



---

# Chapter 12

## Security Protocols of the Transport Layer

- ❑ Secure Socket Layer (SSL)
- ❑ Transport Layer Security (TLS)
- ❑ Secure Shell (SSH)

# Scope of Transport Layer Security Protocols

---



- ❑ The transport layer provides communications between application processes (instead of end-systems) and its main tasks are:
  - ❑ Isolation of higher protocol layers from the technology, structure and deficiencies of deployed communications technology
  - ❑ Transparent transmission of user data
  - ❑ Global addressing of application processes, independently of lower layer addresses (Ethernet addresses, telephone numbers, etc.)
  - ❑ Overall goal: provision of an efficient and reliable service
- ❑ ***Transport layer security protocols*** aim to enhance the service of the transport layer by assuring additional security properties
  - ❑ As they usually require and are build upon a reliable transport service, they actually represent *session layer protocols* according to the terminology of the *Open Systems Interconnection (OSI) reference model*
  - ❑ However, as OSI is no longer “en vogue” they are called transport layer security protocols

# The Secure Socket Layer (SSL) Protocol

---



- ❑ SSL was originally designed to primarily protect HTTP sessions:
  - ❑ In the early 1990's there was a similar protocol called S-HTTP
  - ❑ However, as S-HTTP capable browsers were not free of charge and SSL version 2.0 was included in browsers of Netscape Communications, it quickly became predominant
  - ❑ SSL v.2 contained some flaws and so Microsoft Corporation developed a competing protocol called Private Communication Technology (PCT)
  - ❑ Netscape improved the protocol and SSL v.3 became the de-facto standard protocol for securing HTTP traffic
  - ❑ Nevertheless, SSL can be deployed to secure arbitrary applications that run over TCP
  - ❑ In 1996 the IETF decided to specify a generic *Transport Layer Security (TLS)* protocol that is based on SSL

# SSL Security Services

---



- ❑ *Peer entity authentication:*
  - ❑ Prior to any communications between a client and a server, an authentication protocol is run to authenticate the peer entities
  - ❑ Upon successful completion of the authentication dialogue an *SSL session* is established between the peer entities
- ❑ *User data confidentiality:*
  - ❑ If negotiated upon session establishment, user data is encrypted
  - ❑ Different encryption algorithms can be negotiated: RC4, DES, 3DES, IDEA
- ❑ *User data integrity:*
  - ❑ A MAC based on a cryptographic hash function is appended to user data
  - ❑ The MAC is computed with a negotiated secret in prefix-suffix mode
  - ❑ Either MD5 or SHA-1 can be negotiated for MAC computation

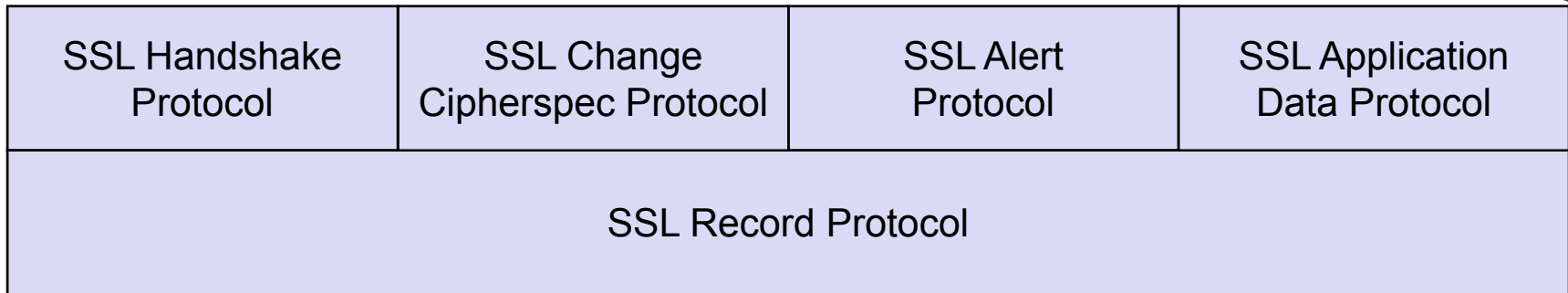
# SSL Session & Connection State

---



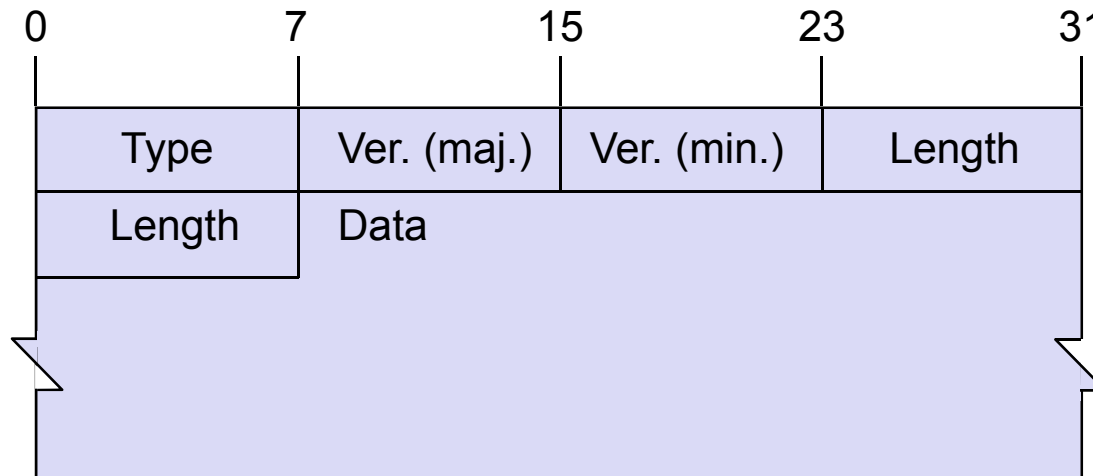
- ❑ Session state:
  - ❑ *Session identifier*: a byte sequence chosen by the server
  - ❑ *Peer certificate*: X.509 v.3 certificate of the peer (optional)
  - ❑ *Compression method*: algorithm to compress data prior to encryption
  - ❑ *Cipher spec*: specifies cryptographic algorithms and parameters
  - ❑ *Master secret*: a negotiated shared secret of length 48 byte
  - ❑ *Is resumable*: a flag indicating if the session supports new connections
  
- ❑ Connection state:
  - ❑ *Server and client random*: byte sequences chosen by server and client
  - ❑ *Server write MAC secret*: used in MAC computations by the server
  - ❑ *Client write MAC secret*: used in MAC computations by the client
  - ❑ *Server write key*: used for encryption by server and decryption by client
  - ❑ *Client write key*: used for encryption by client and decryption by server

# SSL Protocol Architecture



- ❑ SSL is structured as a layered and modular protocol architecture:
  - ❑ Handshake: authentication and negotiation of parameters
  - ❑ Change Cipherspec: signaling of transitions in ciphering strategy
  - ❑ Alert: signaling of error conditions
  - ❑ Application Data: interface for transparent access to the record protocol
  - ❑ Record:
    - Fragmentation of user data into plaintext records of length  $< 2^{14}$
    - Compression (optional) of plaintext records
    - Encryption and integrity protection (both optional)

# SSL Record Protocol



- ❑ Content Type:
  - ❑ Change Cipherspec. (20)
  - ❑ Alert (21)
  - ❑ Handshake (22)
  - ❑ Application Data (23)
- ❑ Version: the protocol version of SSL (major = 3, minor = 0)
- ❑ Length: the length of the data in bytes, may not exceed  $2^{14} + 2^{10}$

# SSL Record Protocol Processing



## ❑ Sending side:

- ❑ The record layer first fragments user data into records of a maximum length of  $2^{14}$  octets, more than one message of the same content type can be assembled into one record
- ❑ After fragmentation the record data is compressed, the default algorithm for this is *null* (~ no compression), and it may not increase the record length by more than  $2^{10}$  octets
- ❑ A message authentication code is appended to the record data:
  - $MAC = H(MAC\_write\_secret + pad\_2 + H(MAC\_write\_secret + pad\_1 + seqnum + length + data))$
  - Note, that seqnum is not transmitted, as it is known implicitly and the underlying TCP offers an assured service
- ❑ The record data and the MAC are encrypted using the encryption algorithm defined in the current cipherspec (may imply prior padding)

## ❑ Receiving side:

- ❑ The record is decrypted, integrity-checked, decompressed, de-fragmented and delivered to the application or SSL higher layer protocol

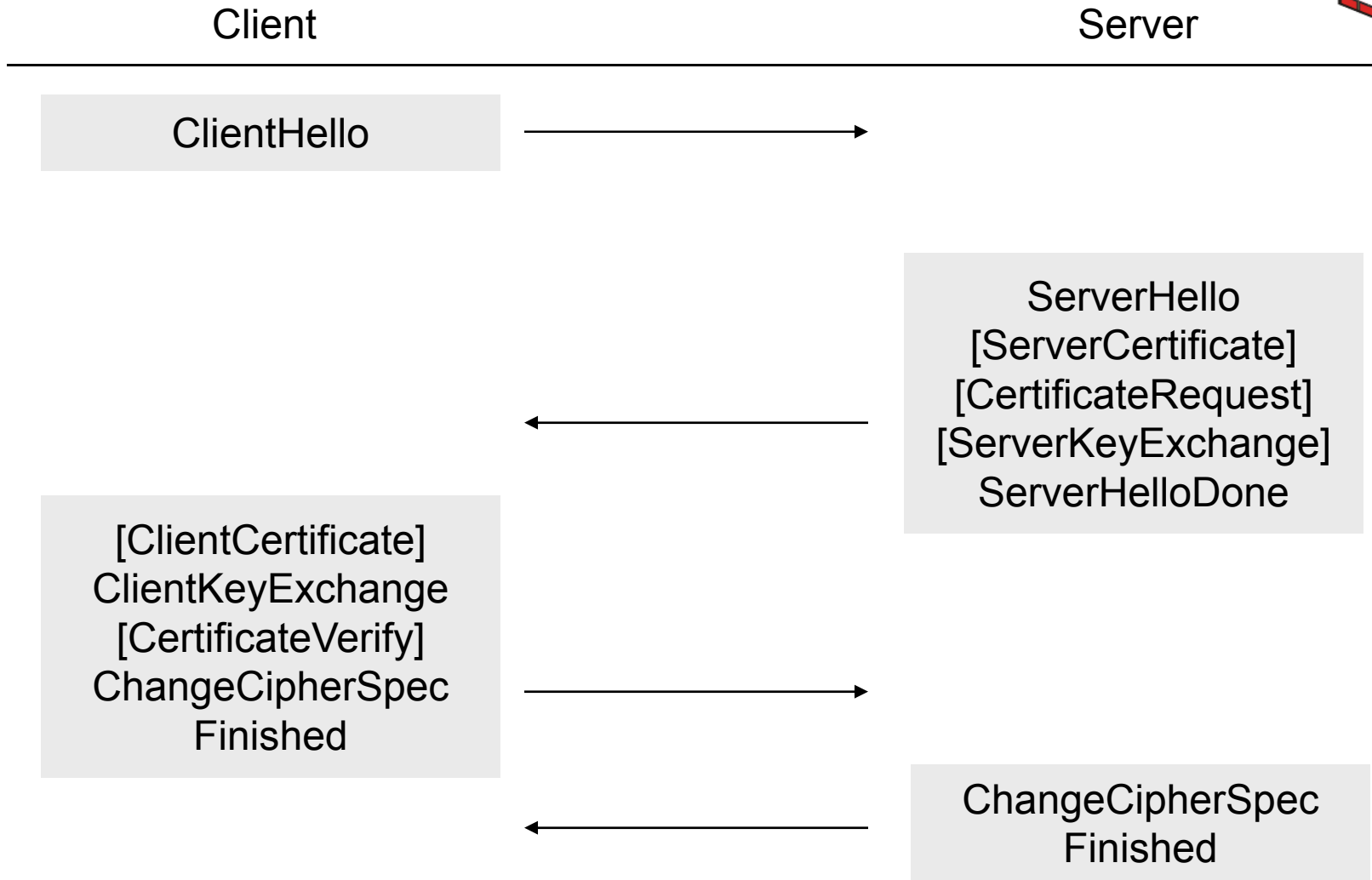
# SSL Handshake Protocol: Introduction

---



- ❑ The SSL handshake protocol is used to establish peer authentication and cryptographic parameters for an SSL session
- ❑ An SSL session can be negotiated to be resumable:
  - ❑ Resuming and duplicating SSL sessions allows to re-use established security context
  - ❑ This is very important for securing HTTP traffic, as usually every item on a web page is transferred an individual TCP connection
  - ❑ When resuming / duplicating an existing session, an abbreviated handshake is performed

# SSL Handshake Protocol: Full Handshake





# SSL Handshake Protocol: Cryptographic Aspects



- ❑ SSL supports three methods for establishing session keys:
  - ❑ *RSA*: a *pre-master-secret* is randomly generated by the client and sent to the server encrypted with the servers public key
  - ❑ *Diffie-Hellman*: a standard Diffie-Hellman exchange is performed and the established shared secret is taken as *pre-master-secret*
  - ❑ *Fortezza*: an unpublished security technology developed by the NSA, that supports key escrow and that is not discussed in this class
- ❑ SSL was primarily designed to secure HTTP traffic → a client wishing to access an authentic web-server is the “default application scenario”:
  - ❑ In this case the web-server sends its public key certificate after the ServerHello message
  - ❑ The server certificate may contain the server’s public DH-values or the server may send them in the optional ServerKeyExchange message
  - ❑ The client uses the server’s certificate / the received DH-values / its Fortezza card to perform an RSA- / DH- / Fortezza-based key exchange

# SSL Handshake Protocol: Cryptographic Aspects



- ❑ The pre-master-secret and the random numbers provided by the client and the server in their hello-messages are used to generate the master-secret of length 48 byte
- ❑ Computation of the master-secret:
  - ❑  $\text{master-secret} = \text{MD5}(\text{pre-master-secret} + \text{SHA}(\text{'A'} + \text{pre-master-secret} + \text{ClientHello.random} + \text{ServerHello.random})) + \text{MD5}(\text{pre-master-secret} + \text{SHA}(\text{'BB'} + \text{pre-master-secret} + \text{ClientHello.random} + \text{ServerHello.random})) + \text{MD5}(\text{pre-master-secret} + \text{SHA}(\text{'CCC'} + \text{pre-master-secret} + \text{ClientHello.random} + \text{ServerHello.random}))$
- ❑ The use of both MD5 and SHA to generate the master-secret is considered to provide security even in case that one of the cryptographic hash functions is “broken”

# SSL Handshake Protocol: Cryptographic Aspects



- ❑ To compute the session keys from the master-secret, a sufficient amount of key material is generated from the master-secret and the client's and server's random numbers in a first step:
  - ❑  $\text{key\_block} = \text{MD5}(\text{master-secret} + \text{SHA}(\text{'A'} + \text{master-secret} + \text{ClientHello.random} + \text{ServerHello.random})) + \text{MD5}(\text{master-secret} + \text{SHA}(\text{'BB'} + \text{master-secret} + \text{ClientHello.random} + \text{ServerHello.random})) + \dots$
  
- ❑ Then, the session key material is consecutively taken from the `key_block`:
  - ❑ `client_write_MAC_secret` = `key_block[1, CipherSpec.hash_size]`
  - ❑ `server_write_MAC_secret` = `key_block[i1, i1 + CipherSpec.hash_size - 1]`
  - ❑ `client_write_key` = `key_block[i2, i2 + CipherSpec.key_material - 1]`
  - ❑ `server_write_key` = `key_block[i3, i3 + CipherSpec.key_material - 1]`
  - ❑ `client_write_IV` = `key_block[i4, i4 + CipherSpec.IV_size - 1]`
  - ❑ `server_write_IV` = `key_block[i5, i5 + CipherSpec.IV_size - 1]`

# SSL Handshake Protocol: Cryptographic Aspects

---



- ❑ Authentication of and with the pre-master-secret:
  - ❑ SSL supports key establishment without authentication (anonymous), in this case man-in-the-middle attacks can not be defended
  - ❑ When using the RSA-based key exchange:
    - The client encrypts the pre-master-secret with the server's public key which can be verified by a certificate chain
    - The client knows that only the server can decrypt the pre-master-secret, thus when the server sends the Finished message using the master-secret, the client can deduce server-authenticity
    - The server can not deduce any client authenticity from the received pre-master-secret
    - If client authenticity is required, the client additionally sends its certificate and a CertificateVerify message that contains a signature over a hash (MD5 or SHA) of the master-secret and all handshake messages exchanged before the CertificateVerify message
  - ❑ With DH-key-exchange, authenticity is deduced from the DH-values which are contained and signed in the server's (and client's) certificate

# SSL Cipher-Suites

---



- ❑ Any combination of the following methods for
  - ❑ Entity authentication and key exchange
  - ❑ Encryption
  - ❑ Message authenticationis supported by SSL.v3
  
- ❑ Entity authentication and key exchange
  - ❑ None, RSA, DH (signed or anonymous), Fortezza
  
- ❑ Encryption
  - ❑ None, RC4, IDEA, DES, 3DES
  
- ❑ Message authentication
  - ❑ None, MD5, SHA-1

# The Transport Layer Security (TLS) Protocol

---



- ❑ In 1996 the IETF started a working group to define a *transport layer security (TLS)* protocol:
  - ❑ Officially, the protocols SSL, SSH and PCT were announced to be taken as input
  - ❑ However, the TLS V.1.0 specification draft published in December 1996 was essentially the same as the SSL V.3.0 specification
- ❑ Actually, the intention of the working group was from the beginning to base TLS on SSL V.3.0 with the following modifications:
  - ❑ The HMAC construction for computing cryptographic hash values should be adopted instead of hashing in prefix and suffix mode
  - ❑ The Fortezza based cipher-suites of SSL should be removed, as they include an unpublished technology
  - ❑ A DSS (digital signature standard) based authentication and key exchange dialogue should be included
  - ❑ The TLS Record Protocol and the Handshake Protocol should be separated out and specified more clearly in separated documents, which actually did not happen

# The Transport Layer Security Protocol

---



- ❑ In order to achieve exportability of TLS compliant products, some cipher-suites specify the use of keys with entropy reduced to 40 bit:
  - ❑ These cipher-suites contain the word “export” in their name
  - ❑ As the government of the USA changed its policy concerning the export of cryptographic products, this is of less importance today
  - ❑ The use of these cipher-suites is strongly discouraged, as they offer virtually no data confidentiality protection
- ❑ Key exchange algorithms:
  - ❑ DH exchange without or with DSS / RSA signatures
  - ❑ DH exchange with certified public DH parameters
  - ❑ RSA based key exchange
  - ❑ none
- ❑ Encryption algorithms: IDEA / DES / 3DES / RC2 in CBC, RC4, null
- ❑ Hash algorithms: MD5, SHA, null
- ❑ Concerning its protocol functions, TLS is essentially the same like SSL

# The Secure Shell (SSH) Protocol

---



- ❑ *Secure Shell (SSH) Version 1* was originally developed by Tatu Ylönen at the Helsinki University of Finland
  - ❑ As the author also provided a free implementation with source code, the protocol found widespread use in the Internet
  - ❑ Later on, the development of SSH was commercialized by the author
  - ❑ Nevertheless, free versions are still available with the most widely deployed version being OpenSSH
- ❑ In 1997 a version 2.0 specification of SSH was submitted to the IETF and has been refined in a series of Internet Drafts since
- ❑ SSH was originally designed to provide a secure replacement for the Unix r-tools (rlogin, rsh, rcp, and rdist), thus it represents an application or session-layer protocol
- ❑ However, as SSH also includes a generic transport layer security protocol and offers tunneling capabilities, it is discussed in this chapter as a transport layer security protocol

# SSH Version 2

---



- ❑ SSH Version 2 is specified in four separate documents:
  - ❑ SSH Protocol Architecture [YKS01a]
  - ❑ SSH Transport Layer Protocol [YKS01b]
  - ❑ SSH Authentication Protocol [YKS01c]
  - ❑ SSH Connection Protocol [YKS01d]
- ❑ SSH Architecture:
  - ❑ SSH follows a client-server approach
  - ❑ Every SSH server has at least one host key
  - ❑ SSH version 2 offers two different trust models:
    - Every client has a local database that associates each host name with the corresponding public host key
    - The hostname to public key association is certified by a CA and every client knows the public key of the CA
  - ❑ The protocol allows full negotiation of encryption, integrity, key exchange, compression, and public key algorithms and formats

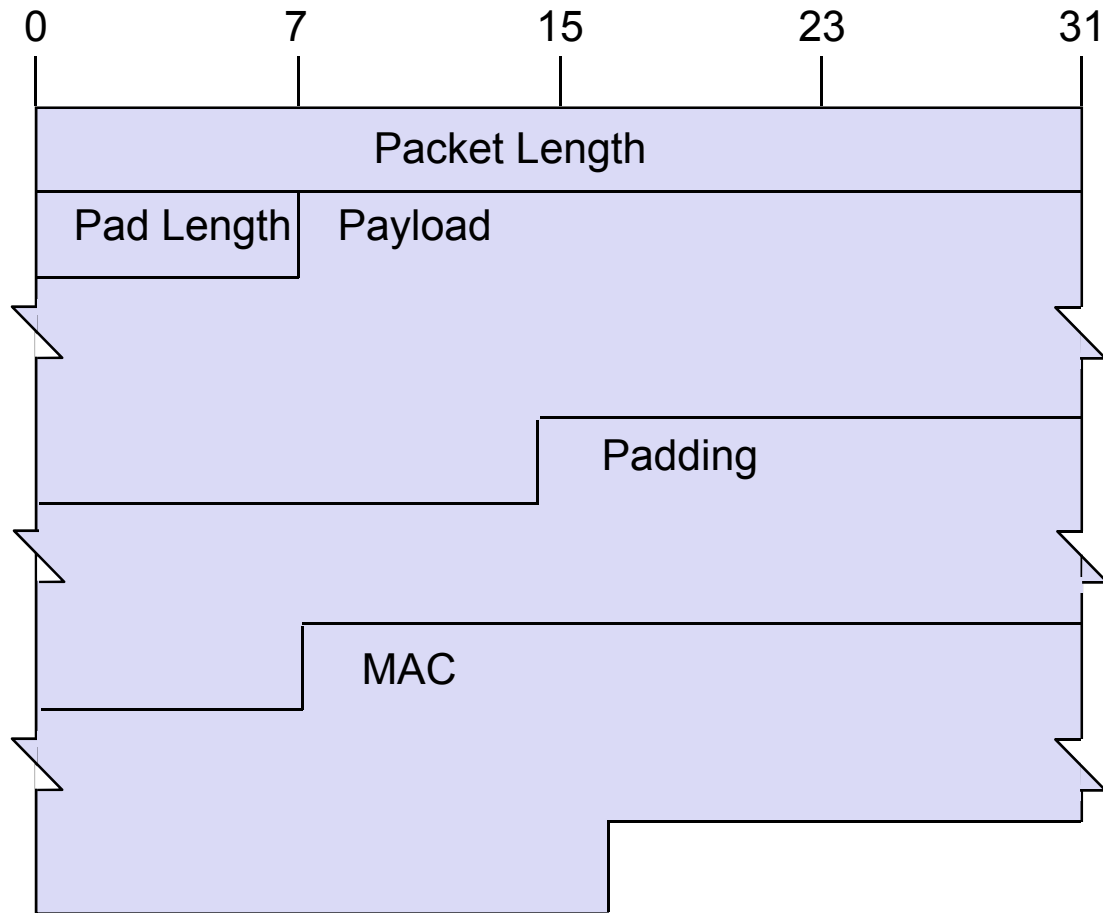
# SSH Transport Protocol

---



- ❑ SSH Transport Protocol needs a reliable transport protocol (e.g. TCP)
- ❑ It provides the following services:
  - ❑ Encryption of user data
  - ❑ Data origin authentication (integrity)
  - ❑ Server authentication (host authentication only)
  - ❑ Compression of user data prior to encryption
- ❑ Supported algorithms:
  - ❑ Encryption:
    - 3DES, Blowfish, Twofish, AES, Serpent, IDEA, CAST in CBC
    - none (not recommended)
  - ❑ Integrity: HMAC with MD5 or SHA-1, none (not recommended)
  - ❑ Key exchange: Diffie-Hellman with SHA-1 and one pre-defined group
  - ❑ Public key: RSA, DSS
  - ❑ Compression: none, zlib (see RFCs 1950, 1951)

# SSH Transport Protocol Packet Format



- ❑ The packet format is not 32-bit-word aligned

# SSH Transport Protocol Packet Format



- ❑ Packet fields:
  - ❑ *Packet length*: the length of the packet itself, not including this length field and the MAC
  - ❑ *Padding length*: length of the padding field, must be between four and 255
  - ❑ *Payload*: the actual payload of the packet, if compression is negotiated this field is compressed
  - ❑ *Padding*: this field consists of randomly chosen octets to fill up the payload to an integer multiple of 8 or the block size of the encryption algorithm, whichever is larger
  - ❑ *MAC*: if message authentication has been negotiated this contains the MAC over the entire packet without the MAC field itself, if the packet is to be encrypted the MAC is computed prior to encryption as follows:
    - $MAC = HMAC(shared\_secret, seq\_number || unencrypted\_packet)$   
with *seq\_number* denoting a 32-bit sequence number for every packet
- ❑ Encryption: if encryption is negotiated, the entire packet without the MAC is encrypted after MAC computation

# SSH Negotiation, Key Exchange & Server Authentication



- ❑ Algorithm Negotiation:
  - ❑ Each entity sends a packet (referred to as *kexinit*) with a specification of methods it support, in the order of preference
  - ❑ Both entities iterate over the list of the client and chose the first algorithm that is also supported by the server
  - ❑ This method is used to negotiate: server-host-key algorithm (~ server authentication), as well as encryption, MAC, and compression algorithm
  - ❑ Additionally, either entity may attach a key exchange packet according to a guess of the preferred key exchange algorithm of the other entity
  - ❑ If a guess is right, the corresponding key exchange packet is accepted as the first key exchange packet of the other entity
  - ❑ Wrong guesses are ignored and new key exchange packets are sent after algorithm negotiation
- ❑ For key exchange [YKS01b] defines only one method:
  - ❑ Diffie-Hellman with SHA-1 and a predefined group
  - ❑  $p = 2^{1024} - 2^{960} - 1 + (2^{64} \times \lfloor 2^{894} \times \pi + 129093 \rfloor)$ ;  $g = 2$ ; order =  $(p - 1) / 2$

# SSH Negotiation, Key Exchange & Server Authentication



- ❑ If key exchange is realized with the pre-defined DH group:
  - ❑ The client chooses a random number  $x$ , computes  $e = g^x \bmod p$  and sends  $e$  to the server
  - ❑ The server chooses a random number  $y$ , computes  $f = g^y \bmod p$
  - ❑ Upon reception of  $e$ , the server further computes  $K = e^y \bmod p$  and a hash value  $h = \text{Hash}(\text{version}_C, \text{version}_S, \text{kexinit}_C, \text{kexinit}_S, +K_S, e, f, K)$  with *version* and *kexinit* denoting the client's and server's version information and initial algorithm negotiation messages
  - ❑ The server signs  $h$  with its private host key  $-K_S$  and sends to the client a message containing  $(+K_S, f, s)$
  - ❑ Upon reception the client checks the host key  $+K_S$ , computes  $K = f^x \bmod p$  as well as the hash value  $h$  and then checks the signature  $s$  over  $h$
- ❑ After performing these checks, the client can be sure that he has in fact negotiated a secret  $K$  with the host that knows  $-K_S$
- ❑ However, the server host can not deduce anything about the client's authenticity, for this purpose the SSH authentication protocol is used

# SSH Session Key Derivation



- ❑ The key exchange method allows to establish a shared secret  $K$  and the hash value  $h$  which are used to derive the SSH session keys:
  - ❑ The hash  $h$  of the initial key exchange is also taken as the *session\_id*
  - ❑  $IV_{Client2Server} = Hash(K, h, "A", session\_id)$  // initialization vector
  - ❑  $IV_{Server2Client} = Hash(K, h, "B", session\_id)$  // initialization vector
  - ❑  $EK_{Client2Server} = Hash(K, h, "C", session\_id)$  // encryption key
  - ❑  $EK_{Server2Client} = Hash(K, h, "D", session\_id)$  // encryption key
  - ❑  $IK_{Client2Server} = Hash(K, h, "E", session\_id)$  // integrity key
  - ❑  $IK_{Server2Client} = Hash(K, h, "F", session\_id)$  // integrity key
- ❑ Key data is taken from the beginning of the hash output
- ❑ If more key bits are needed than produced by the hash function:
  - ❑  $K1 = Hash(K, h, x, session\_id)$  //  $x = "A", "B", \text{etc.}$
  - ❑  $K2 = Hash(K, h, K1)$
  - ❑  $K2 = Hash(K, h, K1, K2)$
  - ❑  $XK = K1 || K2 || \dots$

# SSH Authentication Protocol



- ❑ The SSH authentication protocol serves to verify the client's identity and it is intended to be run over the SSH transport protocol
- ❑ The protocol per default supports the following authentication methods:
  - ❑ *Public key*: the user generates and sends a signature with a per user public key to the server  
$$\text{Client} \rightarrow \text{Server: } E(-K_{User}, (\text{session\_id}, 50, \text{Name}_{User}, \text{Service}, \text{"publickey"}, \text{True}, \text{PublicKeyAlgorithmName}, +K_{User}))$$
  - ❑ *Password*: transmission of a per user password in the encrypted SSH session (the password is presented in clear to the server but transmitted with SSH transport protocol encryption)
  - ❑ *Host-based*: analogous to public key but with with per host public key
  - ❑ *None*: used to query the server for supported methods and if no authentication is required (server directly responds with success message)
- ❑ If the client's authentication message is successfully checked, the server responds with a *ssh\_msg\_userauth\_success* message

# SSH Connection Protocol

---



- ❑ The SSH connection protocol runs on top of the SSH transport protocol and provides the following services:
  - ❑ Interactive login sessions
  - ❑ Remote execution of commands
  - ❑ Forwarded TCP/IP connections
  - ❑ Forwarded X11 connections
- ❑ For each of the above services one or more “*channels*” are established, and all channels are multiplexed into a single encrypted and integrity protected SSH transport protocol connection:
  - ❑ Either side may request to open a channel and channels are identified by numbers at the sender and receiver
  - ❑ Channels are typed, e.g. “*session*”, “*x11*”, “*forwarded-tcpip*”, “*direct-tcpip*”...
  - ❑ Channels are flow-controlled by a window mechanism and no data may be sent via a channel before “window space” is available

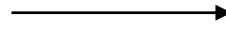
# Requesting an Interactive Session and Starting a Shell



SSH Client

SSH Server

```
ssh_msg_channel_open  
("session", 20, 2048, 512)
```



```
ssh_msg_channel_open_  
confirmation(20, 31, 1024, 256)
```



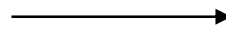
```
ssh_msg_channel_request  
(31, "pty-req", false, ...)
```



```
ssh_msg_channel_request  
(31, "env", false, "home",  
"/home/username")
```



```
ssh_msg_channel_request  
(31, "shell", true, ...)
```



```
ssh_msg_channel_success(20)
```



[Use data exchange takes place from now on...]

# Summary (what do I need to know)

---



- ❑ Principles of transport layer security
  - ❑ Application-specific security w/o modified network / OS
  - ❑ Independent of specific network protocols (IPv4 / IPv6)
  - ❑ Requires reliable transport (e.g., TCP)
  
- ❑ SSL / SSH
  - ❑ Security objectives
  - ❑ Operation principles

# Additional References

---



- [Cop96a] D. Coppersmith, M. K. Franklin, J. Patarin, M. K. Reiter. *Low Exponent RSA with Related Messages*. In *Advance in Cryptology -- Eurocrypt'96*, U. Maurer, Ed., vol. 1070 of *Lectures Notes in Computer Science*, Springer-Verlag, 1996.
- [FKK96a] A. O. Freier, P. Karlton, P. C. Kocher. *The SSL Protocol Version 3.0*. Netscape Communications Corporation, 1996.
- [RFC2246] T. Dierks, C. Allen. *The TLS Protocol Version 1.0*. RFC 2246, 1999.
- [YKS01a] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen. *SSH Protocol Architecture*. Internet Draft (work in progress), draft-ietf-secsh-architecture-09.txt, 2001.
- [YKS01b] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen. *SSH Transport Layer Protocol*. Internet Draft (work in progress), draft-ietf-secsh-transport-09.txt, 2001.
- [YKS01c] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen. *SSH Authentication Protocol*. Internet Draft (work in progress), draft-ietf-secsh-userauth-11.txt, 2001.
- [YKS01d] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen. *SSH Connection Protocol*. Internet Draft (work in progress), draft-ietf-secsh-connect-11.txt, 2001.