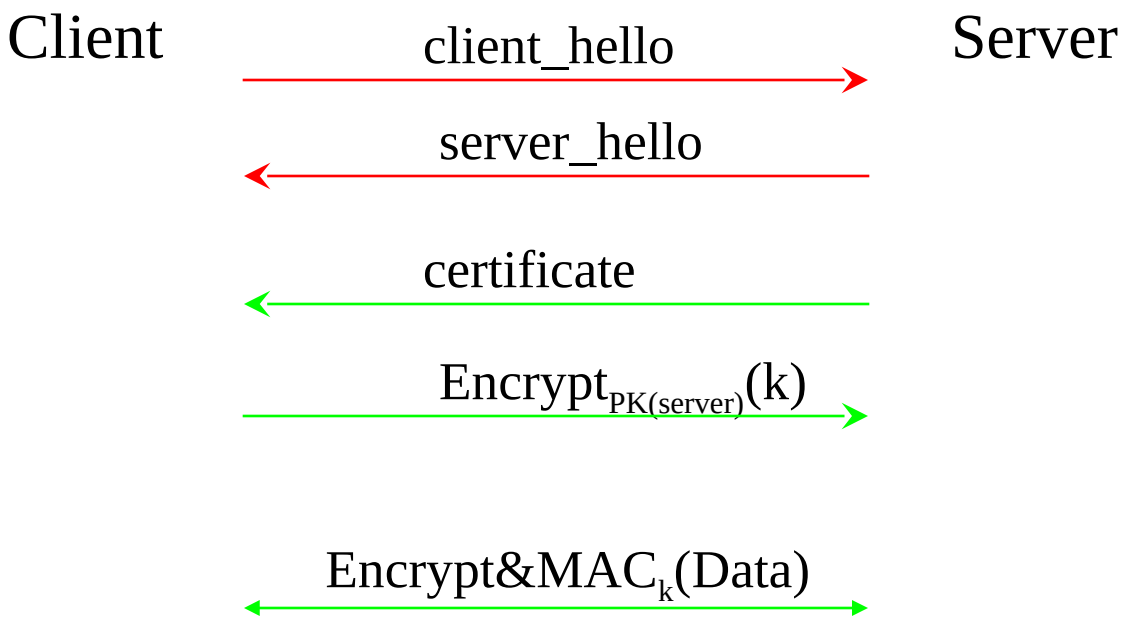
Secure Socket Layer
&
Transport Layer Security

Dipartimento di Informatica

Università degli Studi di Torino

«simplified» SSL



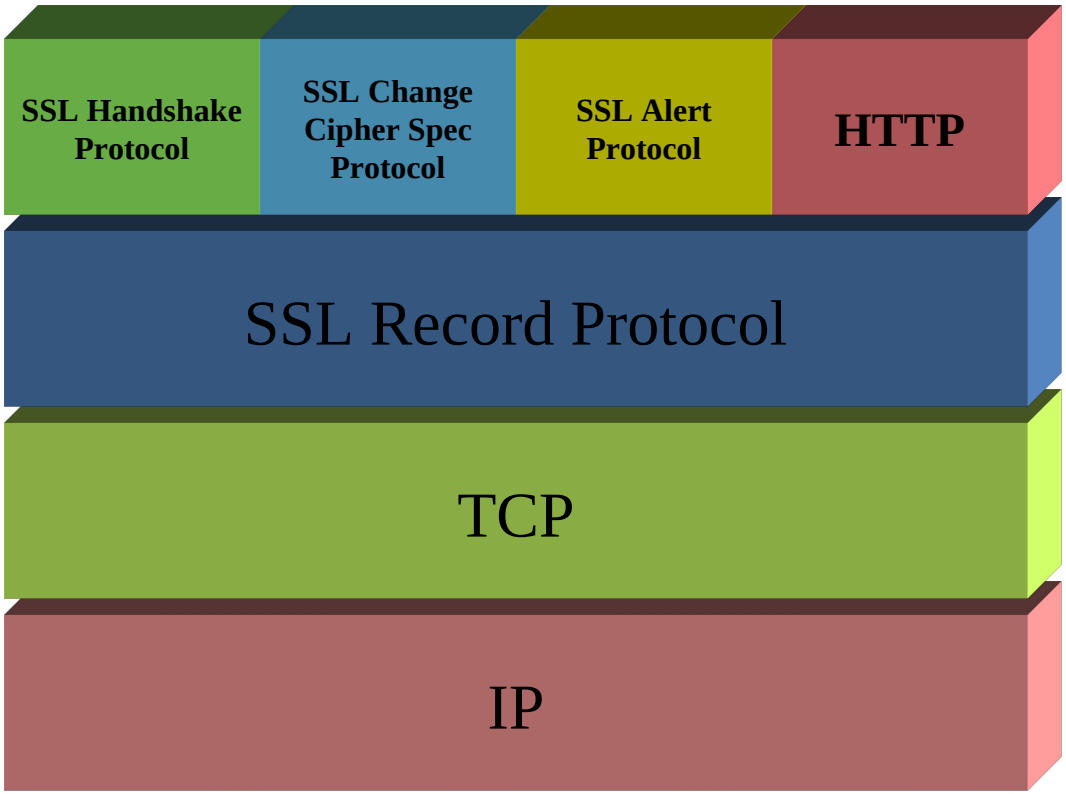
Secure Socket Layer & Transport Layer Security

- SSL was initially proposed by Netscape
- Initially an Internet Draft
- Now an IETF Internet Standard (RFCs defining SSLv3 and TLS)

SSL architecture

- Based on TCP transport
- Two protocol levels
- The SSL Record Protocol provides a security layer to upper-level protocols:
  - E.g. HTTP, POP3, IMAP, LDAP, SMTP
  - 3 protocols that are part of the SSL suite:
    - Handshake
    - Change Cipher Spec
    - Alert

SSL architecture



Connections and Sessions

- **Connection**: based on a connection-oriented protocol, where peer devices communicate in a reliable way (e.g., with TCP). Every connection is associated to one session.
- **Session**: association between two peers. Defines a set of encryption and authentication parameters that are used by one or more related connections, as part of a higher level set of communication

Connections and Sessions

- Two peers (e.g., a Browser and a Web Server) may simultaneously run more than one secure connection.
- Two simultaneous sessions, are also possible, in principle.

Session state

- Session identifier: id of session state
- Peer certificate: certificate X.509v3 of the peer entity (may be empty)
- Compression method
- Cipher spec: encryption and hash algorithms, as well as the hash size
- Master secret: 48 secret bytes, shared by the two peers
- Is resumable: set to *true* if it is possible to start new connections within this session

Connection state

- Server & Client Random: random bytes used within this connection
- Server write MAC secret: secret key used for computing MACs sent by the server
- Client write MAC secret: secret key used for computing MACs sent by the client
- Server write key: encryption key for data sent by server
- Client write key: encryption key for data sent by client
- Initializaton vectors: used for CBC encryption
- Sequence numbers: must be < 2^{64}-1; set to 0 after change cypher spec

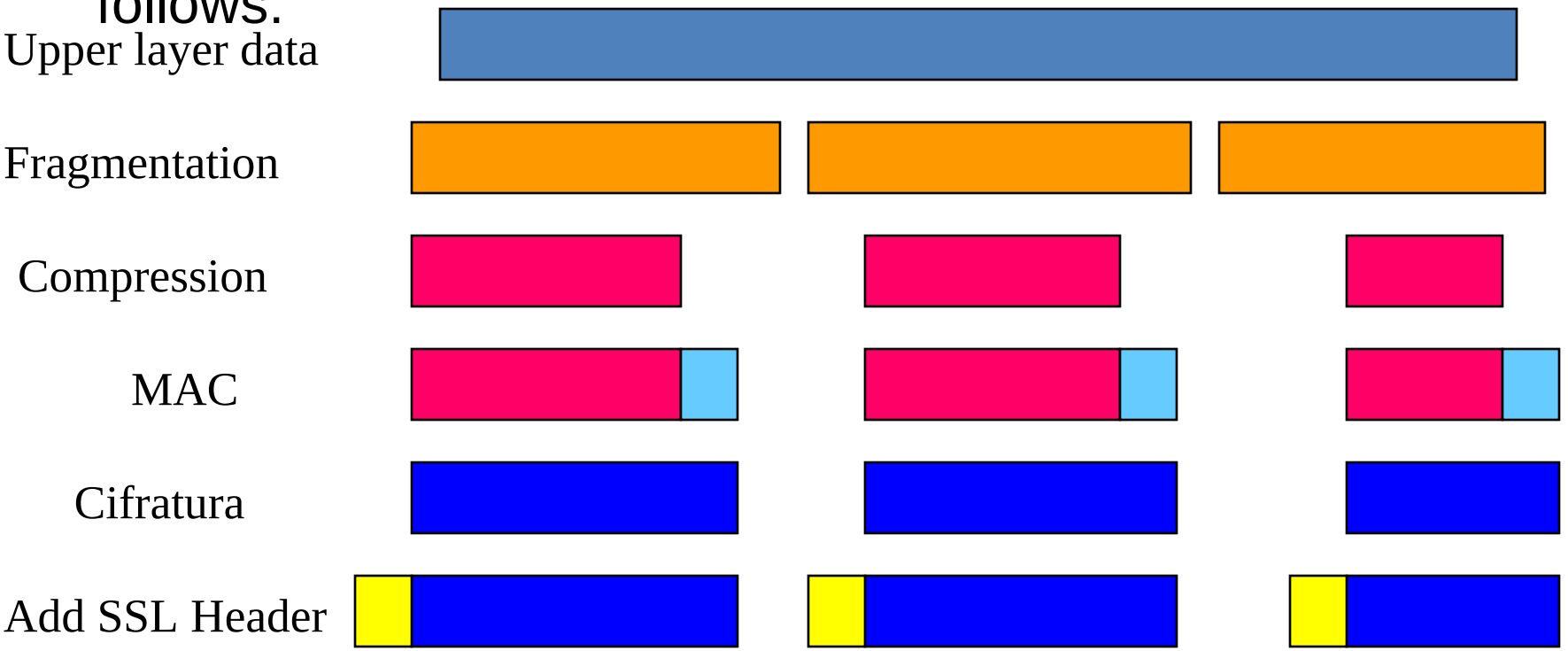
Session state

- The Handshake Protocol creates a “pending” state, that becomes the “current” state after the handshake is complete.

SSL Record Protocol

Provides confidnetiality and integrity, based on two keys that are shared during the handshake.

Data from the upper layer protocol are process as follows:



SSL Record Protocol

- Fragmntation: blocks <= 2^{14} bytes
- Compression: optional, lossless
- MAC = *hash*(MAC _write_secret |
  pad2 | *hash*(MAC _write_secret | pad1 |
  Seq_num | Type| Length| Fragment))

  pad1: 0x36 $\times$ 48 (MD5) or $\times$ 40 (SHA-1)

  pad2: 0x5C $\times$ 48 (MD5) or $\times$ 40 (SHA-1)

  Type: higher level protocol that has produced the *Fragment* of size *Length* (after compression)

SSL Record Protocol

- Encryption: also covers the MAC
  - Block ciphers: IDEA (128), RC2-40 (40), DES-40 (40), DES (56), 3DES (168), Fortezza (80) – *padding may be necessary.*
  - Stream ciphers: RC4-40 (40), RC4-128 (128)

$\Rightarrow$ Max length canno exceed $2^{14}+2048$

SSL Record Protocol

SSL Header :

Content Type	Major Version (3)	Minor Version (0)	Compressed Length $\leq 2^{14}+2048$

Content Type:

- change_cipher_spec
- alert
- handshake
- application protocol

Change Cipher Spec Protocol

- One of the three SSL protocols
- Just one message with one byte, that must be equal to 1 (00000001)
- Pending state => current state

Alert Protocol

- Alert messages are sent over the Record protocol (with possible compression and encryption)
- 2 bytes
  - 1 byte for the alert level (warning [1] or fatal [2])
  - 1 byte for the code
- The fatal level alerts close the connection and prevent further connections from being opened within the same session.

Alert Protocol

- Examples of fatal alerts:
  - unexpected_message
  - bad_record_mac: MAC is wrong
  - decompression_failure
  - handshake_failure: security parameters uncacceptable
- Other errors:
  - certificate_expired
  - certificate_revoked
  - close_notify: sender does not intend to write any further

Peer authentication

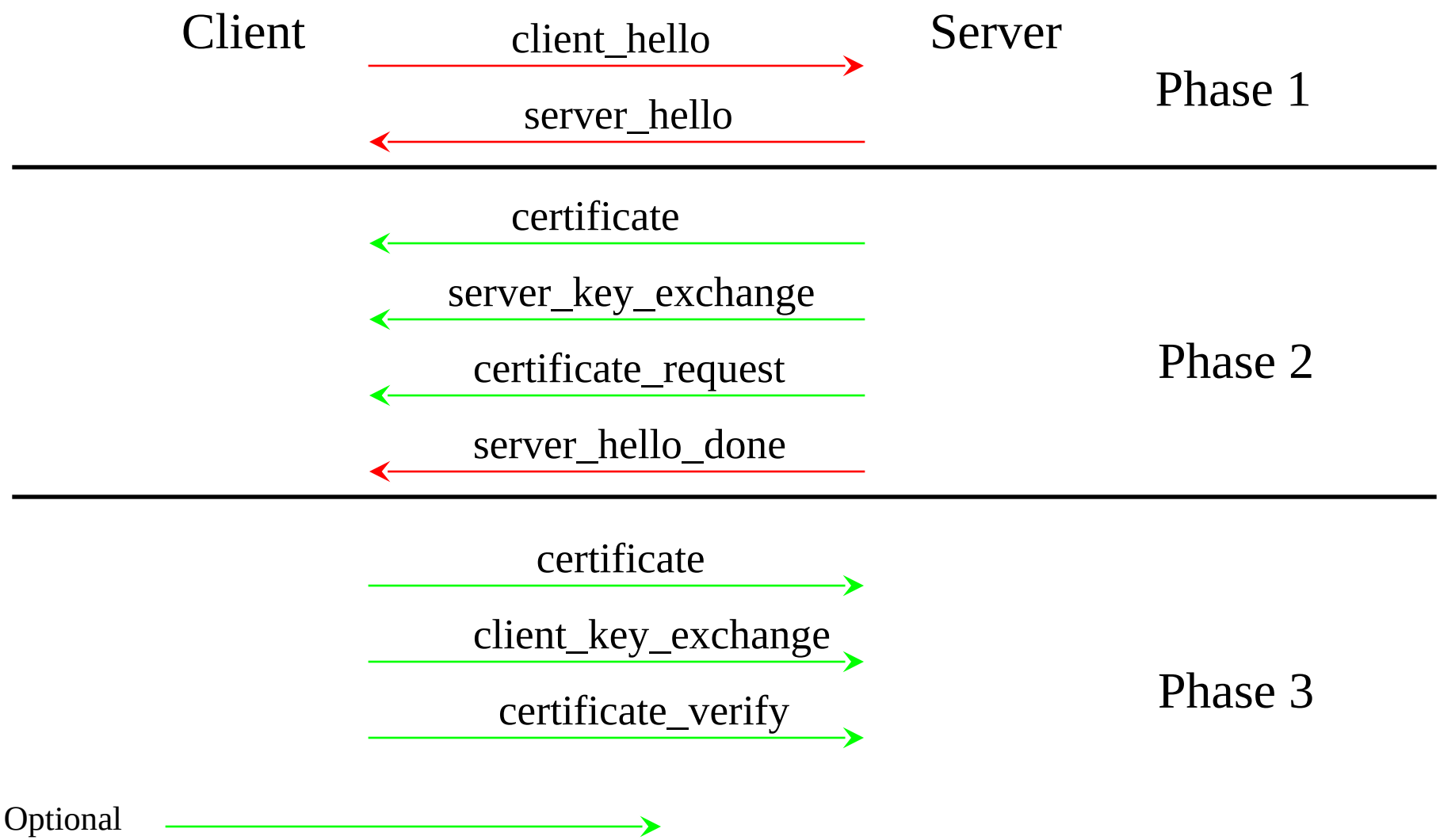
- Server & Client authentication
- Server authentication only
- No authentication (man-in-the-middle attacks are possible)

Handshake Protocol

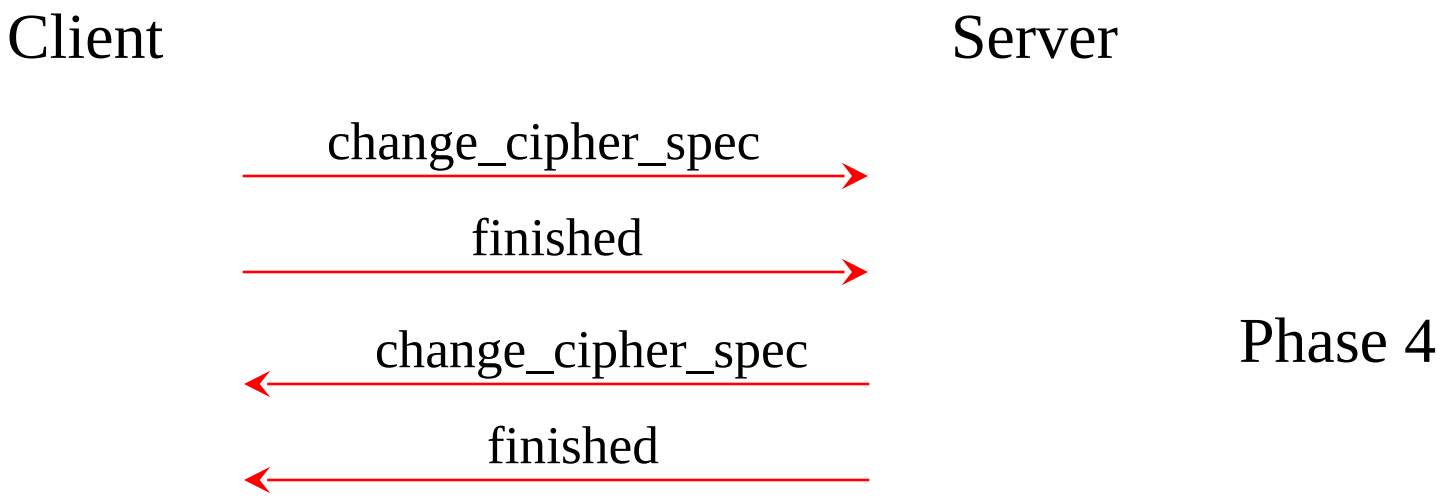
- The handshake protocol provides:
  - possible server and client authentication
  - encryption, hash, and compression algorithm negotiation
  - key exchange mechanisms
- Message format:

Type (1)	Length (3)	Content (≥1)

Handshake Protocol



Handshake Protocol



Handshake Protocol: Phase 1

- Creates a logical connection and associates security parameters & services
- Client initiates the handshake by sending a *client_hello* message with these parameters:
  - Version (highest supported)
  - Random (28 byte random + 4 byte antireplay timestamp)
  - Session Id ($\neq$0 update or new connection, =0 new session and connection)
  - Compression Method (client-supported algorithms)
  - Cipher Suite (security parameters)

Handshake Protocol: Phase 1

- client_hello may also be used to re-negotiate the security parameters
- server may request a client_hello by sending a *hello_request* parameterless message
- *server_hello* completes the parameter negotiation for the session's security parameters: version (lowest between the one proposed by client and the highest supported by server), Cipher Suite, Compression algorithm.
  - Session Id is new if client had sent a 0
  - Random is a new value (28+4 bytes)

Cipher Suite

- Contains two information sets:
  - *Key Exchange Method*: specifies the key exchange method
  - *CipherSpec*: specifies the encryption and message authentication algorithms that will be used

Key Exchange Method

- **RSA**: secret session key is encrypted with the peer's public key
- **Fixed Diffie-Hellman**: the server certificate contains the DH public parameters; the client's DH public parameters are sent in a certificate (if client authentication is required), or in the following key exchange message (If client authentication is not required)

Key Exchange Method

- **Ephemeral Diffie-Hellman**: Diffie-Hellman public keys are signed with the sender's private key K_S (RSA or DSS); the receiver verifies authenticity using the corresponding public key K_P (contained in the sender's certificate)
- **Anonymous Diffie-Hellman**: DH public parameters are sent without authentication (exposed to man-in-the-middle attacks)

CipherSpec

- **Encryption algorithm**
- **MAC algorithm**
- **stream or block cipher choice**
- **Hash size**
- **Initialization Vector size for CBC encryption**

Signature method

- Signatures are based on the following hash value:

hash(client_hello.random | server_hello.random | Data_to_be_signed)

so as to avoid replay attacks

- With DSS hash=SHA-1
- With RSA, hash=$hash_{MD5}$ | $hash_{SHA-1}$

Handshake Protocol: Phase 2

- Server starts by sending its certificate within an X.509 certificate chain (except for anonymous DH)
- If the certificate contains fixed DH parameters, this message also accomplishes the server key exchange
- If RSA key exchange is used, there is no need for a subsequent server_key_exchange message

Handshake Protocol: Phase 2

- A server_key_exchange message is sent for:
  - Anonymous Diffie-Hellman: contains the common parameters (q,α) and the server's public key PU_s
  - Ephemeral Diffie-Hellman: contains the common parameters (q,alfa) and the server's public key PU_s, together with the signature of the same values
  - RSA key exchange: the RSA key included in the certificate may be used for signatures only, and a new RSA key pair is generated to be used for encrypting the pre_master_secret. The public component of this new key pair is signed with the RSA key corresponding to the primary RSA key contained in the certificate, and this signed public key is sent in the server_key_exchange message.

Handshake Protocol: Phase 2

- A non-anonymous server may request a client certificate with a certificate_request message, containing two parameters:
  - Certificate type: signature only RSA (or DSS), RSA (or DSS) for fixed Diffie-Hellman, RSA (or DSS) for ephemeral Diffie-Hellman
  - CA: DN of acceptable CAs
- The server concludes phase 2 with a server_done message

Handshake Protocol: Phase 3

- After the *server_done* message, the client verifies the certificate and the server_hello parameters
- If the server has requested a certificate:
  - If the client has one, this is sent in a *certificate* message.
  - Otherwise a *no_certificate* alert is sent

Handshake Protocol: Phase 3

- A *client_key_exchange* message is sent, containing, depending on the key exchange type:
  - **RSA**: a *pre_master_secret* encrypted with the server's public key
  - **Ephemeral or anonymous Diffie-Hellman**: the client's public parameters
  - **Fixed Diffie-Hellman**: nothing, as the parameters were already part of the certificate

Handshake Protocol: Phase 3

- finally, the client may send a *certificate_verify* message, containing a signature of previously sent data and of the **master_secret**.

Handshake Protocol: Phase 4

- The client sends a *change_cipher_spec* message (not part of the handshake protocol). This will make the pending state current.
- The client sends a *finished* message, using the new context (encryption and authentication methods and keys). The *finished* message contains hash values computed on the master secret and on all previously exchanged handshake protocol messages
- The same is done by the server and the session may continue using the new context

Key generation

- The handshake protocol provides end to end authentication, and after this a **pre_master_secret** is shared;
- The **pre_master_secret** was sent as encrypted data with RSA or is based on the DH shared key if Diffie-Hellman is used;
- The 16*3 = 48 byte **master_secret** is computed as a hash of constant strings (‘A’, ‘BB’, ‘CCC’), previously exchanged random data, and the pre_master_secret;
- The **master_secret** is used as a seed for generating further secrets and keys. It is shared by client and server.

Key generation

- The following keys must be generated:
  - Client MAC write
  - Server MAC write
  - Client write secret key
  - Server write secret key
  - IV for client write
  - IV for server write

These values are generated by applying MD5 and SHA-1 on **master_secret**, random values and constants (‘A’, ‘BB’, ‘CCC’, ...).

Key generation

- **master_secret**:

master_secret = MD5(pre_master_secret | SHA('A' | pre_master_secret | client_hello.random | server_hello.random)) |

MD5(pre_master_secret | SHA('BB' | pre_master_secret | client_hello.random | server_hello.random)) |

MD5(pre_master_secret | SHA('CCC' | pre_master_secret | client_hello.random | server_hello.random))

- **other keys** - repeatedly apply the following:

key_sequence = MD5(master_secret | SHA('A' | master_secret | client_hello.random | server_hello.random)) |

MD5(master_secret | SHA('BB' | master_secret | client_hello.random | server_hello.random)) |

MD5(master_secret | SHA('CCC' | master_secret | client_hello.random | server_hello.random)) | .....

Transport Layer Security

- RFC 2246
- Version 3.1
- Uses HMAC (RFC 2104), e.g,:

  $$MAC = HMAC\_MD5_{MAC\ \_write\_secret}(Seq\_num \mid Type \mid Version \mid Length \mid Data\ Fragment)$$

- Uses a different function for key generation

Transport Layer Security

Pseudorandom function

PRF(secret, label, seed):

Based on the following:

A(0) = seed

A(i+1) = HMAC_hf_{secret} (A(i))

P_*hf*(secret,seed) = HMAC_hf_{secret}(A(1) | seed) |

HMAC_hf_{secret}(A(2) | seed) |

HMAC_hf_{secret}(A(3) | seed) | ....

hf = MD5 or SHA-1

PRF(S1 | S2, label, seed) = P_MD5(S1, label| seed) $\oplus$

P_SHA-1(S2, label| seed)

Transport Layer Security

TLS adds some Alert codes: es. (fatal)

- decryption_failed
- record_overflow
- unknown_ca
- insufficient_security
- protocol_version
- internal_error

Transport Layer Security

- CipherSuites: same as SSLv3, without Fortezza
- Client certificates:
  - rsa_sign
  - dss_sign
  - rsa_fixed_dh
  - dss_fixed_dh
- Padding: data must be aligned to block dimension, max 255 bytes

TLS: key generation

- *certificate_verify*: hash computed on handshake messages only (no master_secret)
- *finished*:

  PRF(master_secret, finished_label, MD5(handshake_messages) | SHA-1(handshake_messages))

  finished_label = “client finished” or “server finished”

- **master_secret** = PRF(pre_master_secret, “master secret”, client_hello.random | server_hello.random)
- **key_sequence** = PRF(master_secret, “key expansion”,

  SecurityParameters.server_random | SecurityParameters.client_random)