Network Security

Chapter 12 Security Protocols of the Transport Layer

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

Scope of Transport Layer Security Protocols

- ❑ The transport layer provides communications between application processes (instead of communications between 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

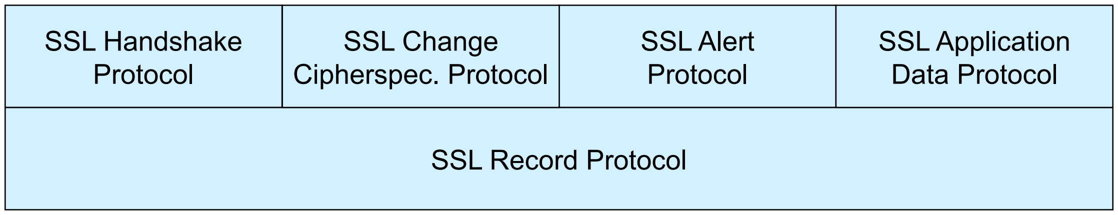
SSL Security Services

- *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 can be negotiated for MAC computation

SSL Session & Connection State

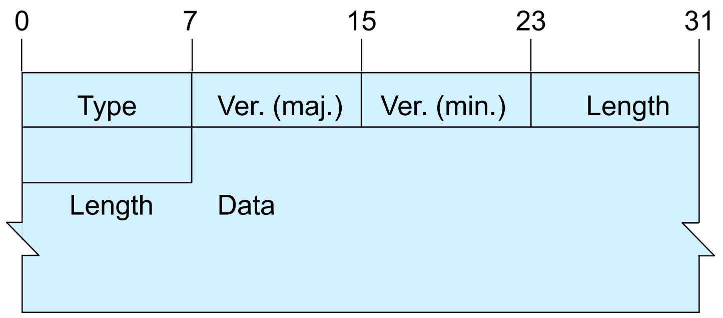
- Session state:
  - *Session identifier:* a byte sequence chosen by the server
  - *Peer certificate:* X.509 v.3 certificate of the peer (optional)
  - *Compression method:* algorithm to compress data prior to encryption
  - *Cipher spec:* specifies cryptographic algorithms and parameters
  - *Master 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 Handshake Protocol
SSL Change Cipherspec. Protocol
SSL Alert Protocol
SSL Application Data Protocol
SSL Record Protocol

- 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 $2^{14} + 2^{10}$

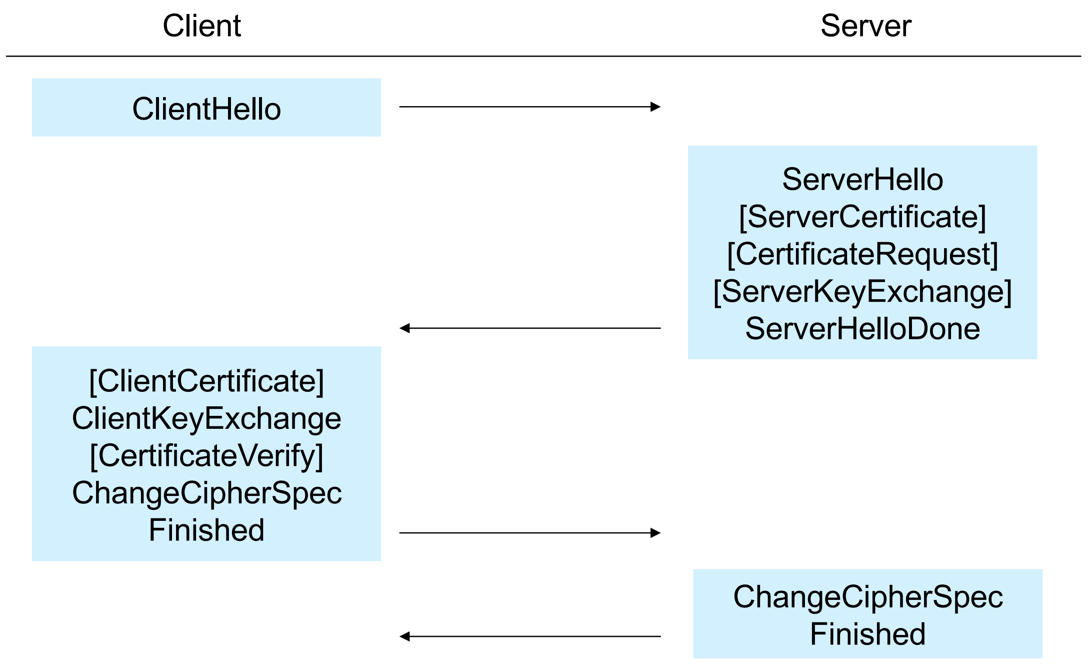
SSL Record Protocol Processing

- 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

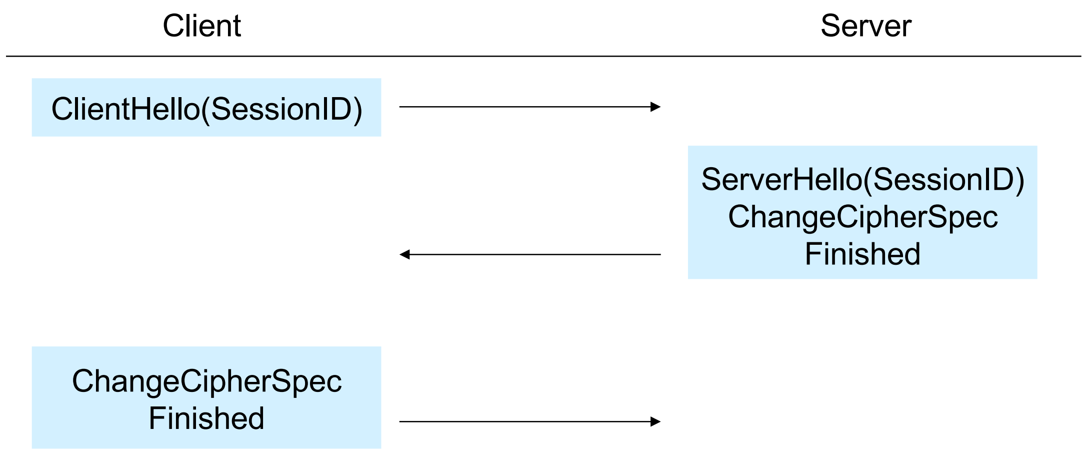
- The SSL handshake protocol is used to establish peer authentication and cryptographic parameters for an SSL session
- An SSL session can be negotiated to be resumable:
  - Resuming and duplicating SSL sessions allows to re-use established security context
  - This is very important for securing HTTP traffic, as usually every item on a web page is transferred an individual TCP connection
    - Since HTTP 1.1 persistent TCP connections are used
    - Nevertheless, resuming SSL sessions still makes a lot sense, as persistent TCP connections may be closed after downloading all items that belong to one page and some period of inactivity by the user.
  - When resuming / duplicating an existing session, an abbreviated handshake is performed

SSL Handshake Protocol: Full Handshake



[...] denotes optional messages

SSL Handshake Protocol: Abbreviated Handshake



- The *Finished* message contains a MAC based on either MD5 or SHA including the master-secret previously established between client and server
- If the server can not resume / decides not to resume the session it answers with the messages of the full handshake

SSL Handshake Protocol: Cryptographic Aspects (1)

- 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
- As SSL was primarily designed to secure HTTP traffic, its “default application scenario” is a client wishing to access an authentic web-server:
  - 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 (2)

- 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:
  - Master-secret = MD5(pre-master-secret + SHA('A' + pre-master-secret + ClientHello.random + ServerHello.random)) +
    MD5(pre-master-secret + SHA('BB' + pre-master-secret + ClientHello.random + ServerHello.random)) +
    MD5(pre-master-secret + SHA('CCC' + pre-master-secret + ClientHello.random + 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 (3)

- 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:
  - key_block = MD5(master-secret + SHA('A' + master-secret + ClientHello.random + ServerHello.random)) +
    MD5(master-secret + SHA('BB' + master-secret + ClientHello.random + ServerHello.random)) +
    [...]
- 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[i_1, i_1 + CipherSpec.hash_size - 1]
client_write_key	= key_block[i_2, i_2 + CipherSpec.key_material - 1]
server_write_key	= key_block[i_3, i_3 + CipherSpec.key_material - 1]
client_write_IV	= key_block[i_4, i_4 + CipherSpec.IV_size - 1]
server_write_IV	= key_block[i_5, i_5 + CipherSpec.IV_size - 1]

SSL Handshake Protocol: Cryptographic Aspects (4)

- 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 Handshake Protocol: A Vulnerability (1)

- In 1998, D. Bleichenbacher discovered a vulnerability in the PKCS #1 (v.1.5) encryption standard that is used in the SSL handshake method
- When the client encrypts the pre-master-secret with the public key of the server, it uses PKCS #1 to format it prior to encryption:
  - *EM = 0x02 | PS | 0x00 | M*
    with *PS* denoting a padding string of at least 8 pseudo-randomly generated nonzero octets, and *M* denoting the message to be encrypted (= pre-master-secret)
    (*PS* is used to add a random component and fill up *M* to the modulus size of the deployed key)
  - Then *EM* is encrypted: $C = E(+K_{Server}, EM)$
  - After the server has decrypted *C*, it checks if the first octet is equal to 0x02 and if there is a 0x00 octet, if this check fails it answers with an error message
  - This reporting of errors can be utilized by an attacker to launch an "oracle-attack"

SSL Handshake Protocol: A Vulnerability (2)

- An oracle-attack against the SSL handshake protocol [BKS98a]:
  - Consider an attacker (Eve) that eavesdropped on a SSL handshake dialogue and that wants to recover the pre-master-secret (and with this all other derived secrets) exchanged between Alice (client) and Bob (server)
  - Eve has successfully eavesdropped the encrypted message *C* containing the pre-master-secret and now wants to recover the plaintext
  - Eve generates a series of related ciphertexts C_1, C_2, ...:
    - $C_i = C \times R_i^e \bmod n$ with (e, n) being the public key of Bob
    - The R_i are chosen in an adaptive way, depending on older "good" R_i that have been processed by Bob without generating error messages (indicating that they have been decrypted to a valid PKCS #1 message)
    - The C_i are submitted to Bob and new C_i are generated accordingly
    - From the "good" R_i, Eve deduces certain bits of the corresponding message $M_i = C_i^d = M \times R_i \bmod n$, based on the PKCS #1 encoding method

SSL Handshake Protocol: A Vulnerability (3)

- An oracle-attack against the SSL handshake protocol (cont.):
  - From the inferred bits of $M \times R_i \bmod n$ for sufficiently many R_i Eve is able to reduce the size of the interval that must contain the unknown message M
  - Essentially, each "good" ciphertext halves the interval in question, so that with enough "good" ciphertexts Eve is able to determine M
  - With PKCS #1 Version 1.5 (as used originally in SSL V.3.0) roughly one in 2^{16} to 2^{18} randomly chosen ciphertexts will be "good"
  - Typically, for a 1024-bit modulus, the total number of required ciphertexts is about 2^{20}, and this is also the number of queries to Bob
  - Thus, after performing about 1 million bogus SSL handshake dialogues (which are all disrupted by either Bob or Eve), Eve is able to recover the pre-master-secret and all derived keys of a previously established SSL session between Alice and Bob

→ *Subtle protocol interactions (here: SSL and PKCS #1) can lead to failure of a security protocol, even if the basic cryptographic algorithm (here: RSA) itself is not broken!*

SSL Handshake Protocol: A Vulnerability (4)

- Countermeasures:
  - Regularly changing the public key pairs (⇒ overhead)
  - Reducing the probability of getting "good" ciphertexts by thoroughly checking the format of decrypted ciphertexts and showing identical behaviour (error message, timing behaviour, etc.) to the client
  - Requiring the client to show knowledge of the plaintext before responding if the message could be successfully decrypted
  - Adding structure to the plaintext, e.g. by adding a hash value to the plaintext:
    - Attention: some care needs to be taken in order to avoid vulnerabilities to a different class of attacks [Cop96a]
  - Changing the public key encryption protocol, that is revising PKCS #1:
    - PKCS #1 Version 2.1 prepares the plaintext prior to encryption with a method called *optimal asymmetric encryption padding (OAEP)* in order to make the PKCS #1 decryption procedure "plaintext aware" which implies that it is not possible to construct a valid ciphertext without knowing the corresponding plaintext

SSL Cipher-Suites (1)

- No protection (default suite):

CipherSuite SSL_NULL_WITH_NULL_NULL	= { 0x00,0x00 }

- Server provides an RSA key suitable for encryption:

SSL_RSA_WITH_NULL_MD5	= { 0x00,0x01 }
SSL_RSA_WITH_NULL_SHA	= { 0x00,0x02 }
SSL_RSA_EXPORT_WITH_RC4_40_MD5	= { 0x00,0x03 }
SSL_RSA_WITH_RC4_128_MD5	= { 0x00,0x04 }
SSL_RSA_WITH_RC4_128_SHA	= { 0x00,0x05 }
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	= { 0x00,0x06 }
SSL_RSA_WITH_IDEA_CBC_SHA	= { 0x00,0x07 }
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	= { 0x00,0x08 }
SSL_RSA_WITH_DES_CBC_SHA	= { 0x00,0x09 }
SSL_RSA_WITH_3DES_EDE_CBC_SHA	= { 0x00,0x0A }

SSL Cipher-Suites (2)

- Cipher-Suites with an authenticated DH-Key-Exchange

SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	= { 0x00,0x0B }
SSL_DH_DSS_WITH_DES_CBC_SHA	= { 0x00,0x0C }
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	= { 0x00,0x0D }
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	= { 0x00,0x0E }
SSL_DH_RSA_WITH_DES_CBC_SHA	= { 0x00,0x0F }
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	= { 0x00,0x10 }
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	= { 0x00,0x11 }
SSL_DHE_DSS_WITH_DES_CBC_SHA	= { 0x00,0x12 }
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	= { 0x00,0x13 }
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	= { 0x00,0x14 }
SSL_DHE_RSA_WITH_DES_CBC_SHA	= { 0x00,0x15 }
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	= { 0x00,0x16 }

(DH stands for suites in which the public DH values are contained in a certificate signed by a CA, DHE for suites in which they are signed with a public key which is certified by a CA)

SSL Cipher-Suites (3)

- The use of the following cipher-suites without any entity authentication is strongly discouraged, as they are vulnerable to man-in-the-middle attacks:

SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	= { 0x00,0x17 }
SSL_DH_anon_WITH_RC4_128_MD5	= { 0x00,0x18 }
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	= { 0x00,0x19 }
SSL_DH_anon_WITH_DES_CBC_SHA	= { 0x00,0x1A }
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	= { 0x00,0x1B }

- The final cipher suite is for the Fortezza token:

SSL_FORTEZZA_DMS_WITH_NULL_SHA	= { 0x00,0x1C }
SSL_FORTEZZA_DMS_WITH_FORTEZZA_CBC_SHA	= { 0x00,0x1D }

(These cipher-suites, of course, do not need to be memorized and are listed here only to illustrate the flexibility of the SSL protocol)

The Transport Layer Security Protocol (1)

- 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 (2)

- In order to achieve exportability of TLS compliant products, some cipher-suites used to specify the use of keys with entropy reduced to 40 bit
  - The use of these cipher-suites is strongly discouraged, as they offer virtually no data confidentiality protection
- As of TLS 1.2 (RFC 5246):
  - Key exchange algorithms:
    - DH or ECDH exchange without or with DSS / RSA / ECDSA signatures
    - DH exchange with certified public DH parameters
    - RSA based key exchange
    - none
  - Encryption algorithms: AES / 3DES in CBC / CCM /GCM, RC4, null
  - Hash algorithms: MD5, SHA-1, SHA-256, SHA-384, SHA-512, null
  - Premaster Secret: No MD5/SHA-1 combination, but SHA-256 only!
- Concerning its protocol functions, TLS is essentially the same like SSL

The Transport Layer Security Protocol (3)

- Security:
  - In SSL 3.0 and TLS 1.0 the initialization vector of a record encrypted in CBC mode is the last block of the previous record
  - If an attacker controls the content of the previous record, he may perform an adaptive chosen plaintext attack to find out the content of the next record
  - Feasible for web traffic, i.e. generate traffic with JavaScript and observe from the outside, leads to so-called BEAST (Browser Exploit Against SSL/TLS) attack [RD10]
  - Also feasible for VPN traffic
  - Mitigated by TLS 1.1, where explicit IVs are used

The Transport Layer Security Protocol (4)

- Security (cont.):
  - In 2009 a so-called *TLS Renegotiation Vulnerability* was identified
  - Attackers may use it to prepend data to a legitimate session by a man-in-the-middle attack (details in [Zo11])
  - The impact heavily depends on the used application protocol
  - In HTTPS this will lead to multiple exploit possibilities, e.g.,
    - Attacker injects:
      ```
      GET /ebanking/transfer?what=LotsOfMoney&to=eve HTTP/1.1 <crlf>
      X-Ignore: <no crlf>
      ```
    - Alice sends:
      ```
      GET /ebanking/start.html HTTP/1.1
      ```
    - The request will become a valid HTTP request:
      ```
      GET /ebanking/transfer?what=LotsOfMoney&to=eve HTTP/1.1 <crlf>
      X-Ignore: GET /ebanking/start.html HTTP/1.1
      ```
  - Mitigated by identifying renegotiated sessions with a different ID [RRDO10]

The Datagram Transport Layer Security Protocol

- TLS provides secure communication over a reliable transport protocol
- DTLS is adapted to work over unreliable transport protocols e.g. UDP
- Used to protect:
  - Real-time speech and video data especially Voice-over-IP
  - Tunneled TCP-data (as TCP over TCP is a bad idea for performance)

- DTLS is currently based on TLS 1.2, but contains several changes:
  - Provides
    - Message retransmits to counter lost handshake packets
    - Own fragmentation mechanism to allow for large handshake packets
  - Adds sequence numbers to allow reordered data packets (and prohibits stream ciphers i.e. RC4)
  - Adds mechanism to detect that a client restarted “connection” with the same ports (e.g. after application crash)
  - Adds a replay protection by a sliding window (same as in IPsec)
  - Adds cookie-based DoS-defense (same as in IKEv2)

The Secure Shell 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 several separate documents, e.g.,:
  - SSH Protocol Assigned Numbers [LL06]
  - SSH Protocol Architecture [YL06a]
  - SSH Authentication Protocol [YL06b]
  - SSH Transport Layer Protocol [YL06c]
  - SSH Connection Protocol [YL06d]
- 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

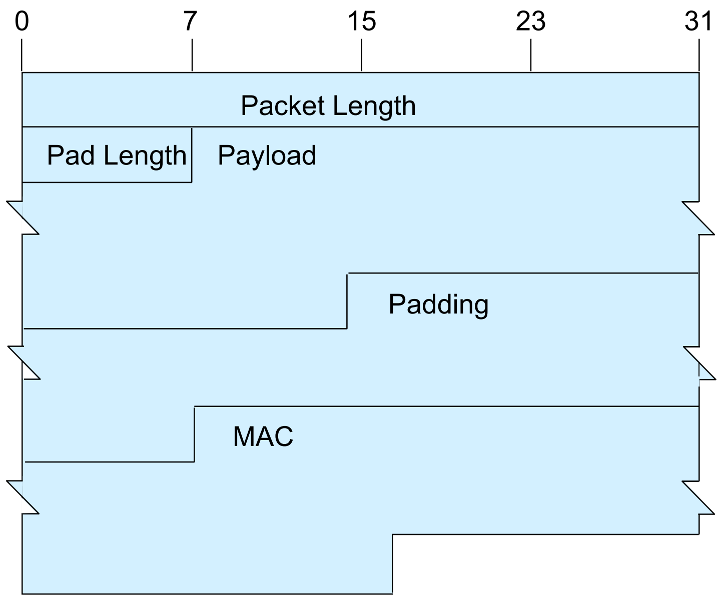
SSH Transport Protocol (I)

- SSH uses a reliable transport protocol (usually 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

SSH Transport Protocol (II)

- Supported algorithms:
  - Encryption:
    - AES, 3DES, Blowfish, Twofish, Serpent, IDEA and CAST in CBC
    - AES in GCM [IS09]
    - Arcfour (“believed” to be compatible with the “unpublished” RC4)
    - none (not recommended)
  - Integrity:
    - HMAC with MD5, SHA-1, SHA-256 or SHA-512
    - none (not recommended)
  - Key exchange:
    - Diffie-Hellman with SHA-1 and two pre-defined groups
    - ECDH with multiple pre-defined NIST groups [SG09] (mandatory three curves over $\mathbb{Z}_p$)
    - Public key: RSA, DSS, ECC (in multiple variants [SG09])
  - Compression: none, zlib (see RFCs 1950, 1951)

SSH Transport Protocol Packet Format (1)



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

SSH Transport Protocol Packet Format (2)

- 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 (1)

- 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 [YL06c] defines only one method:
  - Diffie-Hellman with SHA-1 and two predefined groups (1024 and 2048 bit)
  - E.g. $p = 2^{1024} - 2^{960} - 1 + (2^{64} \times \lfloor 2^{894} \times \pi + 129093 \rfloor)$; $g = 2$

SSH Negotiation, Key Exchange & Server Authentication (2)

- 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 $h = Hash(version_C, version_S, kexinit_C, kexinit_S, +K_S, e, f, K)$ 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 derived the SSH session keys:
  - The hash *h* of the initial key exchange is also taken as the *session_id*
  - $IV_{Client2Server} = Hash(K, h, \text{“A”}, session\_id)$ // initialization vector
  - $IV_{Server2Client} = Hash(K, h, \text{“B”}, session\_id)$ // initialization vector
  - $EK_{Client2Server} = Hash(K, h, \text{“C”}, session\_id)$ // encryption key
  - $EK_{Server2Client} = Hash(K, h, \text{“D”}, session\_id)$ // encryption key
  - $IK_{Client2Server} = Hash(K, h, \text{“E”}, session\_id)$ // integrity key
  - $IK_{Server2Client} = Hash(K, h, \text{“F”}, session\_id)$ // 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”, etc.
  - $K2 = Hash(K, h, K1)$
  - $K2 = Hash(K, h, K1, K2)$
  - $XK = K1 \| K2 \| \ldots$

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
    Client → Server: $E(-K_{User}, (session\_id, 50, Name_{User}, Service, \text{“publickey”}, True, PublicKeyAlgorithmName, +K_{User}))$
  - *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 (1)

- 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

SSH Connection Protocol (2)

- Opening a channel:
  - Either side may send the message *ssh_msg_channel_open* signaled with message code 90 and the following parameters:
    - *channel type:* is of data type string, e.g. *"session", "x11",* etc.
    - *sender channel:* is a local identifier of type uint32 and chosen by the requestor of this channel
    - *initial window size:* is of type uint32 and specifies how many bytes may be send to the initiator before the window needs to be advanced
    - *maximum packet size:* is of type uint32 and defines the maximum packet size the initiator is willing to accept for this channel
    - further parameters depending on the type of the channel may follow
  - If receiver of this message does not want to accept the channel request, it answers with the message *ssh_msg_channel_open_failure* (code 92):
    - *recipient channel:* the id given in the open request by the sender
    - *reason code:* is of type uint32 and signals the reason for the rejection
    - *additional textual information:* is of type string
    - *language tag:* is of type string and according to RFC 1766

SSH Connection Protocol (3)

- Opening a channel (cont.):
  - If receiver of this message wants to accept the channel request it answers with the message *ssh_msg_channel_open_confirmation* (code 91) and parameters:
    - *recipient channel:* the id given in the open request by the sender
    - *sender channel:* the id given to the channel by the responder
    - *initial window size:* is of type uint32 and specifies how many bytes may be send to the responder before the window needs to be advanced
    - *maximum packet size:* is of type uint32 and defines the maximum packet size the responder is willing to accept for this channel
    - further parameters depending on the channel type may follow
- Once a channel is opened, the following actions are possible:
  - Data transfer (however, the receiving side should know "what to do with the data" which may require further prior negotiation)
  - Channel type specific requests
  - Closure of the channel

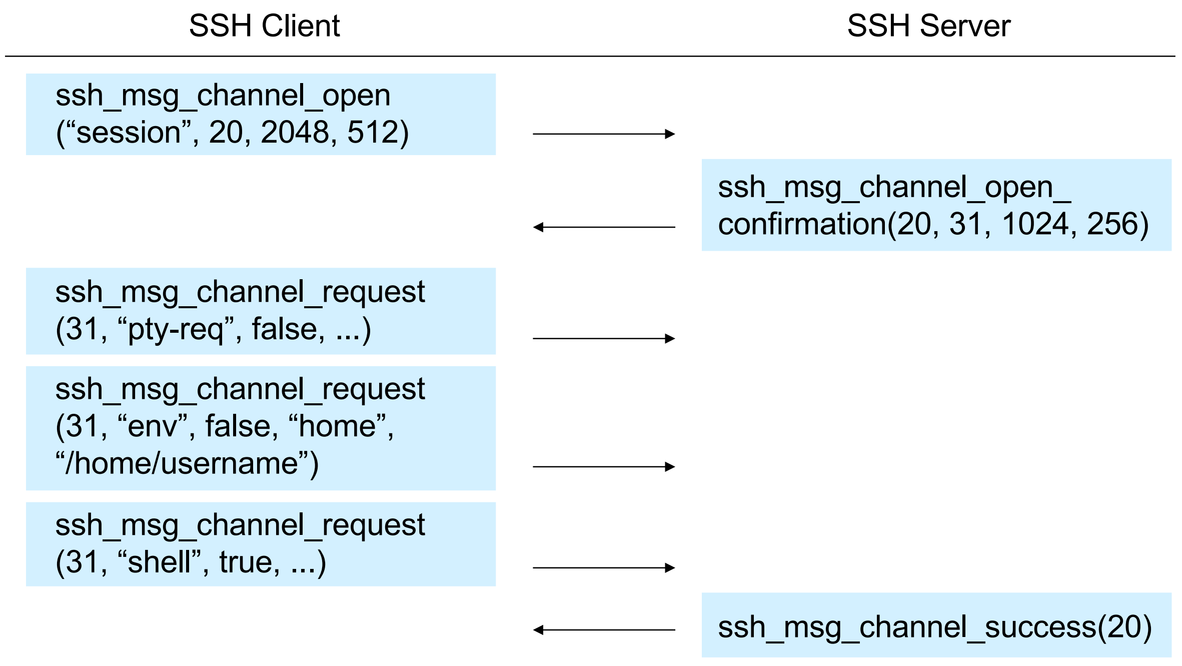
SSH Connection Protocol (4)

- For data transfer the following messages are defined:
  - *ssh_msg_channel_data:* with the two parameters *recipient channel, data*
  - *ssh_msg_channel_extended_data:* allows to additionally specify a *data type code* and is useful to signal errors, e.g. of interactive shells
  - *ssh_msg_channel_window_adjust:* allows to advance the flow control window of the *recipient channel* by the specified number of *bytes to add*
- Closing of channels:
  - When a peer entity will not longer send data to a channel it should signal this to the other side with the message *ssh_msg_channel_eof*
  - When either side wishes to terminate a channel it sends the message *ssh_msg_channel_close* with parameter *recipient channel*
  - Upon reception of the message *ssh_msg_channel_close* a peer entity must answer with a similar message unless it has already requested closure of this channel
  - After both receiving and sending of the *ssh_msg_channel_close* message for a specific channel, the *id* of that channel may be re-used

SSH Connection Protocol (5)

- Channel type specific requests allow to demand for specific properties of a channel, e.g. such that the receiving side knows how to process data send via this channel, and are signaled with:
  - *ssh_msg_channel_request:* with the parameters *recipient channel, request type* (string), *want reply* (bool) and further request specific parameters
  - *ssh_msg_channel_success:* with the parameter *recipient channel*
  - *ssh_msg_channel_failure:* with the parameter *recipient channel*
- Example 1 – requesting an interactive session and starting a shell in it:
  - First, a channel of type *"session"* is opened
  - A pseudo-terminal is requested by sending an *ssh_msg_channel_request* message with the request type set to *"pty-req"*
  - If needed, environment variables can be set by issuing *ssh_msg_channel_request* messages with request type set to *"env"*
  - Then, the start of a shell process is demanded via an *ssh_msg_channel_request* message with the request type set to *"shell"* (usually this results in the start of the default shell for the user as defined in /etc/passwd)

Requesting an Interactive Session and Starting a Shell in it



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...]

SSH Connection Protocol (6)

- Example 2 – requesting X11 forwarding:
  - First, a channel of type *"session"* is opened
  - X11 forwarding is requested by sending an *ssh_msg_channel_request* message with request type set to "x11-req"
  - If later on an application is started on the server that needs to access the terminal of the client machine (the X11-server running on the client machine), a new channel is opened via *ssh_msg_channel_open* with the channel type set to *"x11"* and the originator IP address and port number as additional parameters

SSH Connection Protocol (7)

- Example 3 – setting up TCP/IP port forwarding:
  - A party needs not to explicitly request port forwarding from its own end to the other direction, however, if it wants to have connections to a port on the other side forwarded to its own side, it must explicitly request this via an ssh_msg_global_request message with the parameters *"tcpip-forward", want-reply, address to bind* ("0.0.0.0" for every source address), and *port number to bind* (this request is usually sent by the client)
  - When a connection comes for a port for which forwarding has been requested, a new channel is opened via *ssh_msg_channel_open* with the type set to *"forwarded-tcpip"* and the addresses of the port that was connected as well as of the original source port as parameters (this message is usually sent by the server)
  - When a connection comes to a (client) port that is locally set to be forwarded, a new channel is requested with the type set to *"direct-tcpip"* and the following address information specified in additional parameters:
    - *host to connect, port to connect:* address to which the recipient should connect this channel
    - *originator IP address, originator port:* source address of the connection

Conclusion

- Both SSL, TLS and SSH are suited to secure Internet communications in (above) the transport layer:
  - All three security protocols operate upon and require a reliable transport service, e.g. TCP
  - There is a datagram oriented variant of TLS, called DTLS
  - Even though SSH operates in / above the transport layer the server authentication is host-based and not application-based
  - Transport layer security protocols offer true end-to-end protection for user data exchanged between application processes
  - Furthermore, they may interwork with *packet filtering* of today's firewalls
  - But, protocol header fields of lower layer protocols can not be protected this way, so they offer no countermeasures to threats to the network infrastructure itself

Additional References

[BKS98a] D. Bleichenbacher, B. Kaliski, J. Staddon. *Recent Results on PKCS #1: RSA Encryption Standard.* RSA Laboratories' Bulletin 7, 1998.

[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.

[DA99] T. Dierks, C. Allen. *The TLS Protocol Version 1.0.* RFC 2246, 1999.

[DR08] T. Dierks, E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2.* RFC 5246, 2008.

[RD10] J. Rizzo, T. Duong, Practical Padding Oracle Attacks, 4th USENIX conference on Offensive technologies (WOOT), 2010.

[RRDO10] E. Rescorla, M. Ray, S. Dispensa, N. Oskov. *Transport Layer Security (TLS) Renegotiation Indication Extension*, RFC 5746. 2010

[Zo11] T. Zoller. *TLS & SSLv3 renegotiation vulnerability.* Technical report, G-SEC. 2011

[RM12] E. Rescorla, N. Modadugu. *Datagram Transport Layer Security Version 1.2.* RFC 6347, 2012.

Additional References

[LL06] S. Lehtinen, C. Lonvick. *The Secure Shell (SSH) Protocol Assigned Numbers.* RFC 4250, 2006.

[YL06a] T. Ylonen, C. Lonvick. *The Secure Shell (SSH) Protocol Architecture.* RFC 4251, 2006.

[YL06b] T. Ylonen, C. Lonvick. *The Secure Shell (SSH) Authentication Protocol.* RFC 4252, 2006.

[YL06c] T. Ylonen, C. Lonvick. *The Secure Shell (SSH) Transport Layer Protocol*, RFC 4253, 2006.

[YL06d] T. Ylonen, C. Lonvick. *The Secure Shell (SSH) Connection Protocol.* RFC 4254, 2006.

[SG09] D. Stebila, J. Green. *Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer*, RFC 5656. 2009

[IS09] K. Igoe, J. Solinas. *AES Galois Counter Mode for the Secure Shell Transport Layer Protocol.* RFC 5647. 2009