Network Security

Web Security: Network and Transport layers

Kamalika Bhattacharjee

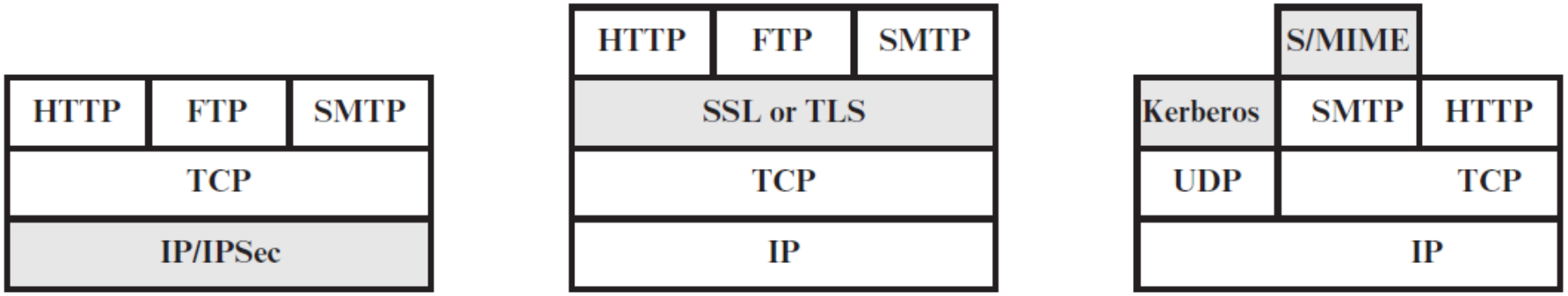
Web Security Considerations

- Need for tailored security tools:
  - Underlying software is extraordinarily complex. This complex software may hide many potential security flaws. The short history of the Web is filled with examples of new and upgraded systems, properly installed, that are vulnerable to a variety of security attacks.
  - A Web server can be exploited as a launching pad into the corporation's or agency's entire computer complex. Once the Web server is subverted, an attacker may be able to gain access to data and systems not part of the Web itself but connected to the server at the local site.
  - Casual and untrained (in security matters) users are common clients for Web based services. Such users are not necessarily aware of the security risks that exist and do not have the tools or knowledge to take effective countermeasures.

Web Security Threats

	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



(a) Network level

(b) Transport level

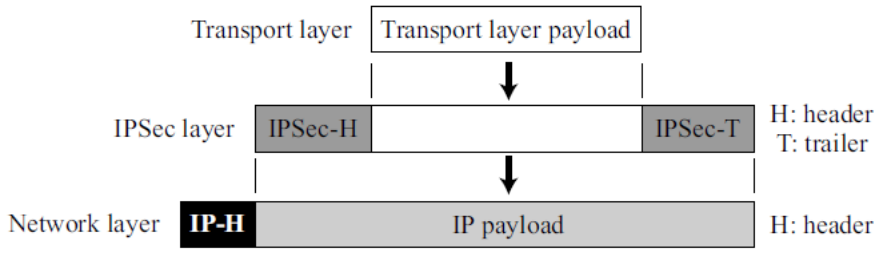
(c) Application level

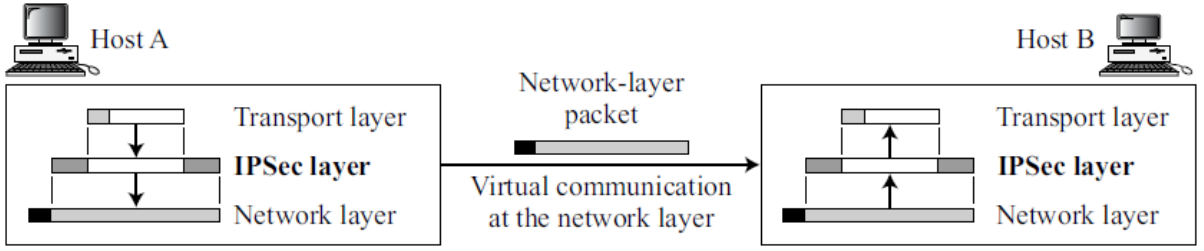
IP Security (IPSec)

- Collection of protocols designed by the Internet Engineering Task Force (IETF) to provide security for a packet at the network level.
- Helps to create authenticated and confidential packets for the IP layer
- Usefulness:
  - Enhance the security of those client/server programs, such as electronic mail, that use their own security protocols.
  - Enhance the security of those client/server programs, such as HTTP, that use the security services provided at the transport layer as well as that do not use the security services provided at the transport layer.
  - Provide security for node-to-node communication programs such as routing protocols.

Modes of IPSec

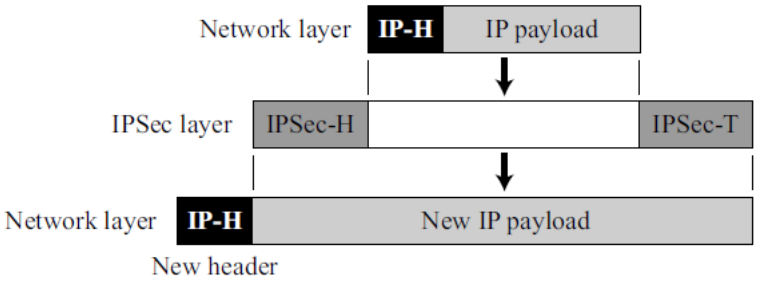
- Transport Mode
  - Here IPSec protects what is delivered from the transport layer to the network layer.
  - Protects the network layer payload, the payload to be encapsulated in the network layer
  - Does not protect the IP header; it only protects the information coming from the transport layer. In this mode, the IPSec header (and trailer) are added to the information coming from the transport layer. The IP header is added later.
  - Normally used for host-to-host (end-to-end) protection of data.
  - The sending host uses IPSec to authenticate and/or encrypt the payload delivered from the transport layer. The receiving host uses IPSec to check the authentication and/or decrypt the IP packet and deliver it to the transport layer.

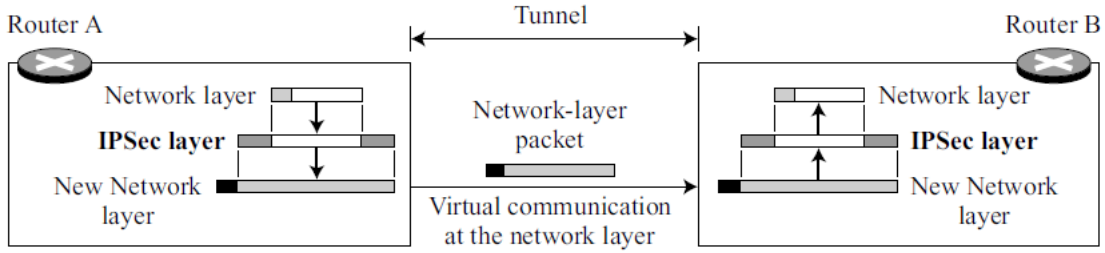




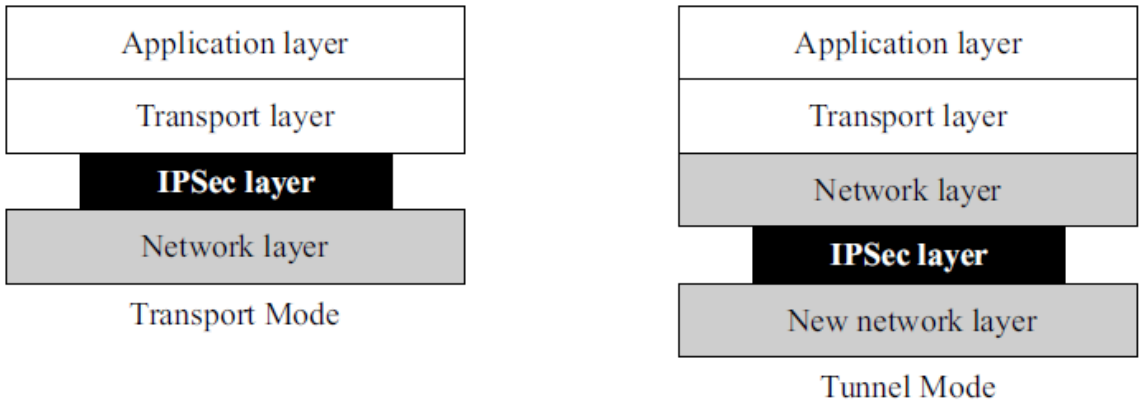
Modes of IPSec

- Tunnel Mode
  - Here, IPSec protects the entire IP packet. It takes an IP packet, including the header, applies IPSec security methods to the entire packet, and then adds a new IP header
  - The new IP header has different information than the original IP header.
  - Normally used when either the sender or the receiver is not a host; example: between two routers, between a host and a router, or between a router and a host.
  - Entire packet is protected from intrusion between the sender and the receiver, as if the whole packet goes through an imaginary tunnel





Modes of IPSec



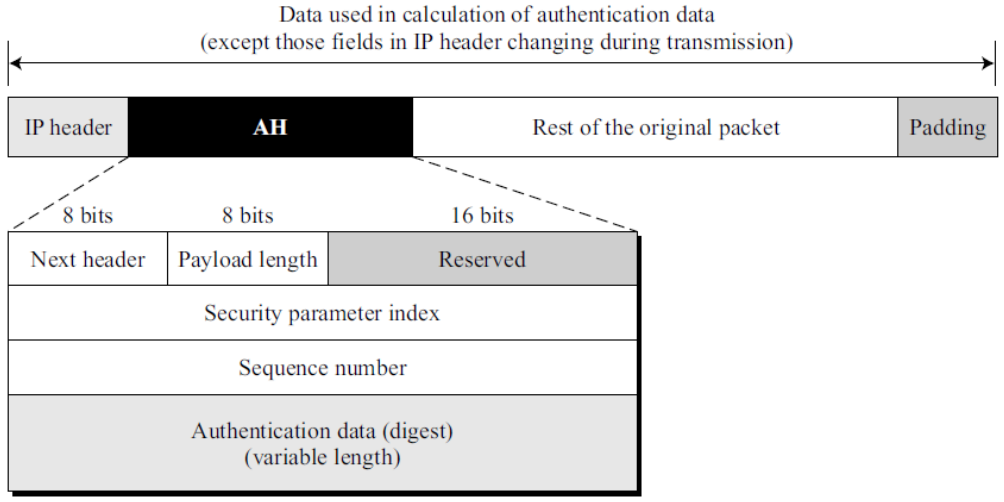
Transport mode versus tunnel mode

Authentication Header (AH) Protocol

- Designed to authenticate the source host and to ensure the integrity of the payload carried in the IP packet.
- Uses a hash function and a symmetric key to create a message digest; the digest is inserted in the authentication header (AH). The AH is then placed in the appropriate location, based on the mode (transport or tunnel).
- Steps:
  - An authentication header is added to the payload with the authentication data field set to 0.
  - Padding may be added to make the total length even for a particular hashing algorithm.
  - Hashing is based on the total packet. However, only those fields of the IP header that do not change during transmission are included in the calculation of the message digest (authentication data).
  - The authentication data are inserted in the authentication header.
  - The IP header is added after changing the value of the protocol field to 51 to show that the packet carries an authentication header.
  - The next header field inside authentication header holds the original value of the protocol field

Authentication Header (AH) Protocol

- **Next header:** defines the type of payload carried by the IP datagram (such as TCP, UDP, ICMP, or OSPF). The process copies the value of the protocol field in the IP datagram to this field.
- **Payload length:** defines the length of the authentication header in 4-byte multiples excluding the first 8 bytes.
- **Security parameter index:** plays the role of a virtual circuit identifier and is the same for all packets sent during a connection called a *Security Association*
- **Authentication data:** result of applying a hash function to the entire IP datagram except for the fields that are changed during transit (e.g., time-to-live).



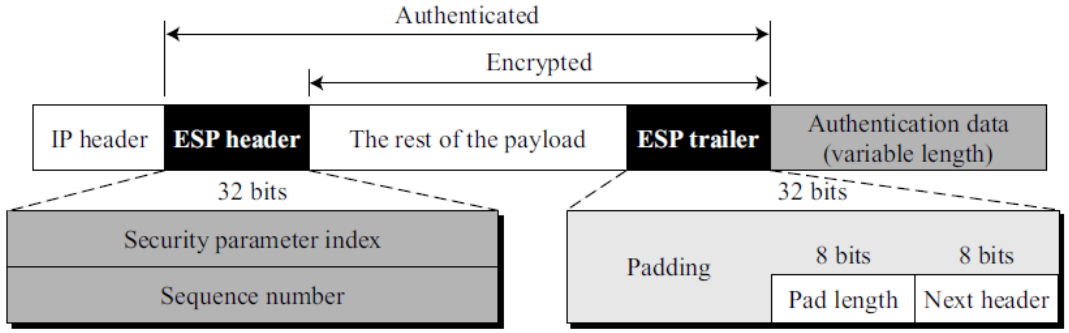
- **Sequence number:** provides ordering information for a sequence of datagrams. The sequence numbers prevent a playback. The sequence number is not repeated even if a packet is retransmitted and does not wrap around after it reaches 2^{32}; a new connection must be established.

Encapsulating Security Payload (ESP)

- AH protocol does not provide privacy, only source authentication and data integrity.
- Encapsulating Security Payload (ESP) provides source authentication, integrity, and privacy.
- ESP adds a header and trailer. Authentication data are added at the end of the packet, which makes its calculation easier.
- Steps:
  - An ESP trailer is added to the payload.
  - The payload and the trailer are encrypted.
  - The ESP header is added.
  - The ESP header, payload, and ESP trailer are used to create the authentication data.
  - The authentication data are added to the end of the ESP trailer.
  - The IP header is added after changing the protocol value to 50.

Encapsulating Security Payload (ESP)

- **Security parameter index and Sequence number:** Similar to that defined for the AH protocol.
- **Padding:** This variable-length field (0 to 255 bytes) of 0s serves as padding.
- **Pad length:** The 8-bit pad-length field defines the number of padding bytes. The value is between 0 and 255; the maximum value is rare.
- **Next header:** Similar to that defined in the AH protocol and serves the same purpose as the protocol field in the IP header before encapsulation.
- **Authentication data:** Result of applying an authentication scheme to parts of the datagram.



The difference between authentication data in AH and ESP is: in AH, part of the IP header is included in the calculation of the authentication data; in ESP, it is not.

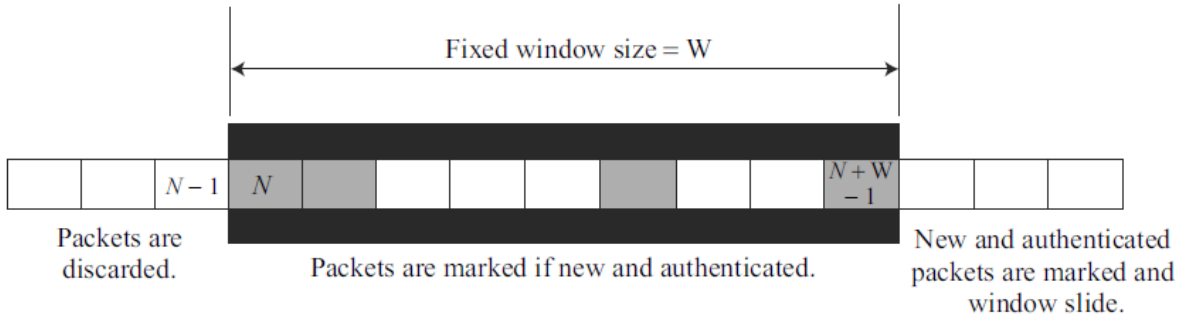
IPSec supports both IPv4 and IPv6. In IPv6, however, AH and ESP are part of the extension header

Services Provided by IPSec

- **Access Control:** IPSec provides access control indirectly using a Security Association Database (SAD). If a packet arrives at a destination with no Security Association already established for this packet, the packet is discarded.
- **Message Integrity:** Message integrity is preserved in both AH and ESP. A digest of data is created and sent by the sender to be checked by the receiver.
- **Entity Authentication:** The Security Association and the keyed-hash digest of the data sent by the sender authenticate the sender of the data in both AH and ESP.
- **Confidentiality:** The encryption of the message in ESP provides confidentiality. AH, however, does not provide confidentiality.
- **Replay Attack Protection:** In both protocols, the replay attack is prevented by using sequence numbers and a sliding receiver window.

Replay Attack Protection

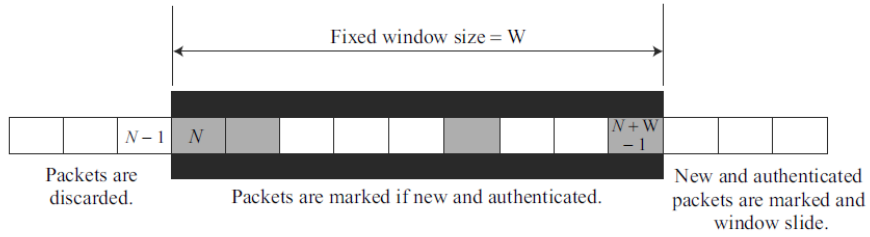
- Each IPSec header contains a unique sequence number when the Security Association is established.
- The number starts from 0 and increases until $2^{32} - 1$.
- When the sequence number reaches the maximum, it is reset to 0 and, at the same time, the old Security Association is deleted and a new one is established.
- To prevent processing duplicate packets, IPSec mandates the use of a fixed-size window at the receiver.
- The size of the window is determined by the receiver with a default value of 64.



The shaded packets signify received packets that have been checked and authenticated

Replay Attack Protection

- **Case 1:** The sequence number of the packet is less than N. This puts the packet to the left of the window. In this case, the packet is discarded. It is either a duplicate or its arrival time has expired.
- **Case 2:** The sequence number of the packet is between N and (N + W − 1), inclusive. This puts the packet inside the window. In this case, if the packet is new (not marked) and it passes the authentication test, the sequence number is marked and the packet is accepted. Otherwise, it is discarded.
- **Case 3:** The sequence number of the packet is greater than (N + W − 1). This puts the packet to the right of the window. In this case, if the packet is authenticated, the corresponding sequence number is marked and the window slides to the right to cover the newly marked sequence number. Otherwise, the packet is discarded.
- If a packet arrives with a sequence number much larger than (N + W), then the sliding of the window may cause many unmarked numbers to fall to the left of the window. These packets, when they arrive, will never be accepted; their time has expired.

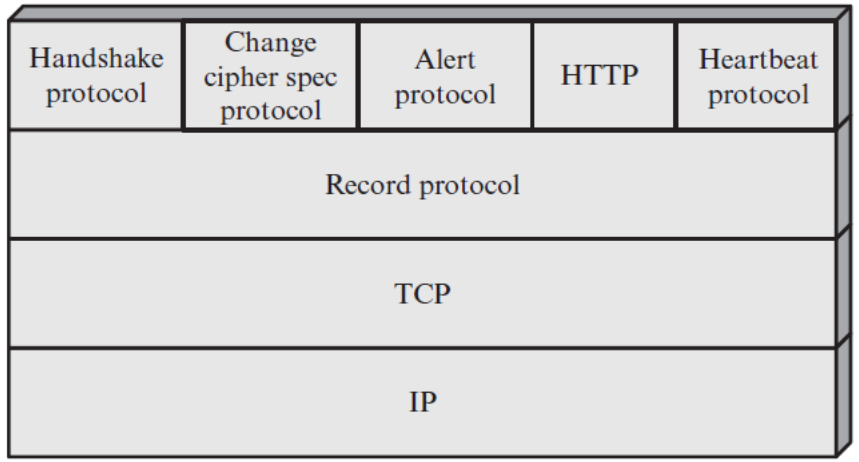


Transport Layer Security (TLS)

- An Internet standard that evolved from a commercial protocol known as **Secure Sockets Layer (SSL)**.
- Current version is Version 1.3
- Although SSL implementations are still around, it has been deprecated by IETF and is disabled by most corporations offering TLS software.
- TLS is a general purpose service implemented as two layers of protocols that rely on TCP.
- 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 TLS, and most Web servers have implemented the protocol.

Transport Layer Security (TLS)

- Designed to make use of TCP to provide a reliable end-to-end secure service
- The TLS 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 and used in the management of TLS exchanges
  - Handshake Protocol
  - Change Cipher Spec Protocol
  - Alert Protocol.
- Heartbeat Protocol is defined in a separate RFC



TLS Protocol Stack

Transport Layer Security (TLS)

- **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.
  - Used to avoid the expensive negotiation of new security parameters for each connection.

Transport Layer Security (TLS)

A session state is defined by the following parameters:

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

Transport Layer Security (TLS)

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 symmetric key used in MAC operations on data sent by the client.
- **Server write key:** The symmetric 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.

Transport Layer Security (TLS)

A connection state is defined by the following parameters:

- **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 **TLS Handshake Protocol**. Thereafter, the final ciphertext block from each record is preserved for use as the IV with the following record.
- **Sequence numbers:** Each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a "**change cipher spec message**," the appropriate sequence number is set to zero. Sequence numbers may not exceed 2^{64}- 1.

Cipher block chaining (CBC) is a mode of operation for a block cipher -- one in which a sequence of bits are encrypted as a single unit, or block, with a cipher key applied to the entire block. It uses an initialization vector (IV) of a certain length.

Transport Layer Security (TLS)

- 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*.

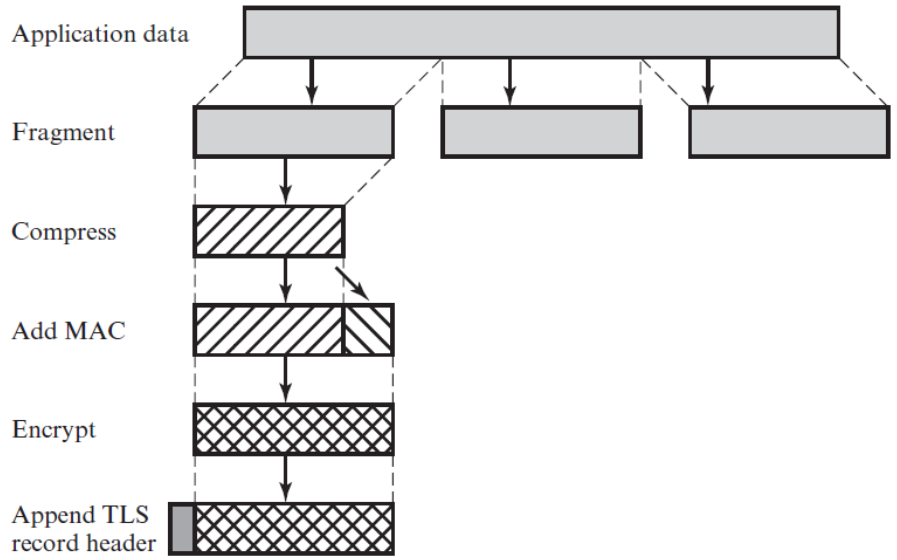
TLS Record Protocol

- Each upper-layer message is fragmented into blocks of 16,384 bytes or less.
- Compression must be lossless and may not increase the content length by more than 1024 bytes. In TLSv2, no compression algorithm is specified, so the default compression algorithm is null.
- The next step in processing is to compute a message authentication code over the compressed data. TLS makes use of the HMAC algorithm:

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

- 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 bytes

TLS Record Protocol Operation



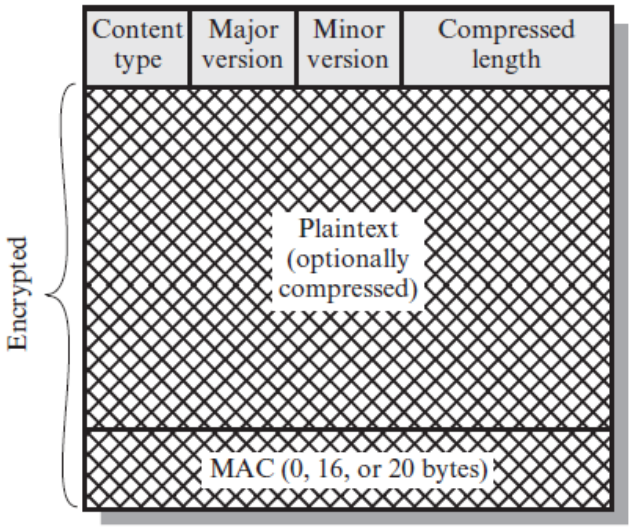
Block Cipher		Stream Cipher	
Algorithm	Key Size	Algorithm	Key Size
AES 3DES	128, 256 168	RC4-128	128

TLS Record Protocol

The final step of TLS Record Protocol processing is to prepend 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 TLS in use. For TLSv2, the value is 3.
- **Minor Version (8 bits):** Indicates minor version in use. For TLSv2, the value is 1.
- **Compressed Length (16 bits):** The length in bytes of the plaintext fragment (or compressed fragment if compression is used).

TLS Record Format

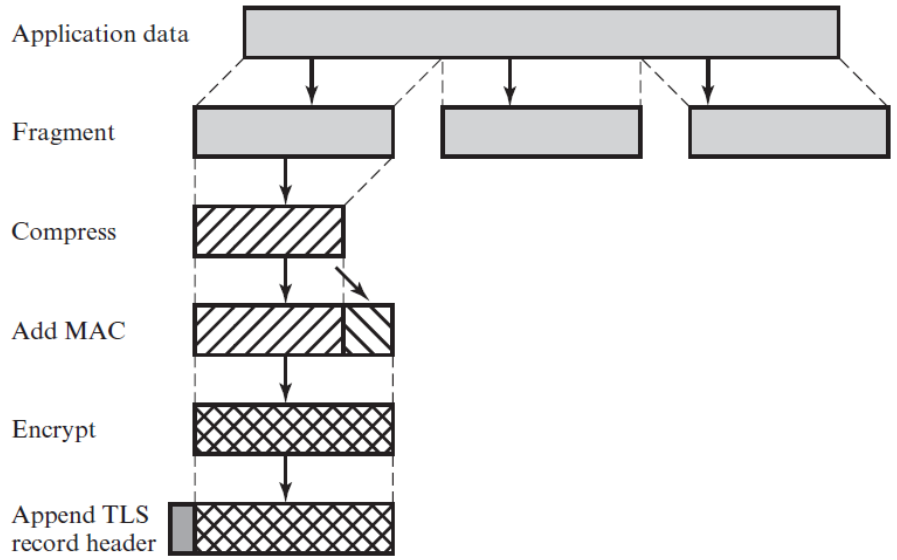


TLS Record Protocol

Provides two services for TLS connections:

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

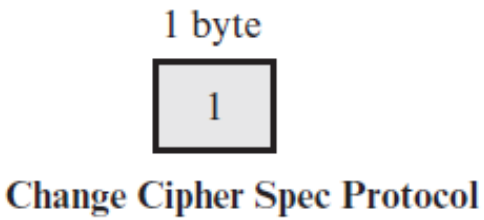
TLS Record Protocol Operation



- 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.

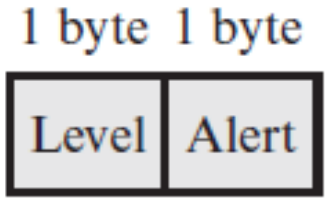
Change Cipher Spec Protocol

- One of the four TLS-specific protocols that use the TLS Record Protocol, and it is the simplest.
- 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.



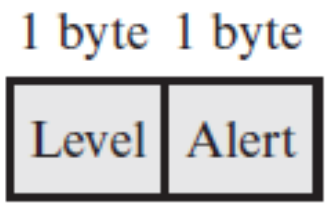
Alert Protocol

- 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.



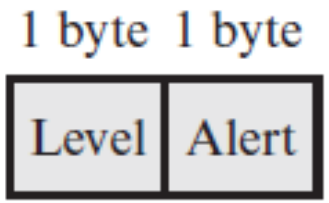
Alert Protocol

- **Fatal Alerts**
  - **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.
  - **decryption_failed:** A ciphertext decrypted in an invalid way; either it was not an even multiple of the block length or its padding values, when checked, were incorrect.
  - **record_overflow:** A TLS record was received with a payload (ciphertext) whose length exceeds $2^{14} + 2048$ bytes, or the ciphertext decrypted to a length of greater than $2^{14} + 1024$ bytes.



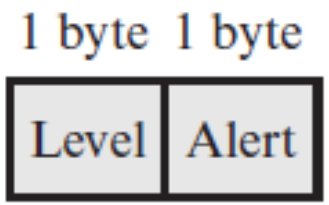
Alert Protocol

- **Fatal Alerts**
  - **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.
  - **export_restriction:** A negotiation not in compliance with export restrictions on key length was detected.
  - **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.
  - **internal_error:** An internal error unrelated to the peer or the correctness of the protocol makes it impossible to continue.



Alert Protocol

- **Warning**
  - **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.
  - **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.
  - **decrypt_error:** A handshake cryptographic operation failed, including being unable to verify a signature, decrypt a key exchange, or validate a finished message.
  - **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.

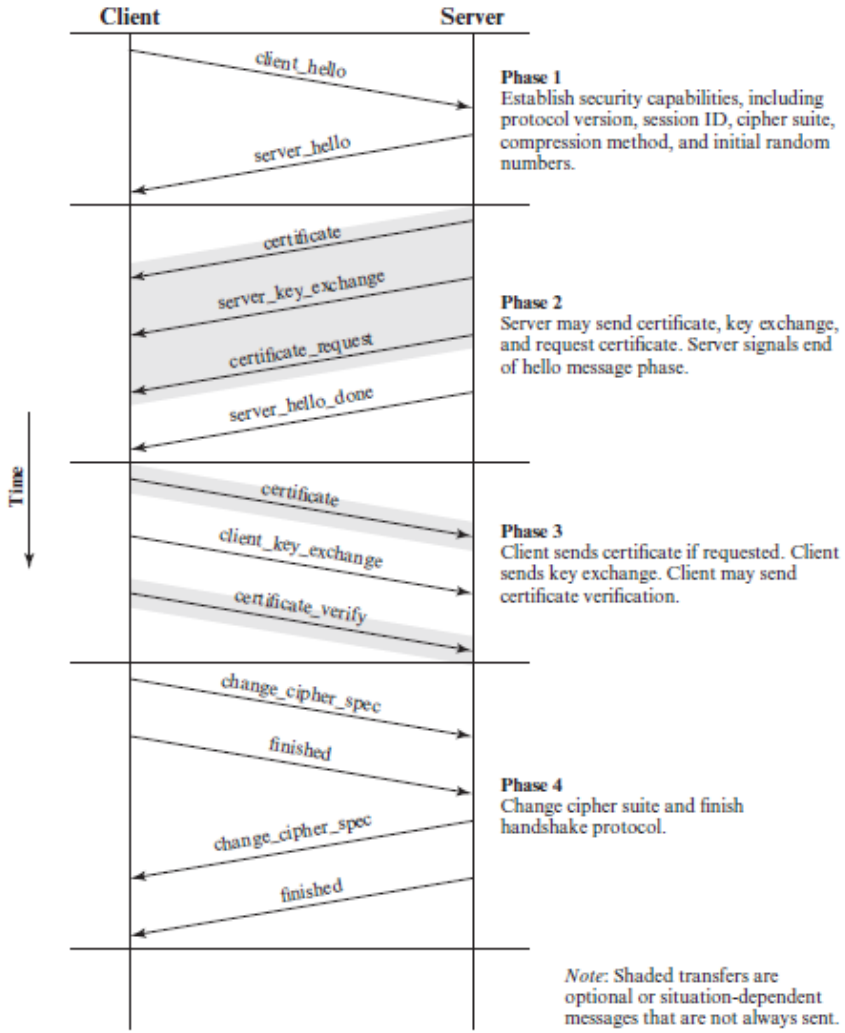


Handshake Protocol

- The most complex part of TLS
- 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 a TLS record.
- used before any application data is transmitted.
- consists of a series of messages exchanged by client and server.
- Each message has three fields:
  - Type (1 byte): Indicates one of 10 messages.
  - Length (3 bytes): The length of the message in bytes.
  - Content (>= 0 bytes): The parameters associated with this message

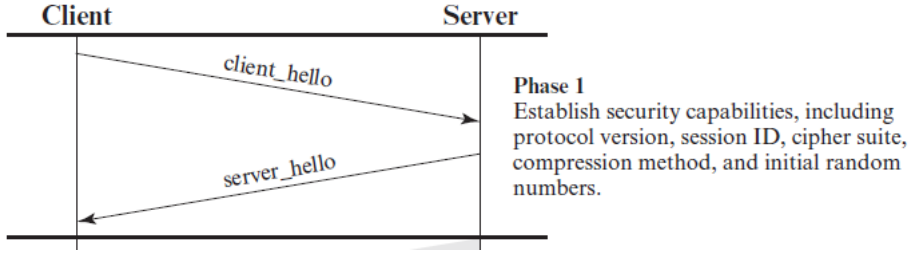
Message Type	Parameters
hello_request	null
client_hello	version, random, session id, cipher suite, compression method
server_hello	version, random, session id, cipher suite, compression method
certificate	chain of X.509v3 certificates
server_key_exchange	parameters, signature
certificate_request	type,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

Handshake Protocol Action



Handshake Protocol Action

Phase 1. Establish security capabilities



- Phase 1 initiates a logical connection and establishes 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:

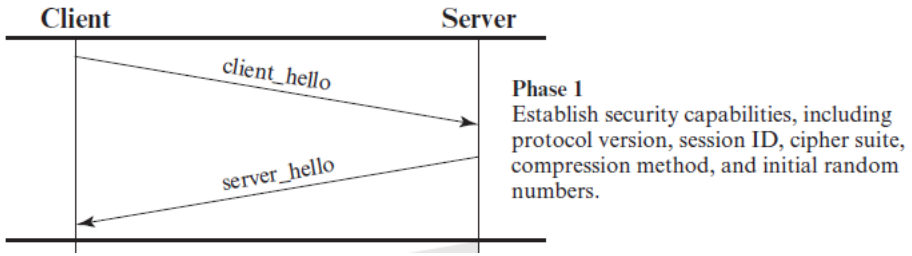
  - **Version:** The highest TLS version understood by the client.

  - **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.

Handshake Protocol Action

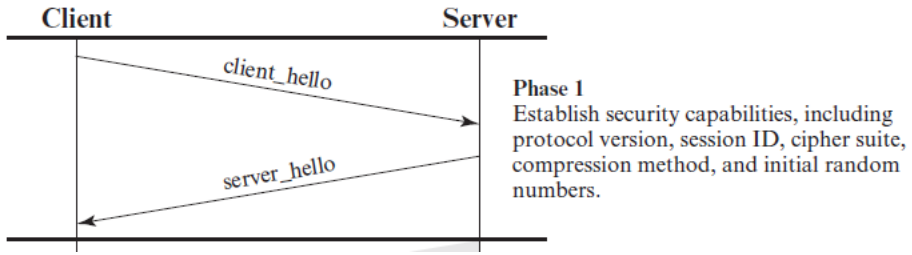
Phase 1. Establish security capabilities



- Phase 1 initiates a logical connection and establishes 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:
  - **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.

Handshake Protocol Action

Phase 1. Establish security capabilities

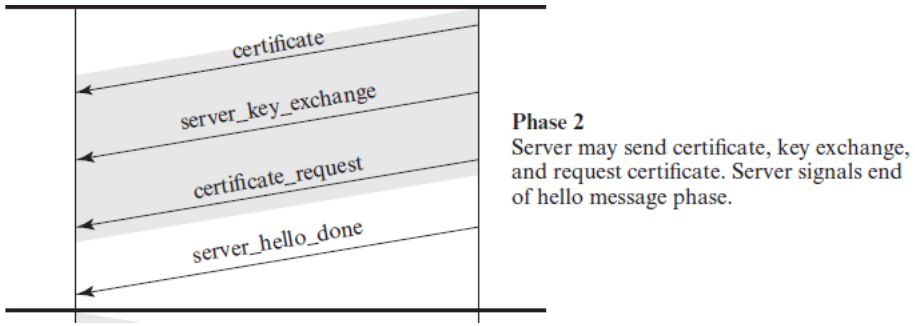


For the server_hello message, the following conventions apply.

- The **Version field** contains the lowest of the version suggested by the client and the highest supported by the server.
- The **Random field** is generated by the server and is independent of the client's Random field.
- If the **SessionID field** of the client was nonzero, the same value is used by the server; otherwise the server's SessionID field contains the value for a new session.
- The **CipherSuite field** contains the single cipher suite selected by the server from those proposed by the client. The first element of the Ciphersuite parameter is the key exchange method for conventional encryption and MAC. The supported methods are **RSA, Fixed Diffie–Hellman, Ephemeral Diffie-Hellman, Anonymous Diffie–Hellman**
- The **Compression field** contains the compression method selected by the server from those proposed by the client.

Handshake Protocol Action

Phase 2. Server authentication and key exchange

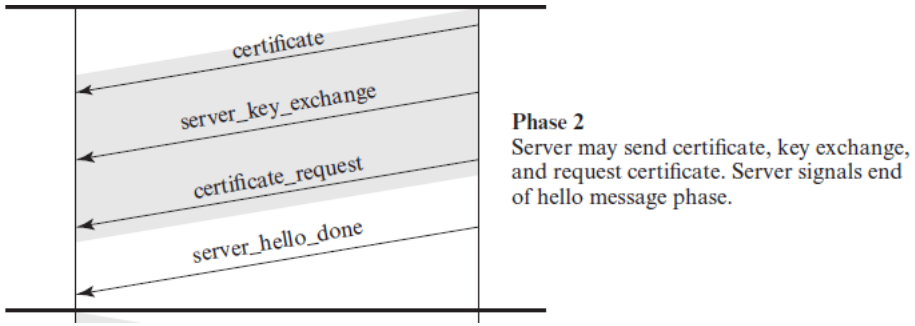


Phase 2
Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase.

- The server begins this phase by sending its certificate if it needs to be authenticated
- The message contains one or a chain of X.509 certificates. The **certificate message** is required for any agreed-on key exchange method except anonymous Diffie–Hellman.
- If fixed Diffie–Hellman is used, this certificate message functions as the server's key exchange message because it contains the server's public Diffie–Hellman parameters.
- Next, a **server_key_exchange message** may be sent if it is required.
  - It is not required in two instances: (1) The server has sent a certificate with fixed Diffie–Hellman parameters; or (2) RSA key exchange is to be used.
  - The server_key_exchange message is needed for: **Anonymous Diffie–Hellman, Ephemeral Diffie–Hellman, RSA key exchange** (in which the server is using RSA but has a signature-only RSA key)

Handshake Protocol Action

Phase 2. Server authentication and key exchange

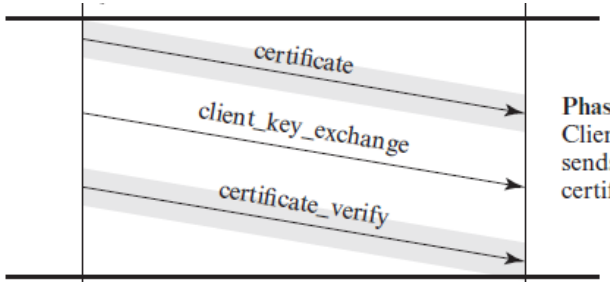


Phase 2
Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase.

- Next, a nonanonymous server (server not using anonymous Diffie–Hellman) can request a certificate from the client.
- The certificate_request message includes two parameters: certificate_type and certificate_authorities.
- The certificate type indicates the public-key algorithm and its use:
  - ■ RSA, signature only, ■ DSS, signature only, ■ RSA for fixed Diffie–Hellman; in this case the signature is used only for authentication, by sending a certificate signed with RSA, ■ DSS for fixed Diffie–Hellman; again, used only for authentication
- The second parameter in the certificate_request message is a list of the distinguished names of acceptable certificate authorities.
- The final message in phase 2, and one that is always required, is the **server_done message**, which is sent by the server to indicate the end of the server hello and associated messages. After sending this message, the server will wait for a client response. This message has no parameters

Handshake Protocol Action

Phase 3. Client authentication and key exchange

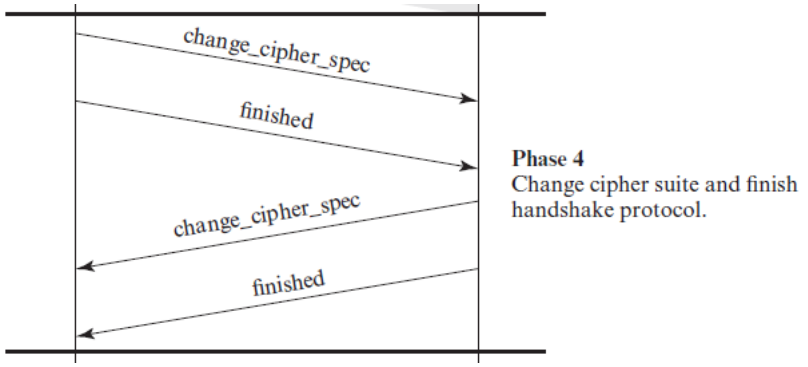


Phase 3
Client sends certificate if requested. Client sends key exchange. Client may send certificate verification.

- Upon receipt of the server_done message, the client should verify that the server provided a valid certificate (if required) and check that the server_hello parameters are acceptable.
- If all is satisfactory, the client sends one or more messages back to the server.
- If the server has requested a certificate, the client begins this phase by sending a certificate message. If no suitable certificate is available, the client sends a **no_certificate alert** instead.
- Next is the **client_key_exchange** message, which must be sent in this phase. The content of the message depends on the type of key exchange.
- 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).

Handshake Protocol Action

Phase 4. Finish



- 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.
  - 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.

Heartbeat Protocol

- With respect to 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 TLS, a Heartbeat protocol was defined in 2012 inRFC 6250 (Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension).
- Heartbeat protocol runs on 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. 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.

Heartbeat Protocol

- 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 random content.
- The padding enables the sender to perform a path MTU (maximum transfer unit) 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.

Heartbeat Protocol

- 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 Datagram Transport Layer Security (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.

SSL/TLS ATTACKS

- **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.
  - As countermeasures were implemented the attack was refined and adjusted to not only thwart the countermeasures but also speed up the attack.
- **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, commonly used libraries for SSL/TLS suffer from vulnerable certificate validation implementations.
  - Attack can be done by exploiting weaknesses in the source code of OpenSSL, GnuTLS, JSSE, ApacheHttpClient, Weberknecht, cURL, PHP, Python and applications built upon or with these products.

SSL/TLS ATTACKS

- **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.
  - 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 [2011].
    - 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.
  - Another attack, **CRIME** (Compression Ratio Info-leak Made Easy) attack [2012], can allow an attacker to recover the content of web cookies when data compression is used along with TLS.
    - When used to recover the content of secret authentication cookies, it allows an attacker to perform session hijacking on an authenticated web session.

SSL/TLS ATTACKS

- **Other attacks**
  - One example is an attack announced in 2011 by the German hacker group The Hackers Choice, which is a DoS attack.
  - 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.

HTTPS

- HTTPS (HTTP over SSL) refers to the combination of HTTP and SSL to implement secure communication between a Web browser and a Web server.
- The HTTPS capability is built into all modern Web browsers. Its use depends on the Web server supporting HTTPS communication.
- 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*. Implementations over both SSL/TLS are referred to as HTTPS.

Connection Initiation

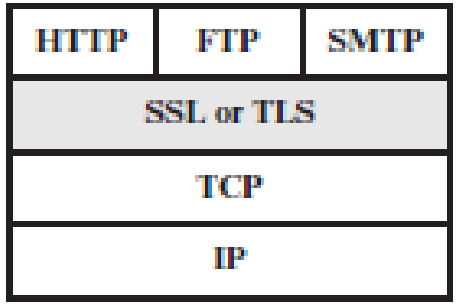
- For HTTPS, the agent acting as the HTTP client also acts as the TLS client.
- The client initiates a connection to the server on the appropriate port and then sends the TLS **ClientHello** to begin the TLS handshake.
- When the TLS handshake has finished, the client may then initiate the first HTTP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

Connection Closure

- An HTTP client or server can indicate the closing of a connection by including the following line in an HTTP record: **Connection: close**.
  - This indicates that the connection will be closed after this record is delivered.
- The closure of an HTTPS connection requires that TLS close the connection with the peer TLS entity on the remote side, which will involve closing the underlying TCP connection.
- At the TLS level, the proper way to close a connection is for each side to use the TLS alert protocol to send a **close_notify alert**.
- TLS implementations must initiate an exchange of closure alerts before closing a connection.
- A TLS implementation may, after sending a closure alert, close the connection without waiting for the peer to send its closure alert, generating an "**incomplete close**".
  - 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.

Connection Closure

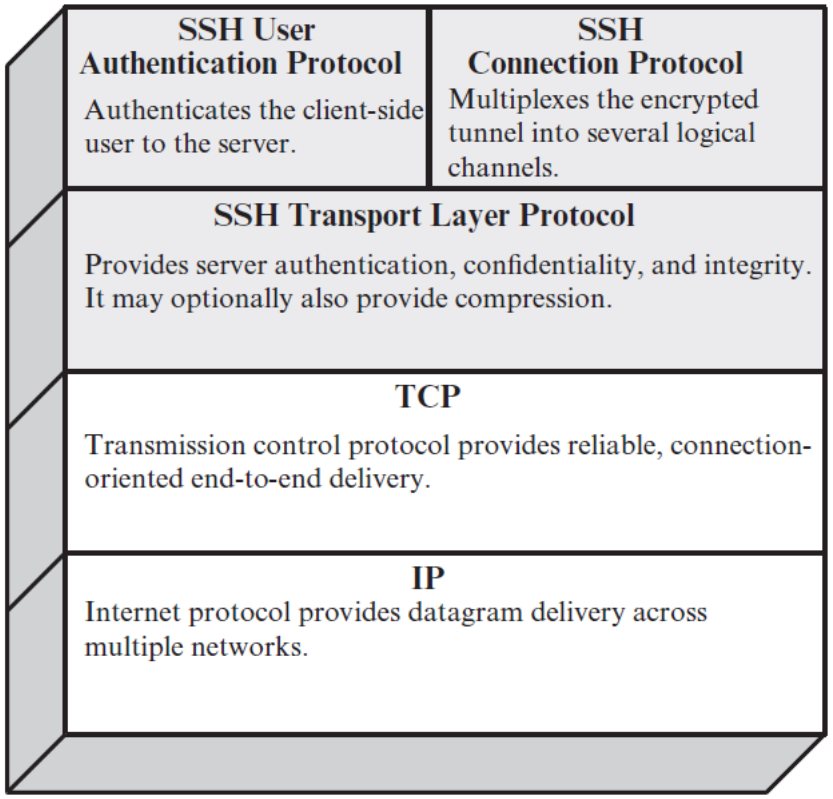
- HTTP clients also must be able to cope with a situation in which the underlying TCP connection is terminated without a prior **close_notify alert** and without a **Connection: close** indicator.
- Such a situation could be due to a programming error on the server or a communication error that causes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



Secure Shell (SSH)

- A protocol for secure network communications designed to be relatively simple and inexpensive to implement.
- The initial version, SSH1 was focused on providing a **secure remote logon** facility to replace TELNET and other remote logon schemes that provided no security.
- SSH also provides a more general client/server capability and can be used for such network functions as file transfer and email.
- A new version, SSH2, fixes a number of security flaws in the original scheme. SSH2 is documented as a proposed standard in IETF RFCs 4250 through 4256.
- SSH client and server applications are widely available for most operating systems.
- It has become the method of choice for **remote login** and **X tunneling** and is rapidly becoming one of the most pervasive applications for encryption technology outside of embedded systems
- SSH is organized as three protocols that typically run on top of TCP

Secure Shell (SSH)



SSH Protocol Stack

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

Transport Layer Protocol

HOST KEYS :

- Server authentication occurs at the transport layer, based on the server possessing a public/private key pair.
- A server may have multiple host keys using multiple different asymmetric encryption algorithms.
- Multiple hosts may share the same host key.
- The server host key is used during key exchange to authenticate the identity of the host. For this to be possible, the client must have a priori knowledge of the server's public host key.
- RFC 4251 dictates two alternative trust models that can be used:
    1. The client has a local database that associates each host name (as typed by the user) with the corresponding public host key.
        - This method requires no centrally administered infrastructure and no third-party coordination. The downside is that the database of name-to-key associations may become burdensome to maintain.

Transport Layer Protocol

HOST KEYS :

- Server authentication occurs at the transport layer, based on the server possessing a public/private key pair.
- A server may have multiple host keys using multiple different asymmetric encryption algorithms.
- Multiple hosts may share the same host key.
- The server host key is used during key exchange to authenticate the identity of the host. For this to be possible, the client must have a priori knowledge of the server's public host key.
- RFC 4251 dictates two alternative trust models that can be used:
    2. The host name-to-key association is certified by a trusted certification authority (CA). The client only knows the CA root key and can verify the validity of all host keys certified by accepted CAs.
        - This alternative eases the maintenance problem, since ideally, only a single CA key needs to be securely stored on the client. On the other hand, each host key must be appropriately certified by a central authority before authorization is possible.

Transport Layer Protocol

PACKET EXCHANGE:

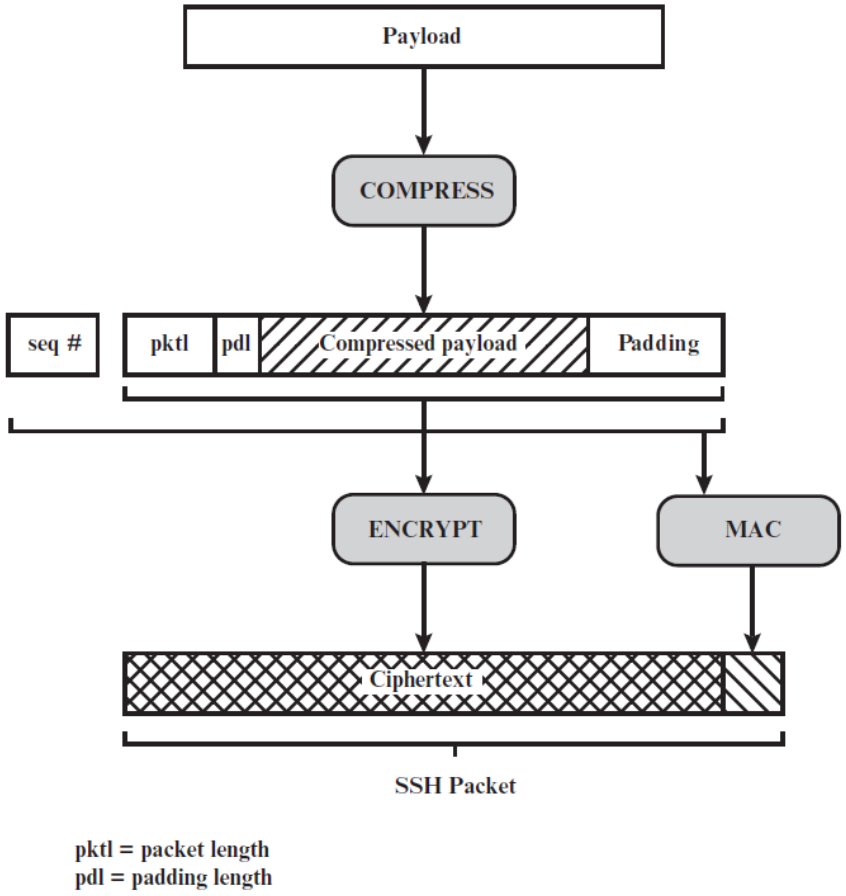
Each packet is in the following format

- **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 the 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. This 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. The sequence number is not included in the packet sent over the TCP connection.

Transport Layer Protocol

PACKET EXCHANGE:

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



SSH Transport Layer Protocol Packet Formation

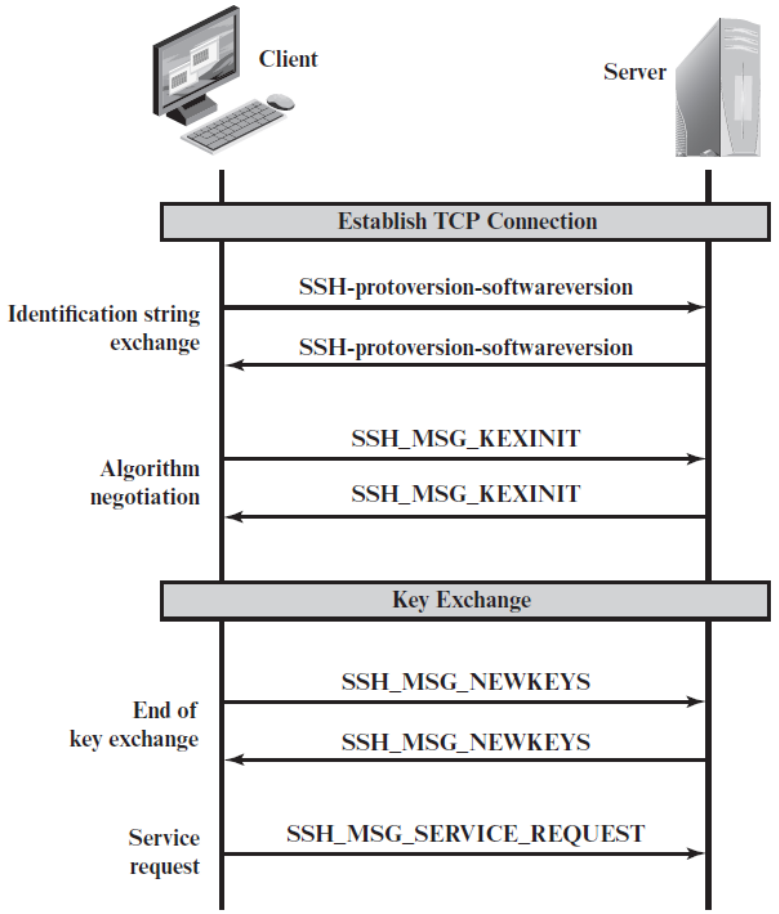
Transport Layer Protocol

PACKET EXCHANGE:

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

    SSH-protoversion-softwareversion SP comments CR LF

    `SP`, `CR`, and `LF` are space character, carriage return, and line feed, respectively.



SSH Transport Layer Protocol Packet Exchanges

Transport Layer Protocol

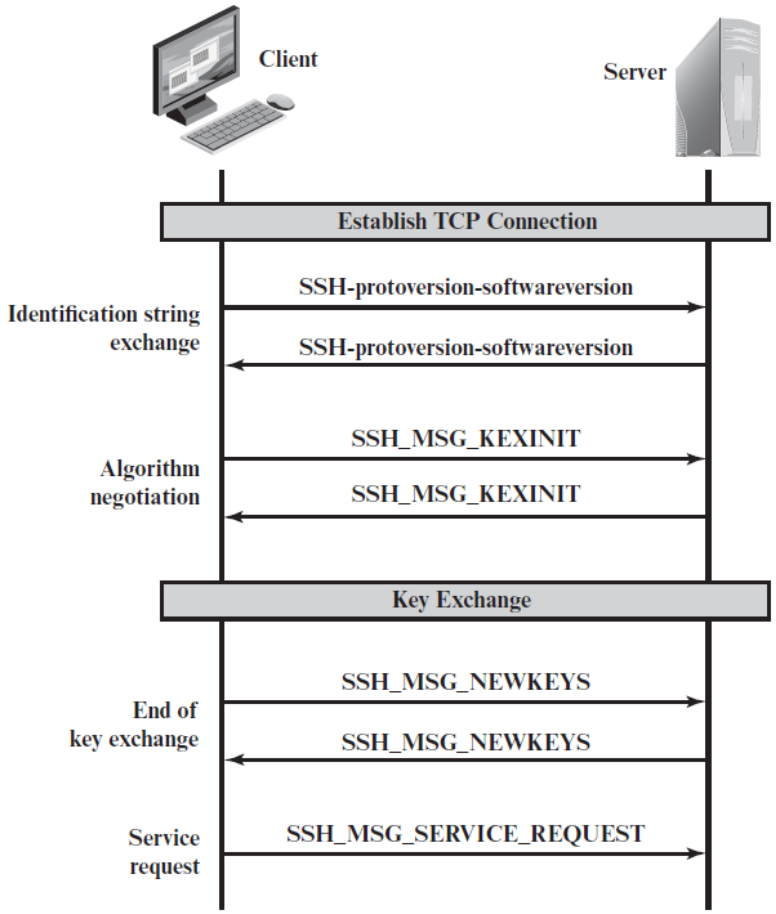
PACKET EXCHANGE:

- The server responds with its own identification string. These strings are used in the Diffie–Hellman key exchange.
- Next comes algorithm negotiation. Each side sends an

SSH_MSG_KEXINIT

containing lists of supported algorithms in the order of preference to the sender.

- There is one list for each type of cryptographic algorithm. The algorithms include **key exchange, encryption, MAC algorithm**, and **compression algorithm**.
- For each category, the algorithm chosen is the first algorithm on the client's list that is also supported by the server.



SSH Transport Layer Protocol Packet Exchanges

Transport Layer Protocol

Cipher	
3des-cbc*	Three-key 3DES in CBC mode
blowfish-cbc	Blowfish in CBC mode
twofish256-cbc	Twofish in CBC mode with a 256-bit key
twofish192-cbc	Twofish with a 192-bit key
twofish128-cbc	Twofish with a 128-bit key
aes256-cbc	AES in CBC mode with a 256-bit key
aes192-cbc	AES with a 192-bit key
aes128-cbc**	AES with a 128-bit key
Serpent256-cbc	Serpent in CBC mode with a 256-bit key
Serpent192-cbc	Serpent with a 192-bit key
Serpent128-cbc	Serpent with a 128-bit key
arcfour	RC4 with a 128-bit key
cast128-cbc	CAST-128 in CBC mode

MAC algorithm	
hmac-sha1*	HMAC-SHA1; digest length = key length = 20
hmac-sha1-96**	First 96 bits of HMAC-SHA1; digest length = 12; key length = 20
hmac-md5	HMAC-MD5; digest length = key length = 16
hmac-md5-96	First 96 bits of HMAC-MD5; digest length = 12; key length = 16

Compression algorithm	
none*	No compression
zlib	Defined in RFC 1950 and RFC 1951

* = Required
** = Recommended

SSH Transport Layer Cryptographic Algorithms

- 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.

Transport Layer Protocol

- **KEY EXCHANGE STEPS:**

1. C generates a random number $x(1 < x < q)$ and computes $e = g^x \bmod p$. C sends e to S.
2. S generates a random number $y(0 < y < q)$ and computes $f = g^y \bmod p$. S receives e. It computes $K = e^y \bmod p, H = \text{hash}(\text{V\_C} \| \text{V\_S} \| \text{I\_C} \| \text{I\_S} \| \text{K\_S} \| e \| f \| \text{K})$, and signature s on H with its private host key. S sends $(\text{K\_S} \| f \| s)$ to C. The signing operation may involve a second hashing operation.
3. C verifies that K_S really is the host key for S (e.g., using certificates or a local database). C is also allowed to accept the key without verification; however, doing so will render the protocol insecure against active attacks (but may be desirable for practical reasons in the short term in many environments). C then computes $K = f^x \bmod p, H = \text{hash}(\text{V\_C} \| \text{V\_S} \| \text{I\_C} \| \text{I\_S} \| \text{K\_S} \| e \| f \| K)$, and verifies the signature s on H.

- The values of *p*, *g*, and *q* are known to both client and server as a result of the algorithm selection negotiation.
- The hash function *hash*() is also decided during algorithm negotiation.

- C is client
- S is server
- *p* is a large **safe prime**
- *g* is a generator for a subgroup of GF(*p*)
- *q* is order of the subgroup
- V_S is S's identification string; V_C is C's identification string
- K_S is S's public host key
- I_C is C's SSH_MSG_KEXINIT message
- I_S is S's SSH_MSG_KEXINIT message

➢ A safe prime is **a prime number p of the form p = 2q + 1 where q is also prime**. In such a case, q is called a Sophie Germain prime

Transport Layer Protocol

- As a result of these steps, the two sides now share a master key *K*.
- The server has been authenticated to the client, because the server has used its private key to sign its half of the Diffie-Hellman exchange.
- The hash value *H* serves as a session identifier for this connection. Once computed, the session identifier is not changed, even if the key exchange is performed again for this connection to obtain fresh keys.
- The **end of key exchange** is signalled by the exchange of **SSH_MSG_NEWKEYS** packets. At this point, both sides may start using the keys generated from *K*
- 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.

Transport Layer Protocol

KEY GENERATION:

- The keys used for encryption and MAC (and any needed IVs) are generated from the shared secret key *K*, the hash value from the key exchange *H*, and the session identifier, which is equal to *H* unless there has been a subsequent key exchange after the initial key exchange.
  - Initial IV client to server: $\text{HASH}(K \| H \| \text{“A”} \| \text{session\_id})$
  - Initial IV server to client: $\text{HASH}(K \| H \| \text{“B”} \| \text{session\_id})$
  - Encryption key client to server: $\text{HASH}(K \| H \| \text{“C”} \| \text{session\_id})$
  - Encryption key server to client: $\text{HASH}(K \| H \| \text{“}D\text{”} \| \text{session\_id})$
  - Integrity key client to server: $\text{HASH}(K \| H \| \text{“}E\text{”} \| \text{session\_id})$
  - Integrity key server to client: $\text{HASH}(K \| H \| \text{“}F\text{”} \| \text{session\_id})$

where HASH() is the hash function determined during algorithm negotiation.

User Authentication Protocol

- Provides the means by which the client is authenticated to the server. Three types of message.
- 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

- **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)
- **method name** is authentication method being used in the request.
- The first byte has decimal value 50, which is interpreted as SSH_MSG_USERAUTH_REQUEST

- Server either **rejects** the authentication request or **accepts** the request but requires one or more additional authentication methods.
- Server sends a message with the format:
  - **name-list** is a list of methods that may productively continue the dialog

byte	SSH_MSG_USERAUTH_FAILURE (51)
name-list	authentications that can continue
boolean	partial success

- If the server accepts authentication, it sends a single byte message: SSH_MSG_USERAUTH_SUCCESS (52)

User Authentication Protocol

MESSAGE EXCHANGE

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.

User Authentication Protocol

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.
- That 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.

Connection Protocol

CHANNEL MECHANISM:

- 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

- The channel type identifies the application for this channel
- The sender channel is the local channel number.
- initial window size specifies how many bytes of channel data can be sent to the sender of this message without adjusting the window.
- Maximum packet size specifies the maximum size of an individual data packet that can be sent to the sender.

Connection Protocol

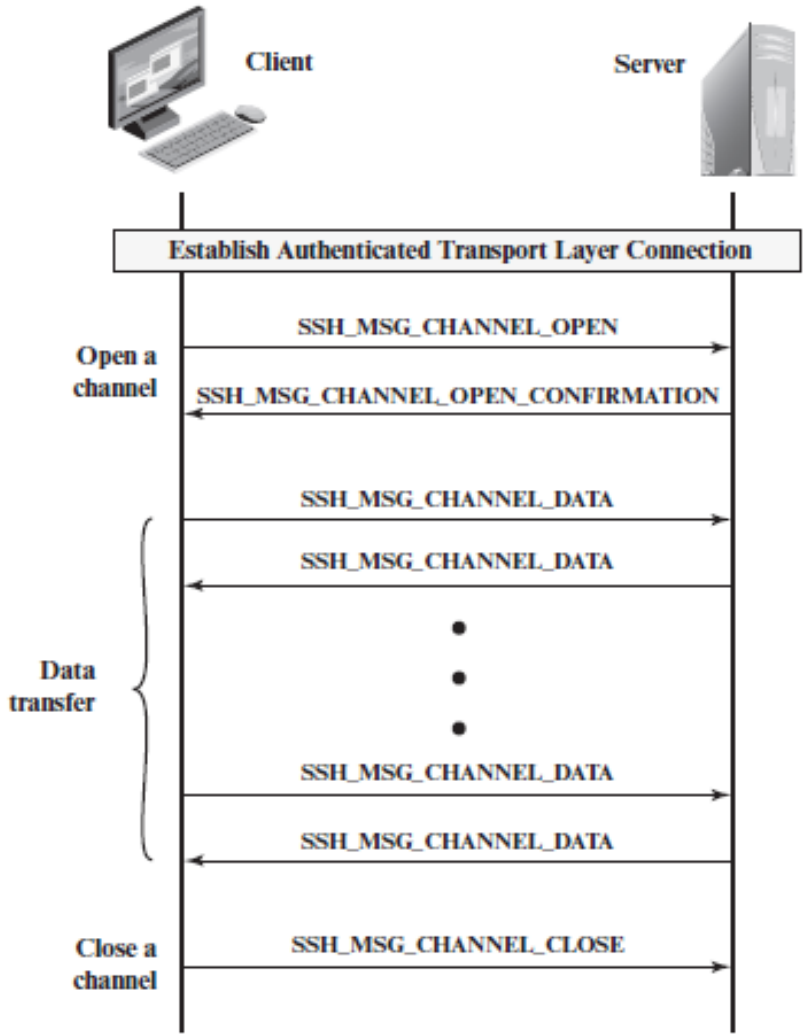
CHANNEL MECHANISM:

- 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.

Connection Protocol

Example of SSH Connection Protocol Message Exchange



Connection Protocol

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 email, 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.

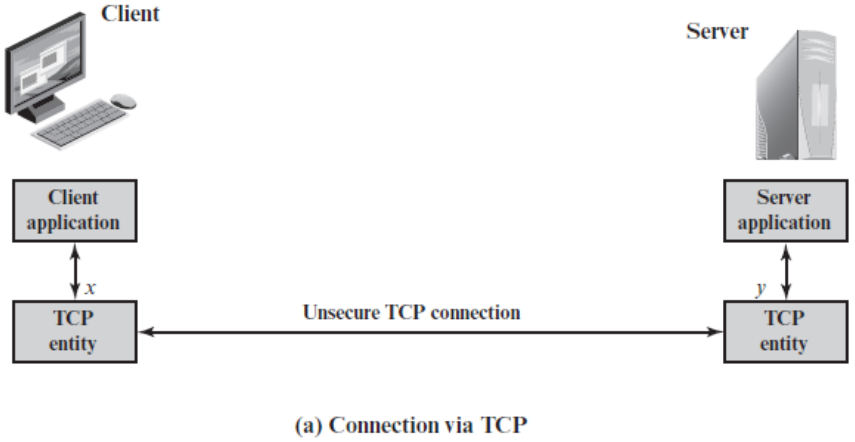
- **forwarded-tcpip:** This is remote port forwarding
- **direct-tcpip:** This is local port forwarding

Connection Protocol

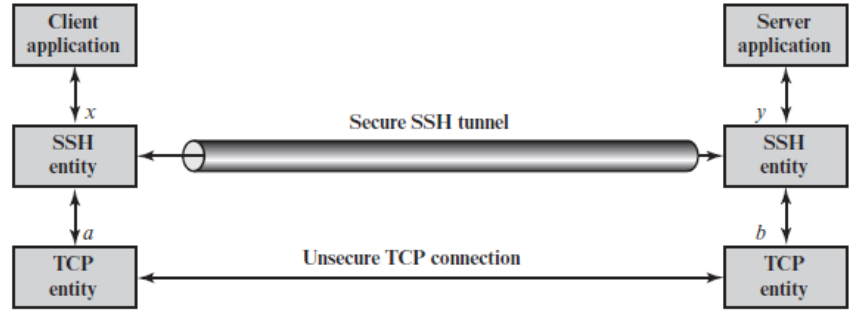
PORT FORWARDING

- One of the most useful features of SSH is port forwarding.
- provides the ability to convert any insecure TCP connection into a secure SSH connection. This is also referred to as **SSH tunneling**.
- A **port** is an identifier of a user of TCP.
  - 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.

Connection Protocol



(a) Connection via TCP



(b) Connection via SSH tunnel

- A client application 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.

Connection Protocol

- 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.

Connection Protocol

- **Local forwarding Example**:

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

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

Connection Protocol

Remote Forwarding

- With **remote forwarding**, the user's SSH client acts on the server's behalf.
- The client receives traffic with a given destination port number, places the traffic on the correct port and sends it to the destination the user chooses.
- **Example:**

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.