL, and most Web servers have implemented the protocol.

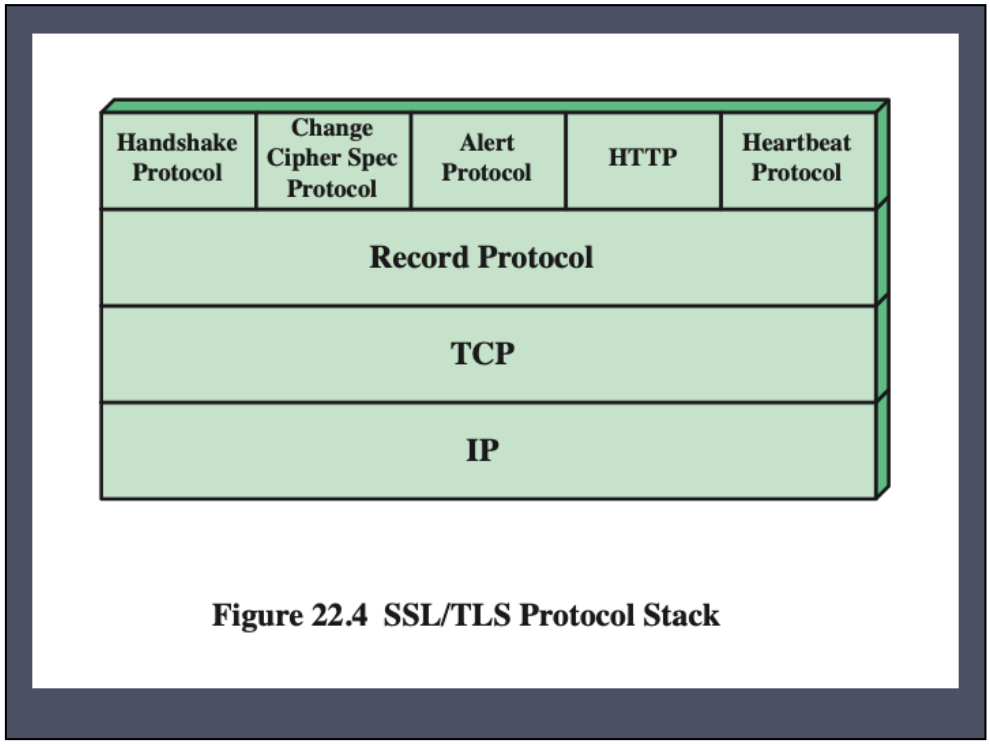
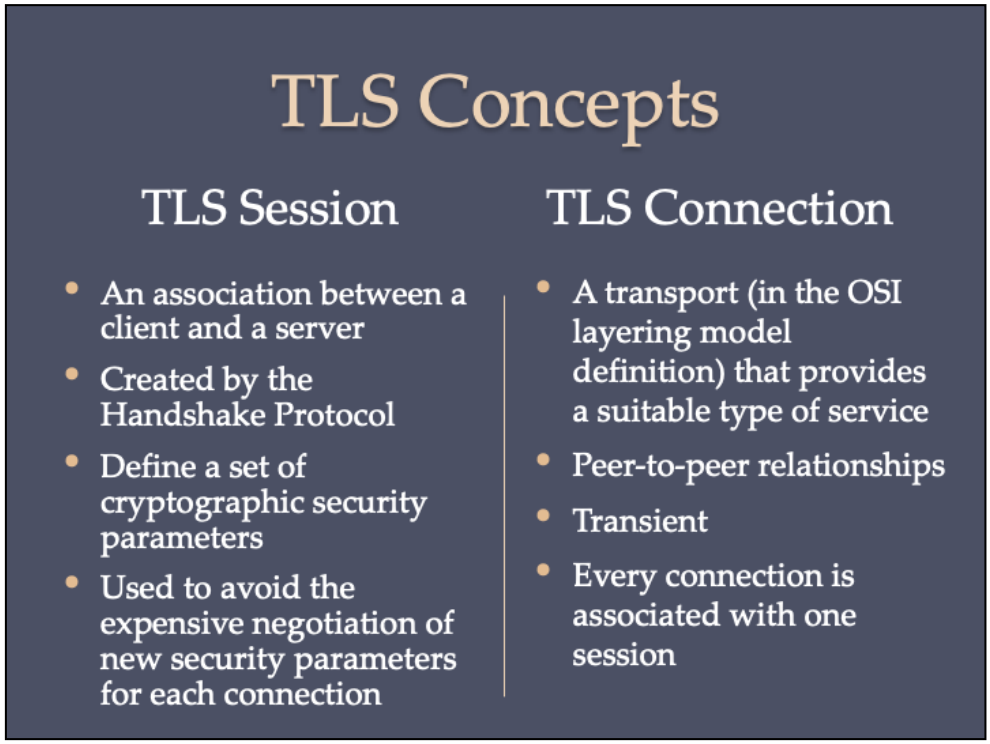


Figure 22.4 SSL/TLS Protocol Stack

TLS is designed to make use of TCP to provide a reliable end-to-end secure service. TLS is not a single protocol but rather two layers of protocols, as illustrated in Figure 22.4.

The 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 TLS. Three higher-layer protocols are defined as part of TLS: the Handshake Protocol, the Change Cipher Spec Protocol, and the Alert Protocol. These TLS-specific protocols are used in the management of TLS exchanges, and are examined later in this section.



Two important TLS concepts are the TLS session and the TLS 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 TLS, such connections are peer-to-peer relationships. The connections are transient. Every connection is associated with one session.

• Session : A TLS 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.

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.

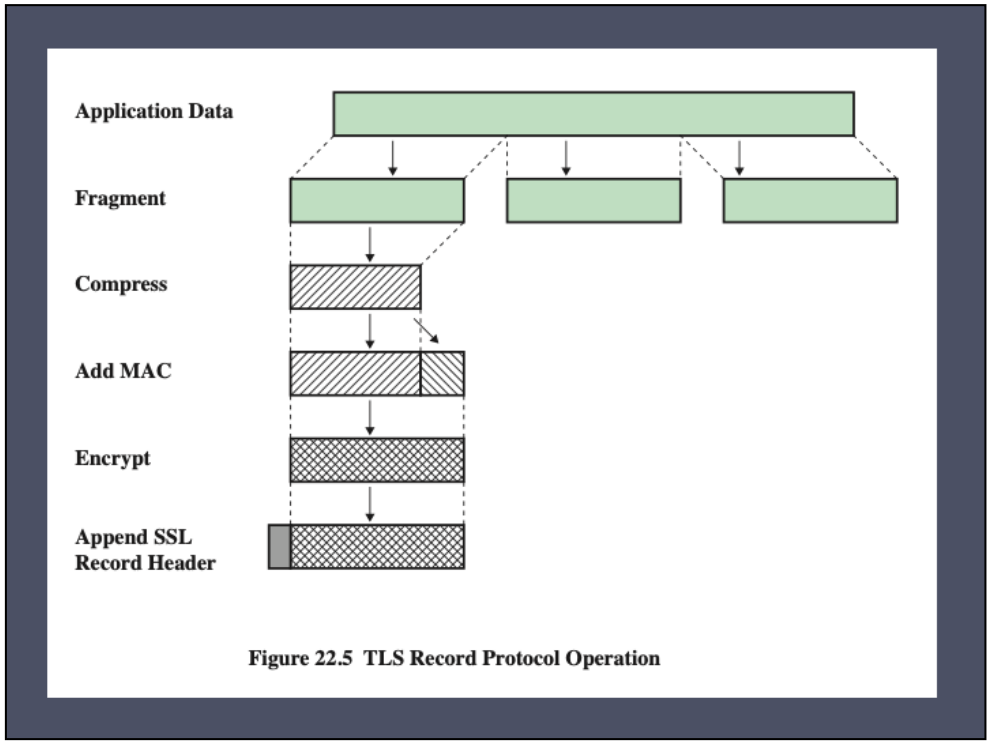


Figure 22.5 TLS Record Protocol Operation

The SSL Record Protocol provides two services for SSL connections:

• **Confidentiality:** The Handshake Protocol defines a shared secret key that is used for symmetric encryption of SSL payloads.

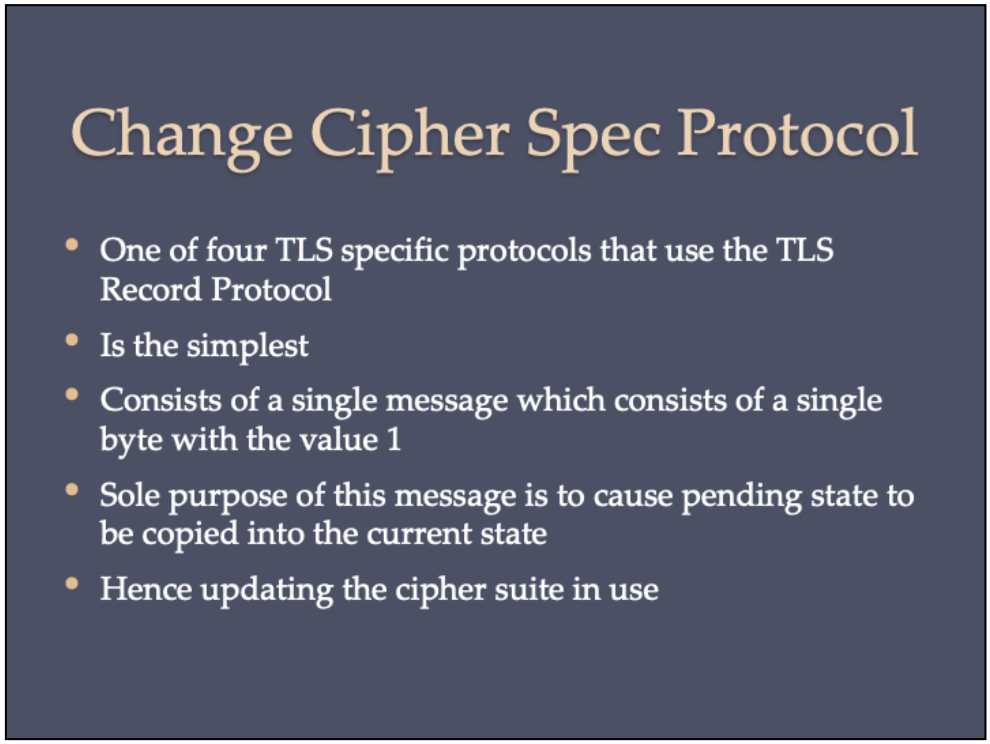
• **Message integrity:** The Handshake Protocol also defines a shared secret key that is used to form a message authentication code (MAC).

Figure 22.5 indicates the overall operation of the SSL Record Protocol. The first step is fragmentation . Each upper-layer message is fragmented into blocks of 214 bytes (16,384 bytes) or less. Next, compression is optionally applied. The next step in processing is to compute a message authentication code over the compressed data. Next, the compressed message plus the MAC are encrypted using symmetric encryption.

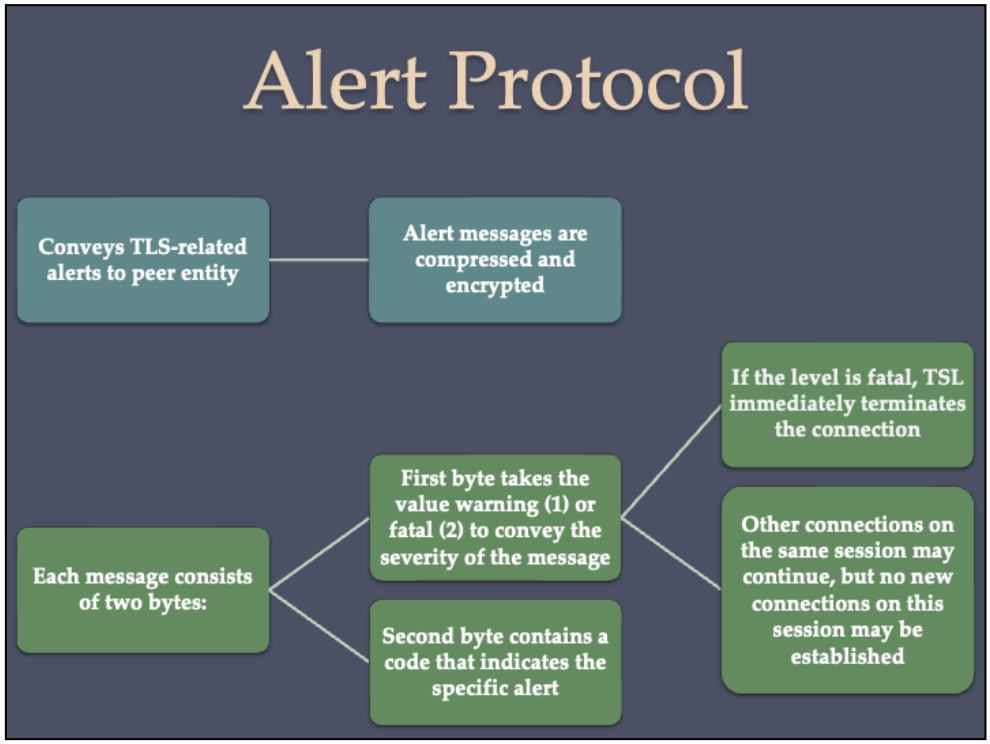
The final step of SSL Record Protocol processing is to prepend a header, which includes version and length fields.

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

The Record Protocol then transmits the resulting unit in a TCP segment. Received data are decrypted, verified, decompressed, and reassembled, and then delivered to higher-level users.

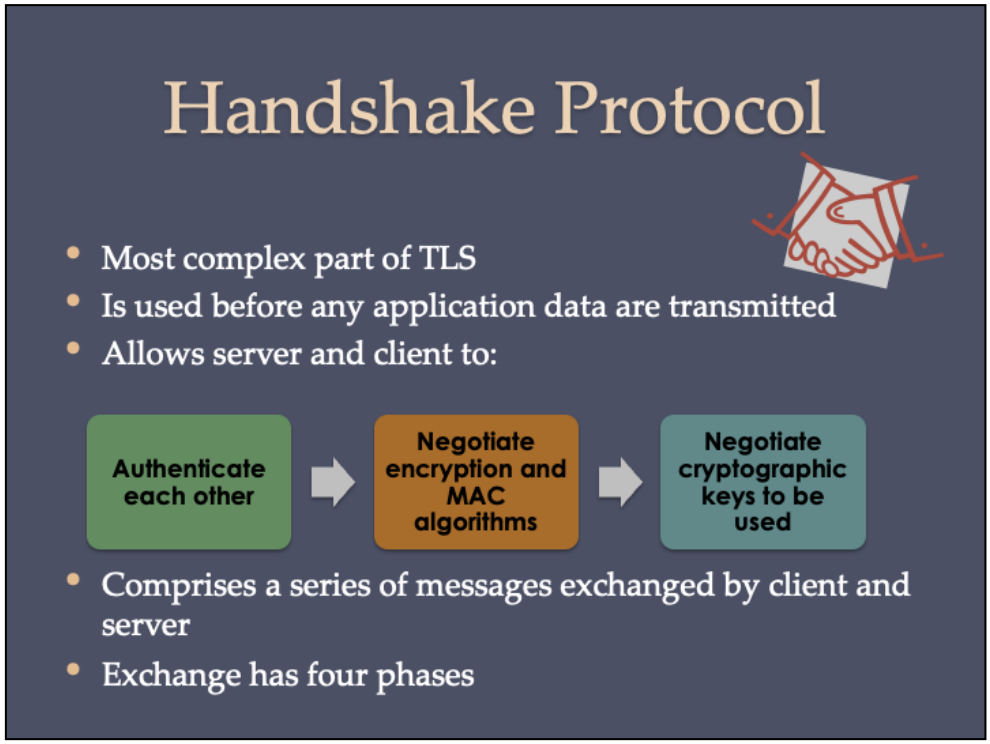


The Change Cipher Spec Protocol is one of the four TLS-specific protocols that use the TLS Record Protocol, and it is the simplest. This protocol consists of a single message, which consists of a single byte with the value 1. The sole purpose of this message is to cause the pending state to be copied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



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

Each message in this protocol consists of two bytes. The first byte takes the value warning(1) or fatal(2) to convey the severity of the message. If the level is fatal, TLS 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. An example of a fatal alert is an incorrect MAC. An example of a nonfatal alert is a close_notify message, which notifies the recipient that the sender will not send any more messages on this connection.



The most complex part of TLS 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 TLS record. The Handshake Protocol is used before any application data are transmitted.

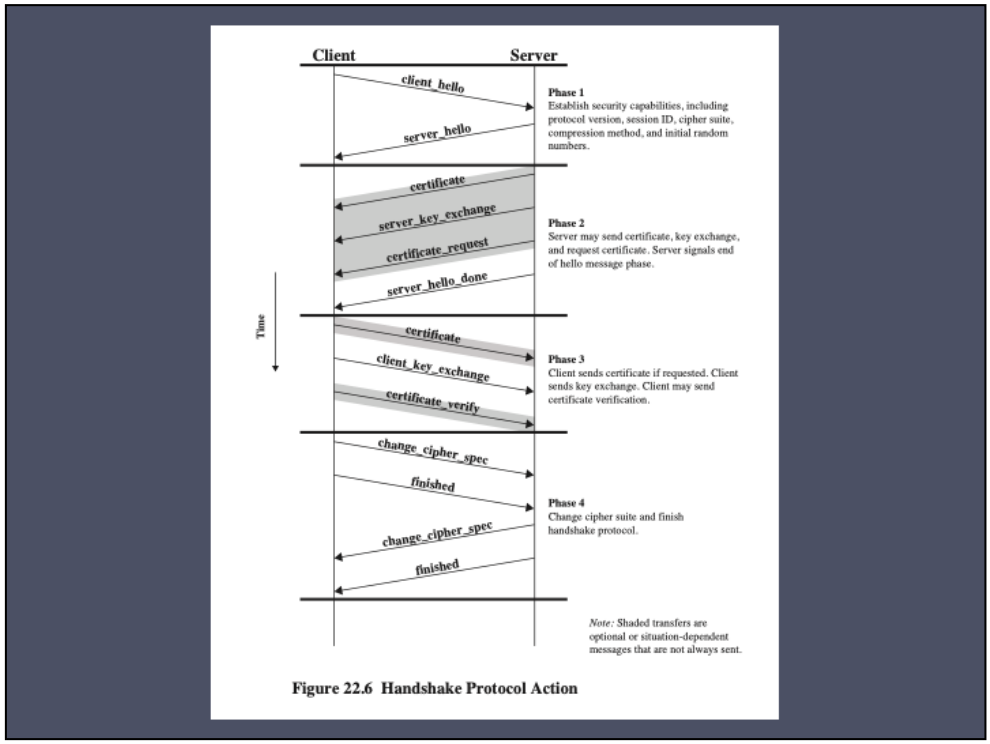


Figure 22.6 Handshake Protocol Action

The Handshake Protocol consists of a series of messages exchanged by client and server. Figure 22.6 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 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 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

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.

• **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 details of **phase 2** depend on the underlying public-key encryption scheme that is used. In some cases, the server passes a certificate to the client, possibly additional
key information, and a request for a certificate from the client.

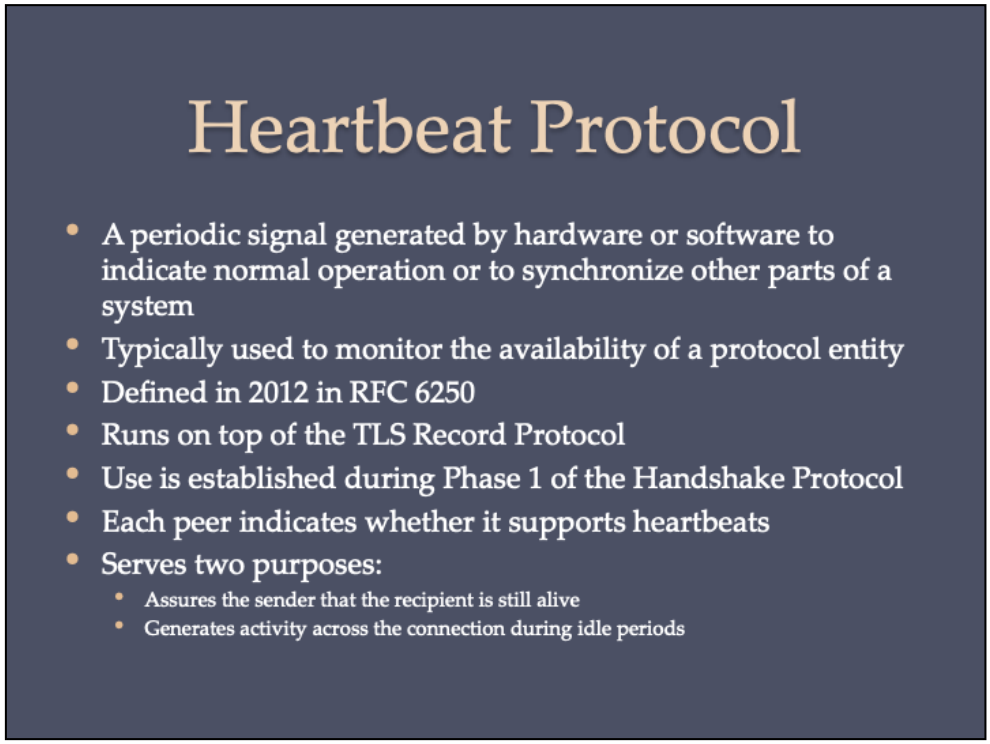
The final message in phase 2, and one that is always required, is the server_ done message, which is sent by the server to indicate the end of the server hello and associated messages. After sending this message, the server will wait for a client response.

In **phase 3 ,** 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, depending on the underlying public-key scheme.

Phase 4 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.

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.



In the context of computer networks, a heartbeat is a periodic signal generated by hardware or software to indicate normal operation or to synchronize other parts of a system. A Heartbeat Protocol is typically used to monitor the availability of a protocol entity. In the specific case of SSL/TLS, a Heartbeat protocol was defined in 2012 in RFC 6250 (*Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension,* 2011 *)*.

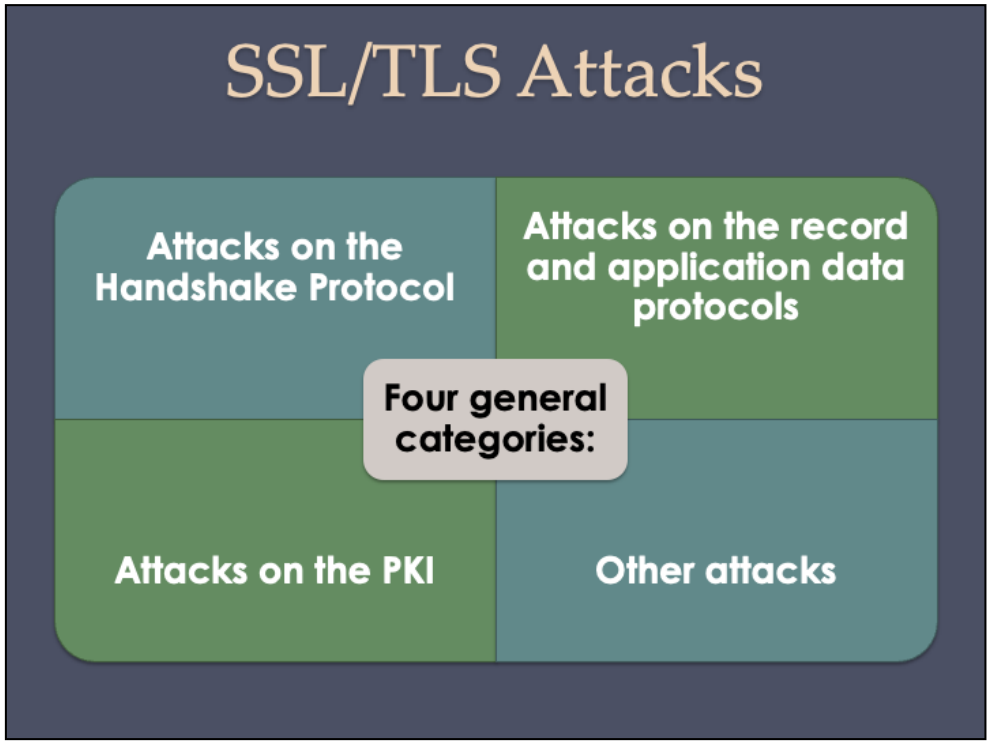
The Heartbeat Protocol runs on the top of the TLS Record Protocol and consists of two message types: heartbeat_request and heartbeat_response. The use of the Heartbeat Protocol is established during Phase 1 of the Handshake Protocol (Figure 22.6). Each peer indicates whether it supports heartbeats. If heartbeats are supported, the peer indicates whether it is willing to receive heartbeat_request messages and respond with heartbeat_response messages or only willing to send heartbeat request messages.

A heartbeat_request message can be sent at any time. Whenever a request message is received, it should be answered promptly with a corresponding heartbeat_

response message. The heartbeat_request message includes payload length, payload, and padding fields. The payload is a random content between 16 bytes and 64 Kbytes in length. The corresponding heartbeat_response message must include an exact copy of the received payload. The padding is also a random content. The padding enables the sender to perform a path maximum transfer unit (MTU) discovery operation, by sending requests with increasing padding until there is no answer anymore, because one of the hosts on the path cannot handle the message.

The heartbeat serves two purposes. First, it assures the sender that the recipient is still alive, even though there may not have been any activity over the underlying TCP connection for a while. Second, the heartbeat generates activity across the connection during idle periods, which avoids closure by a firewall that does not tolerate idle connections.

The requirement for the exchange of a payload was designed into the Heartbeat Protocol to support its use in a connectionless version of TLS known as DTLS. Because a connectionless service is subject to packet loss, the payload enables the requestor to match response messages to request messages. For simplicity, the same version of the Heartbeat Protocol is used with both TLS and DTLS. Thus, the payload is required for both TLS and DTLS.



Since the first introduction of SSL in 1994, and the subsequent standardization of TLS, numerous attacks have been devised against these protocols. The appearance of each attack has necessitated changes in the protocol, the encryption tools used, or some aspects of the implementation of SSL and TLS to counter these threats.

We can group the attacks into four general categories:

• **Attacks on the Handshake Protocol**: As early as 1998, an approach to compromising
the Handshake Protocol based on exploiting the formatting and implementation of the RSA encryption scheme was presented [BLEI98]. As countermeasures were implemented, the attack was refined and adjusted to not only thwart the countermeasures but also speed up the attack [e.g., BARD12].

• **Attacks on the record and application data protocols:** A number of vulnerabilities have been discovered in these protocols, leading to patches to counter the new threats. As a recent example, in 2011, researchers Thai Duong

and Juliano Rizzo demonstrated a proof of concept called BEAST (Browser Exploit Against SSL/TLS) that turned what had been considered only a theoretical vulnerability into a practical attack [GOOD11]. BEAST leverages a type of cryptographic attack called a chosen-plaintext attack. The attacker mounts the attack by choosing a guess for the plaintext that is associated with a known ciphertext. The researchers developed a practical algorithm for launching successful attacks. Subsequent patches were able to thwart this attack. The authors of the BEAST attack are also the creators of the 2012 CRIME (Compression Ratio Info-leak Made Easy) attack, which can allow an attacker to recover the content of web cookies when data compression is used along with TLS [GOOD12b]. When used to recover the content of secret authentication cookies, it allows an attacker to perform session hijacking on an authenticated web session.

• **Attacks on the PKI:** Checking the validity of X.509 certificates is an activity subject to a variety of attacks, both in the context of SSL/TLS and elsewhere. For example, [GEOR12] demonstrated that commonly used libraries for SSL/TLS suffer from vulnerable certificate validation implementations. The authors revealed weaknesses in the source code of OpenSSL, GnuTLS, JSSE, ApacheHttpClient, Weberknecht, cURL, PHP, Python, and applications build upon or with these products.

• **Other attacks:** [MEYE13] lists a number of attacks that do not fit into any of the preceding categories. One example is an attack announced in 2011 by the German hacker group The Hackers Choice, which is a DoS attack [KUMA11]. The attack creates a heavy processing load on a server by overwhelming the target with SSL/TLS handshake requests. Boosting system load is done by establishing
new connections or using renegotiation. Assuming that the majority of computation during a handshake is done by the server the attack creates more system load on the server than on the source device, leading to a DoS. The server is forced to continuously recompute random numbers and keys.

The history of attacks and countermeasures for SSL/TLS is representative of that for other Internet-based protocols. A “perfect” protocol and a “perfect” implementation strategy are never achieved. A constant back-and-forth between threats and countermeasures determines the evolution of Internet-based protocols.

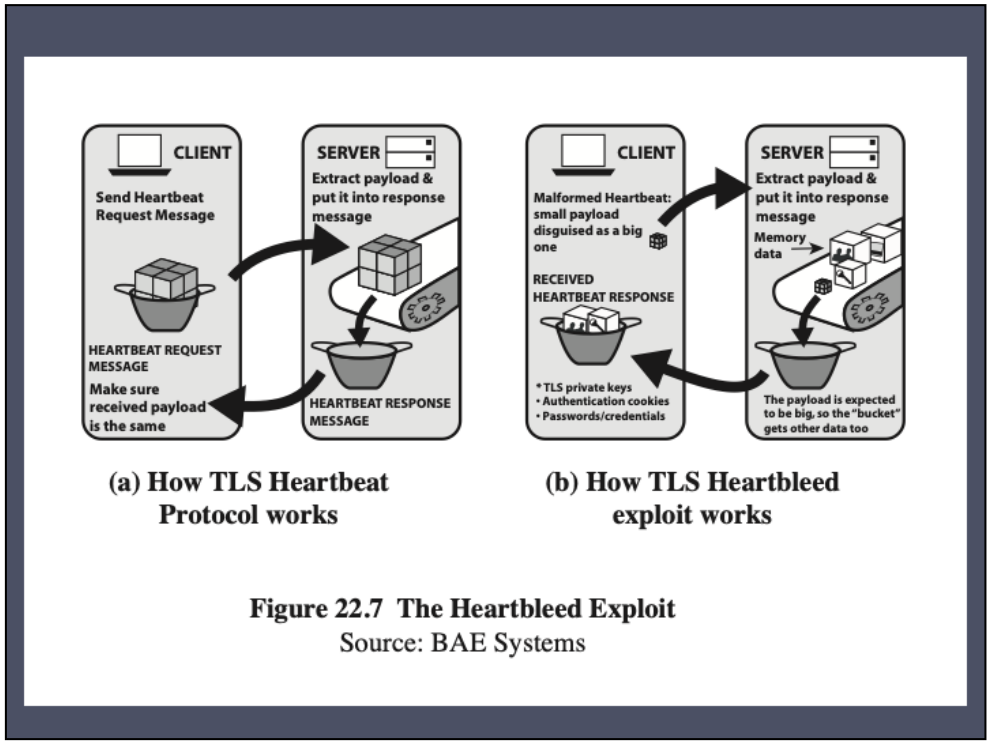


Figure 22.7 The Heartbleed Exploit
Source: BAE Systems

A bug discovered in 2014 in the TLS software created one of the potentially most catastrophic TLS vulnerabilities. The bug was in the open-source OpenSSL implementation of the Heartbeat Protocol. It is important to note that this vulnerability is not a design flaw in the TLS specification; rather it is a programming mistake in the OpenSSL library.

To understand the nature of the vulnerability, recall from our previous discussion that the heartbeat_request message includes payload length, payload and padding fields. Before the bug was fixed, the OpenSSL version of the Heartbeat Protocol worked as follows: The software reads the incoming request message and allocates a buffer large enough to hold the message header, the payload, and the padding. It then overwrites the current contents of the buffer with the incoming message, changes the first byte to indicate the response message type, then transmits a response message, which includes the payload length field and the payload. However, the software does not check the message length of the incoming message. As a result, an adversary can send a message that indicates the maximum payload length (64 KB) but only includes the minimum payload (16 bytes). This means that almost 64 KB of the buffer is not

overwritten and whatever happened to be in memory at the time will be sent to the requestor. Repeated attacks can result in the exposure of significant amounts of memory on the vulnerable system. Figure 22.7 illustrates the intended behavior and the actual behavior for the Heartbleed exploit.

This is a spectacular flaw. The untouched memory could contain private keys, user identification information, authentication data, passwords, or other sensitive data. The flaw was not discovered for several years. Even though eventually the bug was fixed in all implementations, large amounts of sensitive data were exposed to the Internet. Thus, we have a long exposure period, an easily implemented attack, and an attack that leaves no trace. Full recovery from this bug could take years.

Compounding the problem is that OpenSSL is the most widely used TLS implementation. Servers using OpenSSL for TLS include finance, stock trading, personal and corporate email, social networks, banking, online shopping, and government agencies. It has been estimated that over two-thirds of the Internet's Web servers use OpenSSL, giving some idea of the scale of the problem [GOOD14].

HTTPS (HTTP over SSL)

- Combination of HTTP and SSL to implement secure communication between a Web browser and a Web server
- Built into all modern Web browsers
  - Search engines do not support HTTPS
  - URL addresses begin with https://
- Documented in RFC 2818, HTTP Over TLS
- Agent acting as the HTTP client also acts as the TLS client
- 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

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,* 2000) . There is no fundamental change in using HTTP over either SSL or TLS, and both implementations are referred to as HTTPS.

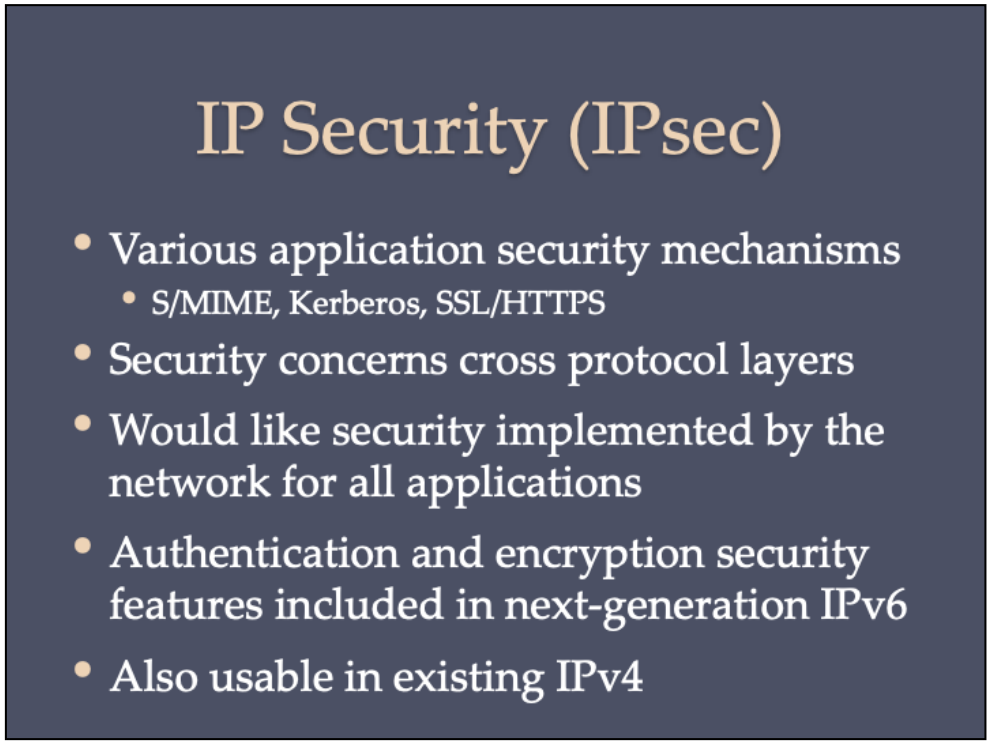
For HTTPS, the agent acting as the HTTP client also acts as the TLS client. The client initiates a connection to the server on the appropriate port and then sends the TLS ClientHello to begin the TLS handshake. When the TLS handshake has finished, the client may then initiate the first HTTP request. 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.

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.

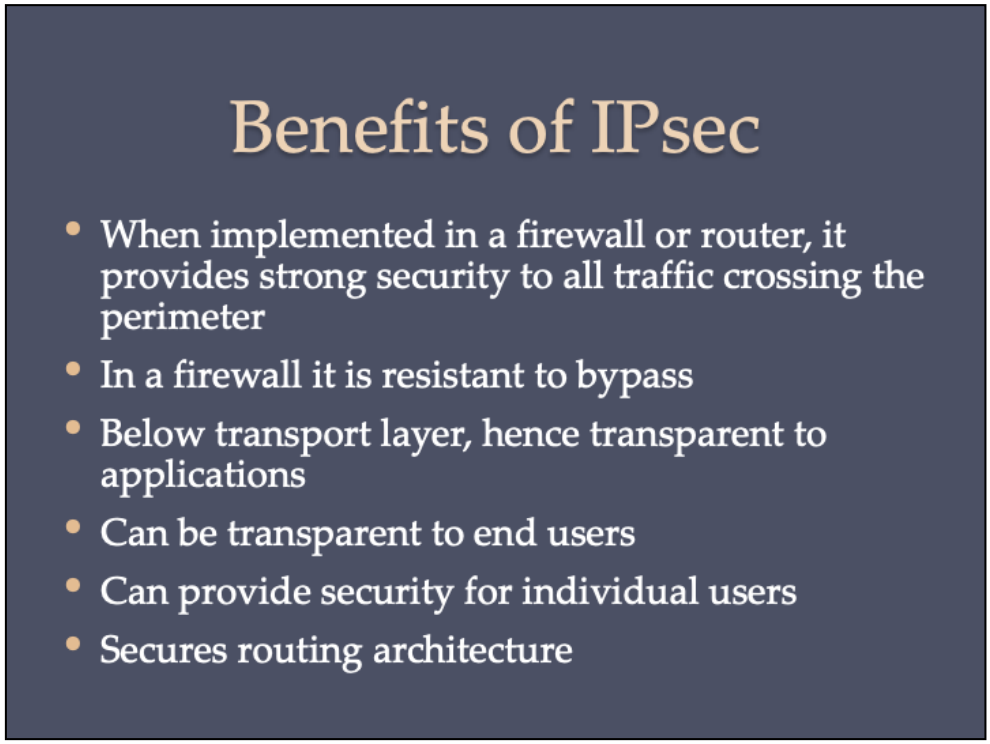
HTTP clients also must be able to cope with a situation in which the underlying TCP connection is terminated without a prior close_notify alert and without a Connection: close indicator. Such a situation could be due to a programming error on the server or a communication error that causes the TCP connection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



The Internet community has developed application-specific security mechanisms in a number of areas, including electronic mail (S/MIME), client/server (Kerberos), Web access (SSL), and others. However, users have some security concerns that cut across protocol layers. For example, an enterprise can run a secure, private TCP/IP network by disallowing links to untrusted sites, encrypting packets that leave the premises, and authenticating packets that enter the premises. By implementing security at the IP level, an organization can ensure secure networking not only for applications that have security mechanisms but also for the many security-ignorant applications.

In response to these issues, the Internet Architecture Board (IAB) included authentication and encryption as necessary security features in the next-generation IP, which has been issued as IPv6. Fortunately, these security capabilities were designed to be usable both with the current IPv4 and the future IPv6. This means that vendors can begin offering these features now, and many vendors do now have some IPsec capability in their products.

IP-level security encompasses three functional areas: authentication, confidentiality, and key management. The authentication mechanism assures that a received packet was, in fact, transmitted by the party identified as the source in the packet header. In addition, this mechanism assures that the packet has not been altered in transit. The confidentiality facility enables communicating nodes to encrypt messages to prevent eavesdropping by third parties. The key management facility is concerned with the secure exchange of keys. The current version of IPsec, known as IPsecv3, encompasses authentication and confidentiality. Key management is provided by the Internet Key Exchange standard, IKEv2.



The benefits of IPsec include the following:

• When IPsec is implemented in a firewall or router, it provides strong security that can be applied to all traffic crossing the perimeter. Traffic within a company or workgroup does not incur the overhead of security-related processing.

• IPsec in a firewall is resistant to bypass if all traffic from the outside must use IP and the firewall is the only means of entrance from the Internet into the organization.

• IPsec is below the transport layer (TCP, UDP) and so is transparent to applications. There is no need to change software on a user or server system when IPsec is implemented in the firewall or router. Even if IPsec is implemented in end systems, upper-layer software, including applications, is not affected.

• IPsec can be transparent to end users. There is no need to train users on security

mechanisms, issue keying material on a per-user basis, or revoke keying material when users leave the organization.

• IPsec can provide security for individual users if needed. This is useful for off-site workers and for setting up a secure virtual subnetwork within an organization for sensitive applications.

In addition to supporting end users and protecting premises systems and networks, IPsec can play a vital role in the routing architecture required for internetworking. [HUIT98] lists the following examples of the use of IPsec. IPsec can assure that

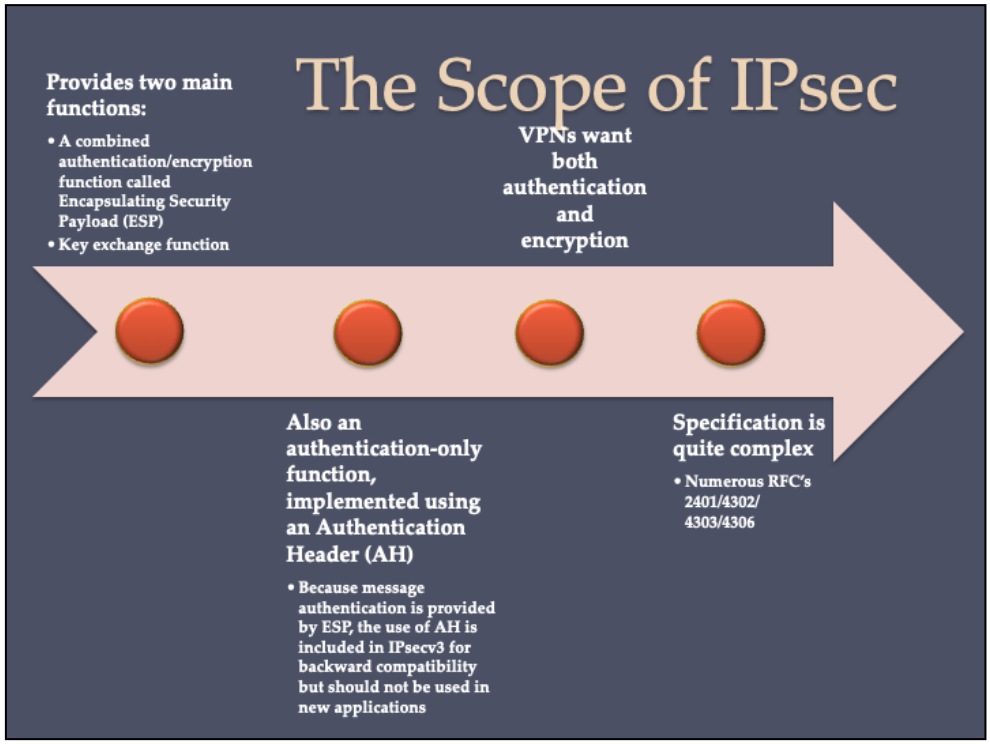
• A router advertisement (a new router advertises its presence) comes from an authorized router.

• A neighbor advertisement (a router seeks to establish or maintain a neighbor relationship with a router in another routing domain) comes from an authorized router.

• A redirect message comes from the router to which the initial packet was sent.

• A routing update is not forged.

Without such security measures, an opponent can disrupt communications or divert some traffic. Routing protocols such as Open Shortest Path First (OSPF) should be run on top of security associations between routers that are defined by IPsec.



IPsec provides two main functions: a combined authentication/encryption function called Encapsulating Security Payload (ESP) and a key exchange function. For virtual private networks, both authentication and encryption are generally desired, because it is important both to (1) assure that unauthorized users do not penetrate the virtual private network and (2) assure that eavesdroppers on the Internet cannot read messages sent over the virtual private network. There is also an authentication-only function, implemented using an Authentication Header (AH). Because message authentication is provided by ESP, the use of AH is deprecated. It is included in IPsecv3 for backward compatibility but should not be used in new applications.

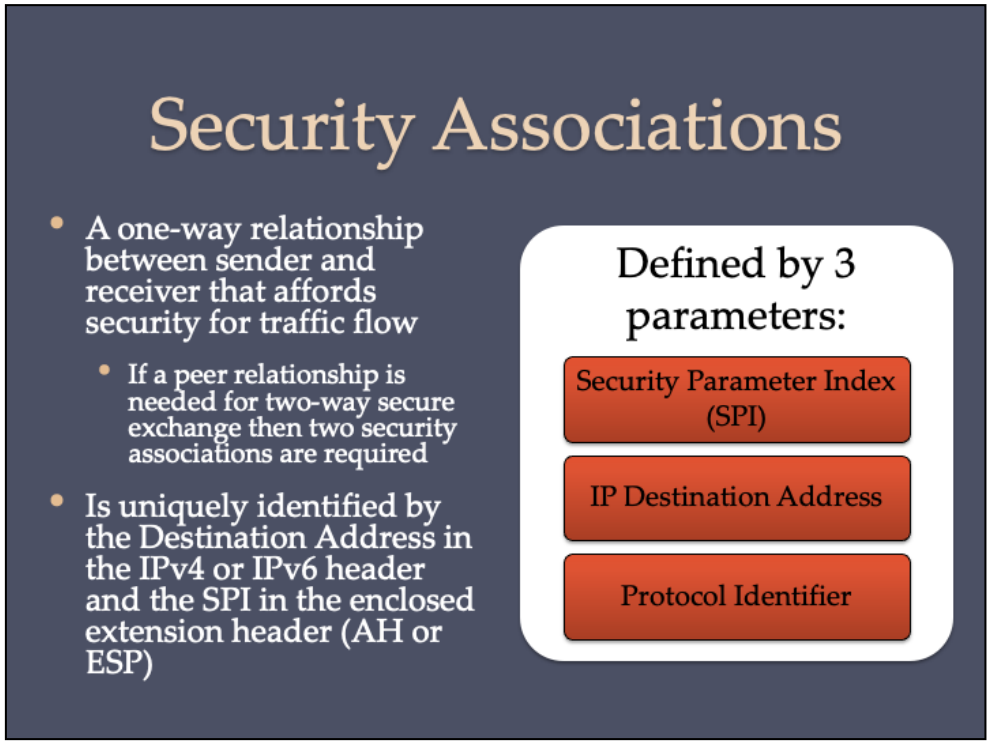
We do not discuss AH in this chapter.

The key exchange function allows for manual exchange of keys as well as an automated scheme.

The IPsec specification is quite complex and covers numerous documents. The most important of these are:

- RFC 2401 (*Security Architecture for the Internet Protocol, 1*998*)*
- RFC 4302 (*IP Authentication Header*, 2005)
- RFC 4303 (*IP Encapsulating Security Payload (ESP)*, 2005)
- RFC 4306 (*Internet Key Exchange (IKEv2) Protocol*, 2005)

In this section, we provide an overview of some of the most important elements of IPsec.



A key concept that appears in both the authentication and confidentiality mechanisms for IP is the security association (SA). An association is a one-way relationship between a sender and a receiver that affords security services to the traffic carried on it. If a peer relationship is needed, for two-way secure exchange, then two security associations are required. Security services are afforded to an SA for the use of ESP.

An SA is uniquely identified by three parameters:

• **Security parameter index (SPI):** A bit string assigned to this SA and having local significance only. The SPI is carried in an ESP header to enable the receiving system to select the SA under which a received packet will be processed.

• **IP destination address:** This is the address of the destination endpoint of the SA, which may be an end-user system or a network system such as a firewall or router.

• **Protocol identifier:** This field in the outer IP header indicates whether the association is an AH or ESP security association.

Hence, in any IP packet, the security association is uniquely identified by the Destination Address in the IPv4 or IPv6 header and the SPI in the enclosed extension header (AH or ESP).

An IPsec implementation includes a security association database that defines the parameters associated with each SA. An SA is characterized by the following parameters:

• **Sequence number counter:** A 32-bit value used to generate the Sequence Number field in AH or ESP headers.

• **Sequence counter overflow:** A flag indicating whether overflow of the sequence number counter should generate an auditable event and prevent further transmission of packets on this SA.

• **Antireplay window:** Used to determine whether an inbound AH or ESP packet is a replay, by defining a sliding window within which the sequence number must fall.

• **AH information**: Authentication algorithm, keys, key lifetimes, and related parameters being used with AH.

• **ESP information:** Encryption and authentication algorithm, keys, initialization values, key lifetimes, and related parameters being used with ESP.

• **Lifetime of this security association:** A time interval or byte count after which an SA must be replaced with a new SA (and new SPI) or terminated, plus an indication of which of these actions should occur.

• **IPsec protocol mode:** Tunnel, transport, or wildcard (required for all implementations).
These modes are discussed later in this section.

• **Path MTU:** Any observed path maximum transmission unit (maximum size of a packet that can be transmitted without fragmentation) and aging variables (required for all implementations).

The key management mechanism that is used to distribute keys is coupled to the authentication and privacy mechanisms only by way of the security parameters index. Hence, authentication and privacy have been specified independent of any specific key management mechanism.

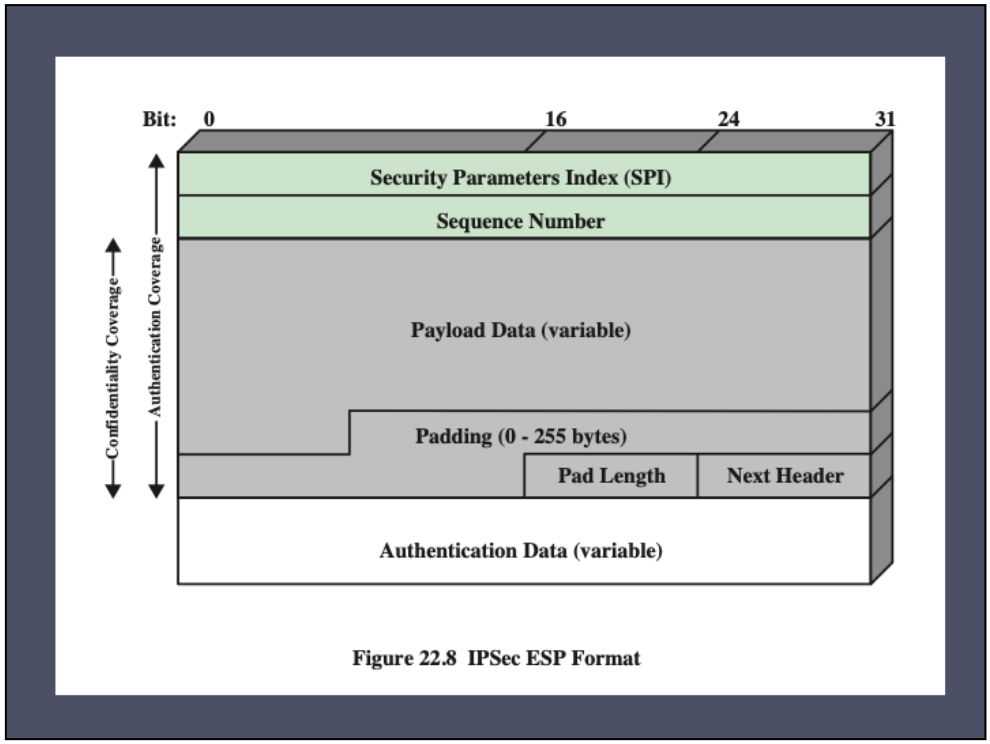


Figure 22.8 IPSec ESP Format

The Encapsulating Security Payload provides confidentiality services, including confidentiality of message contents and limited traffic flow confidentiality. As an optional feature, ESP can also provide an authentication service.

Figure 22.8 shows the format of an ESP packet. It contains the following fields:

• **Security Parameters Index (32 bits):** Identifies a security association.

• **Sequence Number (32 bits):** A monotonically increasing counter value.

• **Payload Data (variable):** This is a transport-level segment (transport mode) or IP packet (tunnel mode) that is protected by encryption.

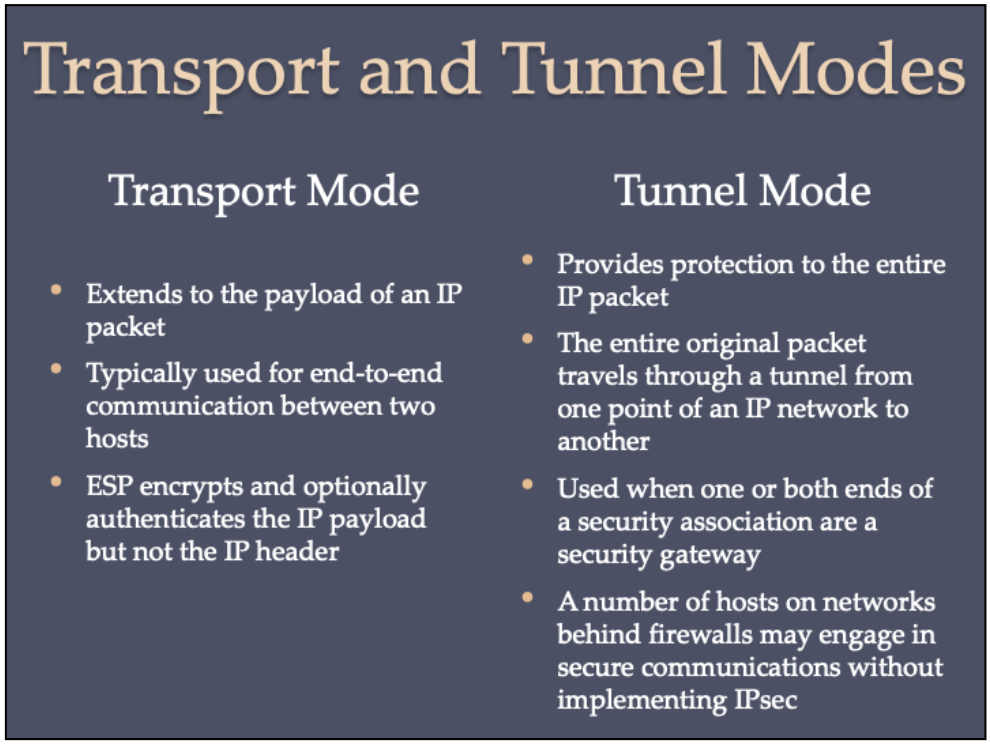
• **Padding (0–255 bytes):** May be required if the encryption algorithm requires the plaintext to be a multiple of some number of octets.

• **Pad Length (8 bits):** Indicates the number of pad bytes immediately preceding

this field.

• **Next Header (8 bits):** Identifies the type of data contained in the Payload Data field by identifying the first header in that payload (e.g., an extension header in IPv6, or an upper-layer protocol such as TCP).

• **Integrity Check Value (variable):** A variable-length field (must be an integral number of 32-bit words) that contains the integrity check value computed over the ESP packet minus the Authentication Data field



ESP supports two modes of use: transport and tunnel modes. We begin this section with a brief overview.

Transport mode provides protection primarily for upper-layer protocols. That is, transport mode protection extends to the payload of an IP packet. Examples include a TCP or UDP segment, both of which operate directly above IP in a host protocol stack. Typically, transport mode is used for end-to-end communication between two hosts (e.g., a client and a server, or two workstations). When a host runs ESP over IPv4, the payload is the data that normally follow the IP header. For IPv6, the payload is the data that normally follow both the IP header and any IPv6 extension headers that are present, with the possible exception of the destination options header, which may be included in the protection.

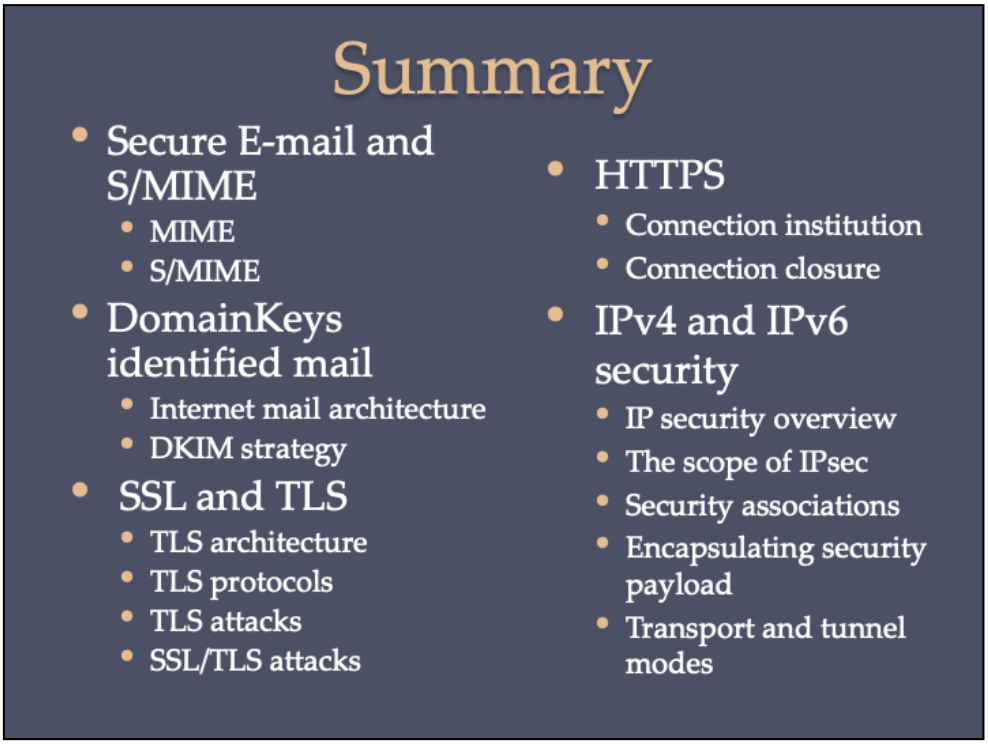
ESP in transport mode encrypts and optionally authenticates the IP payload but not the IP header.

Tunnel mode provides protection to the entire IP packet. To

achieve this, after the ESP fields are added to the IP packet, the entire packet plus security fields are treated as the payload of new outer IP packet with a new outer IP header. The entire original, inner, packet travels through a tunnel from one point of an IP network to another; no routers along the way are able to examine the inner IP header. Because the original packet is encapsulated, the new, larger packet may have totally different source and destination addresses, adding to the security. Tunnel mode is used when one or both ends of a security association are a security gateway, such as a firewall or router that implements IPsec. With tunnel mode, a number of hosts on networks behind firewalls may engage in secure communications without implementing IPsec. The unprotected packets generated by such hosts are tunneled through external networks by tunnel mode SAs set up by the IPsec software in the firewall or secure router at the boundary of the local network.

Here is an example of how tunnel mode IPsec operates. Host A on a network generates an IP packet with the destination address of host B on another network, similar to that shown in Figure 9.3. This packet is routed from the originating host to a firewall or secure router at the boundary of A's network. The firewall filters all outgoing packets to determine the need for IPsec processing. If this packet from A to B requires IPsec, the firewall performs IPsec processing and encapsulates the packet with an outer IP header. The source IP address of this outer IP packet is this firewall, and the destination address may be a firewall that forms the boundary to B's local network. This packet is now routed to B's firewall, with intermediate routers examining only the outer IP header. At B's firewall, the outer IP header is stripped off, and the inner packet is delivered to B.

ESP in tunnel mode encrypts and optionally authenticates the entire inner IP packet, including the inner IP header.



Chapter 22 summary.