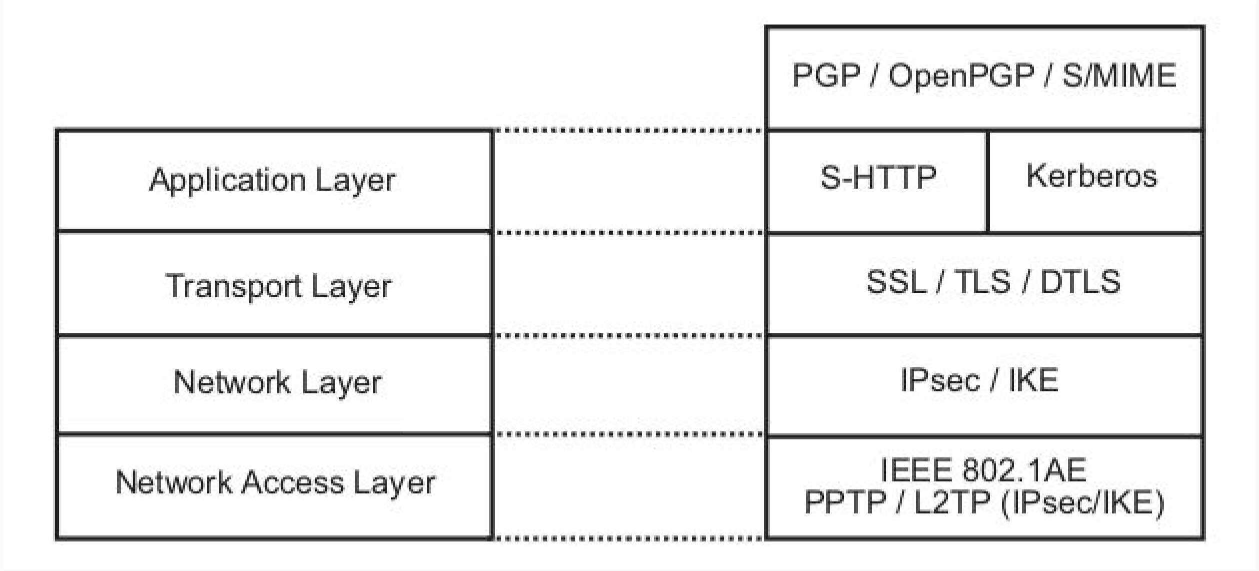
TLS Overview

Illya Gerasymchuk - 78134
Date: 10/10/2017

What Is TLS?

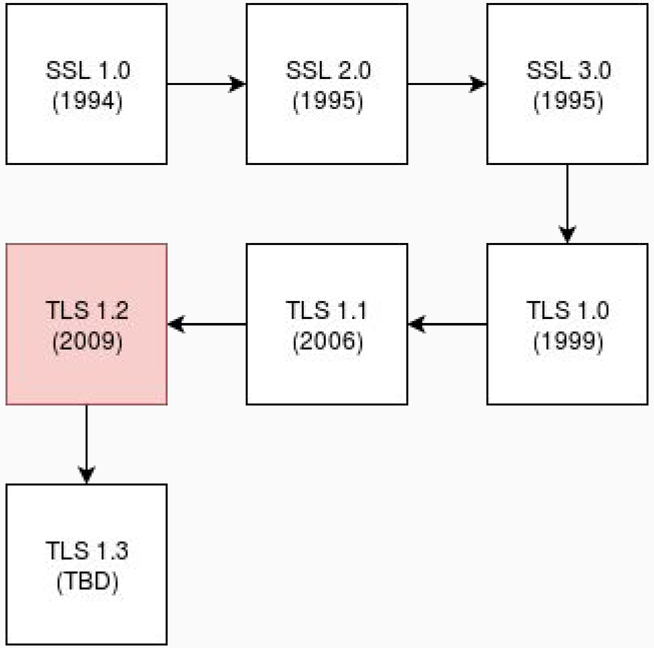
- Transport Layer Security (TSL) [RFC 5246 (TLS 1.2)] ← **Main Focus**
- Client/Server protocol
- Runs on top of a **connection oriented** and **reliable protocol**
  - ex: TCP
  - there is also **DTLS**, which can run on a **best-effort** datagram delivery protocol like **UDP**
- Designed from developer’s POV:
  - make developers life as simple, as possible
  - create a “secure connection” instead of a “normal one”
    - applications can use it **transparently**
  - follows the **end-to-end principle:**
    - all encryption and authentication takes place at endpoints

TLS Placement In The TCP/IP Protocol Stack



Placed between the **Transport Layer** and the **Application Layer**

A Little Bit Of History....



SSL/TLS: Main Functions

- The SSL/TLS protocols have two main functions:
    - establish **secure connections** between the communicating peers
    - **securely** transmit higher-layer protocol data over those connections

Security Services

- From SSL 3.0 and onwards, the following security services are provided:
    - Authentication
        - peer entity authentication
        - data origin authentication
    - Confidentiality
    - Integrity
- Despite using Public Key Cryptography, the following security servers are **NOT** provided:
    - non-repudiation with proof of origin
    - Non-repudiation with proof of delivery

Security Services: Authentication

- **Peer entity authentication** - we can be sure that we're talking to a certain entity (for example, www.google.com)
  - peer's identity is authenticated using asymmetric, or public key, cryptography (RSA, DSA, etc).
- **Data origin authentication** - we can be sure that the **data** that we're receiving is coming from the **expected entity** (for example, we can be sure that the index.html file sent to us when we connected to www.google.com in fact came from www.google.com) and that it was not modified (i.e tampered with) en route by an attacker (**data integrity**).
  - a **keyed MAC** is used (only client and server know the shared key)

Security Services: Confidentiality

- **Confidentiality** - the data transmitted between the communicating entities (the client and the server) is encrypted. **Symmetric cryptography** is used of data encryption (for example, AES, RC4, etc).

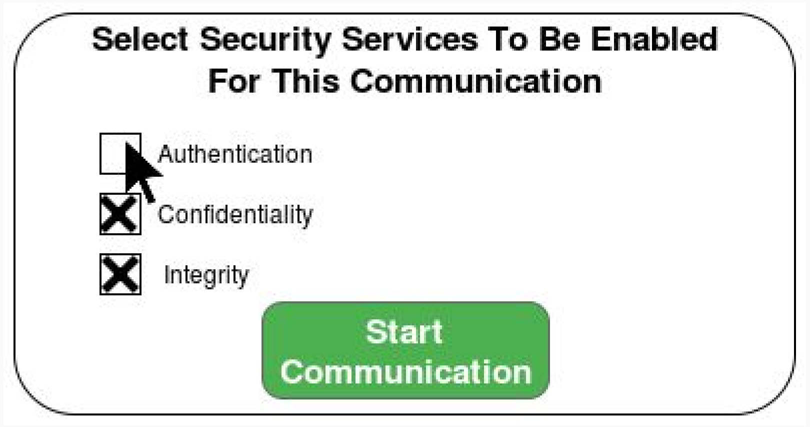
Security Services: Integrity

- **Integrity -** we can be sure that the data was **not modified** or forged.
    - achieved through the use of a **keyed MAC**. Secure hash functions (for example, SHA-1) are used for MAC computations.

Security Services That Are Not Provided

- **Non-repudiation with proof of origin** - addresses the user denying they have sent the message.
- **Non-repudiation with proof of delivery** - addressed the user denying they have received the message.
- Why are they not provided?
    - instead of digital signatures, a **keyed** MAC is used
        - this requires a **shared secret key** between the client and the server
        - due to a **shared key** the **non-repudiation** requirement is dropped

Security Services: You Don't Have To Use Them All


Select Security Services To Be Enabled For This Communication
Authentication
Confidentiality
Integrity
Start Communication

SSL/TLS is like a framework

Keying Material Generation

1. Generate the **premaster secret**
2. From the **premaster secret** generate the **master secret**
3. From the **master secret** generate each one of the items:

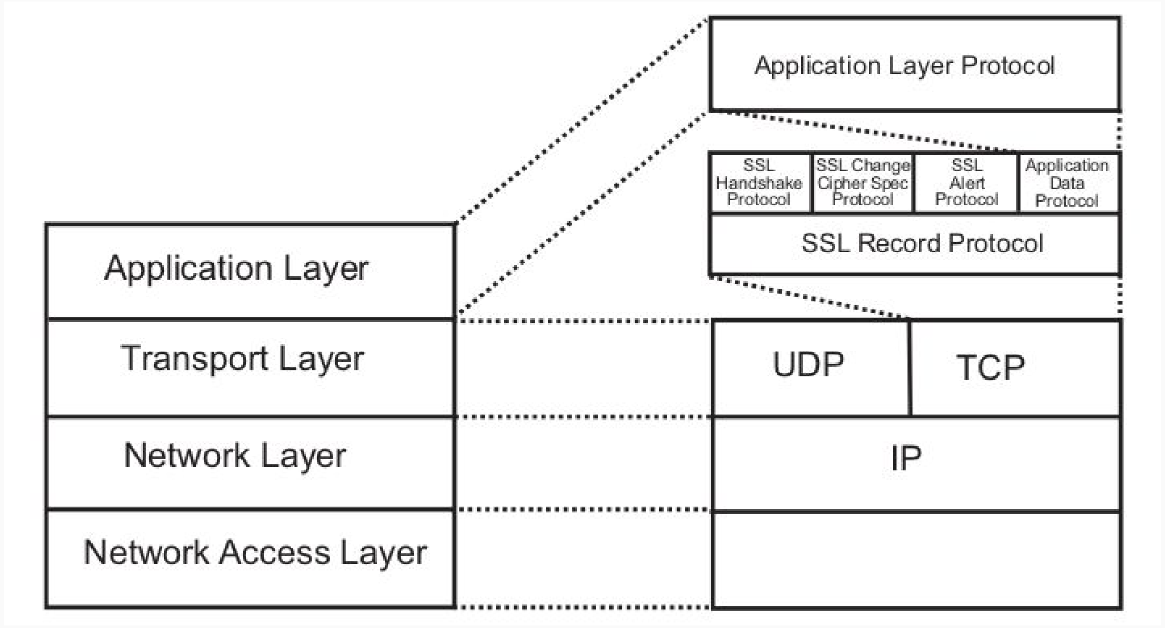
- client write MAC key
- server write MAC key
- client write encryption key
- server write encryption key
- [client write IV]
- [server write IV]
    - client_write_IV and server_write_IV are only generated for implicit nonce techniques used in AEAD Ciphers.

SSL/TLS Subprotocols

In reality, SSL/TLS is not just one protocol, it consists of several sub-protocols:

- **SSL/TLS Record Protocol**
- **SSL/TLS Handshake Protocol**
- **SSL/TLS Change Cipher Spec Protocol** [Removed in TLS 1.3]
- **SSL Alert Protocol**
- **Application Data Protocol**

SSL/TLS Subprotocols

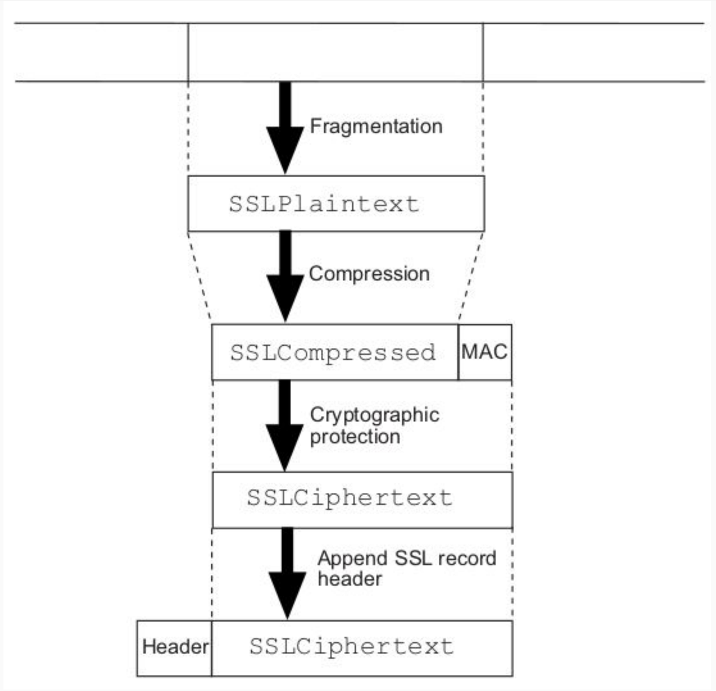


SSL/TLS Record Protocol: Fragmentation

- **SSL/TLS Record Protocol** is used to encapsulate higher-layer protocol data
- SSL/TLS layer **fragments** the data to be sent into manageable pieces
  - those pieces are called **fragments**
- Before being sent to the recipient, each **fragment** is:
  - 1. Compressed (optional)
  - 2. Authenticated with a MAC
  - 3. Encrypted
  - 4. Prepended with a header
- Each one of those fragments is called a **TLS Record**

SSL/TLS Record Processing

-



Cipher Spec vs Cipher Suite

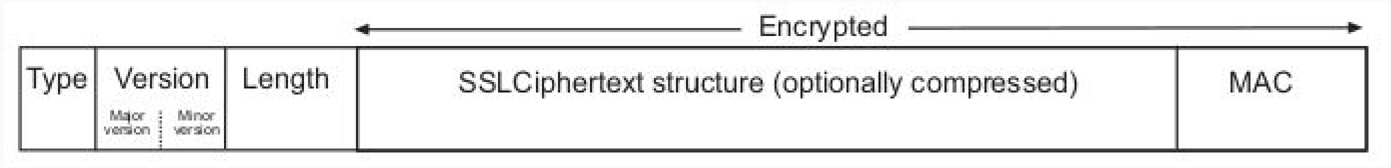
- **Cipher Spec** - message encryption algorithm + message authentication algorithm.
- **Cipher Suite** - **Cipher Spec** + key exchange algorithm.
- Example of a ciphe suite (taken from RFC 5246):
  - TLS_RSA_WITH_AES_128_CBC_SHA256
    - **RSA** used for key exchange
    - **AES in CBC mode with 128 bit keys** used for bulk data encryption
    - **SHA-256** used for message authentication

SSL/TLS Record Protocol

- **Encryption:**
  - Each record is encrypted according to the **cipher spec**
  - Since TLS 1.1 **IVs** (or **nonces**) are sent explicitly with each TLSCipherText
    - For each encrypted fragment, there is a **new, random IV**
    - Response to the BEAST attack (attack on CBC)
- **Integrity: MAC used**
  - TLS Record Layer uses a keyed **MAC** to protect message integrity
  - The cipher suites defened in the RFC 5246 use **HMAC** as their **MAC** function
  - Other cipher suites **may define their own MAC functions,** if needed

$$HMAC(K,m) = H\Big((K' \oplus opad) \,||\, H\big((K' \oplus ipad) \,||\, m\big)\Big)$$

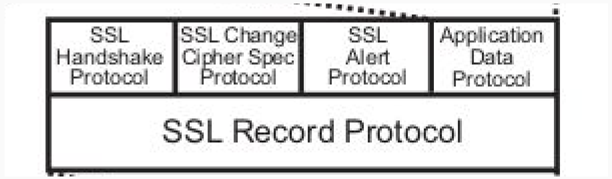
SSL/TLS Record Header



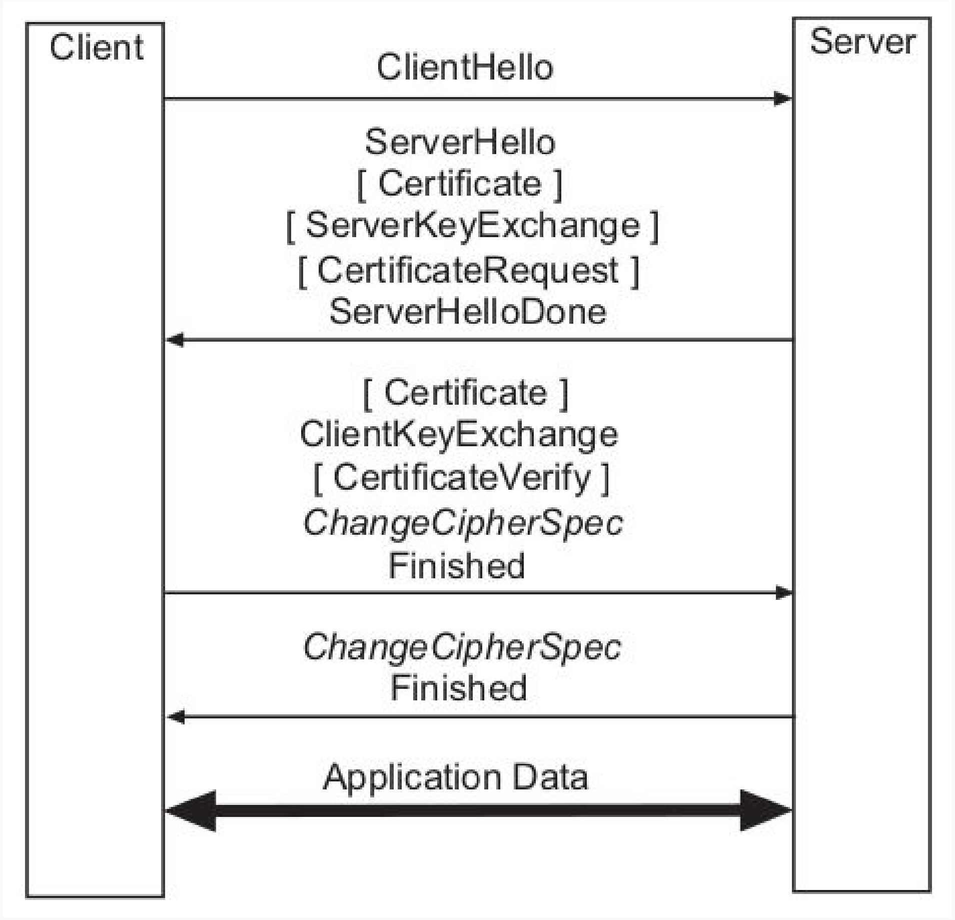
- 1-byte **Type** field (20 - Change Cipher Spec Protocol, 21 - Alert Protocol, 22 - Handshake Protocol, 23 Application Data Protocol)
- 2-byte **Version** field (3.2) for TLS 1.2
- 2-byte **Length** field, specifying the length in bytes of the following higher-layer protocol messages

SSL/TLS Handshake Protocol

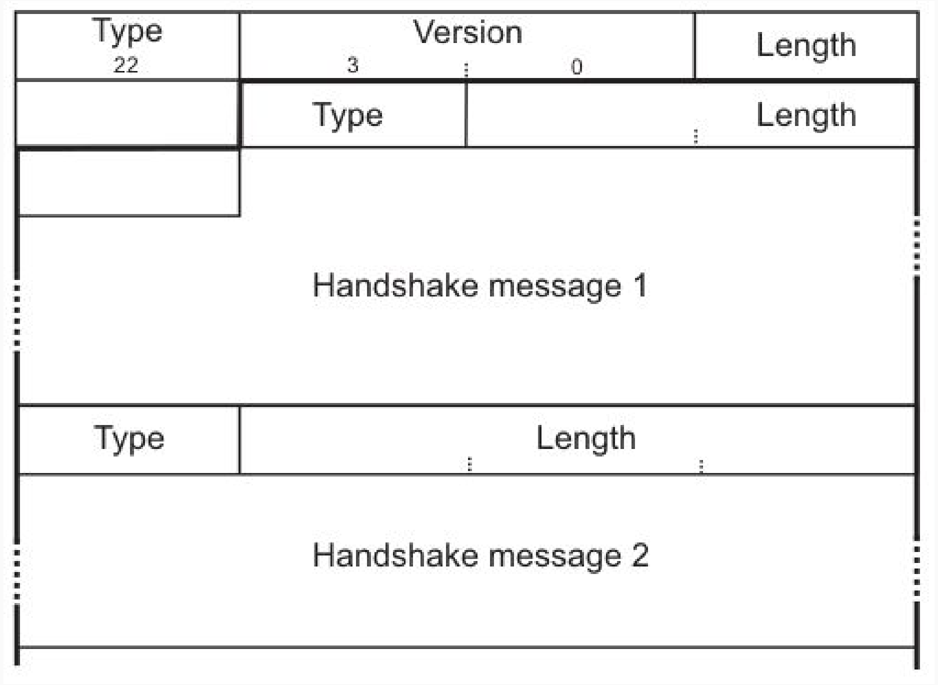
- Layered on top of **SSL/TLS Record Protocol**
- Used to **authenticate** the client to the server (and vice-versa), as well as to negotiate items like the **cipher suite** and the **compression method**



SSL/TLS Handshake Protocol



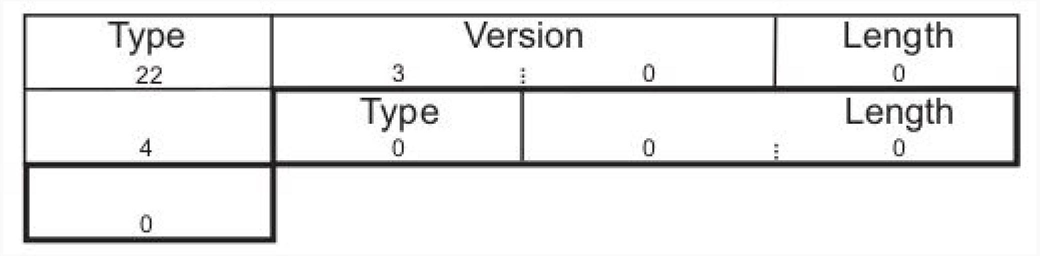
SSL/TLS Handshake Protocol: Message Structure

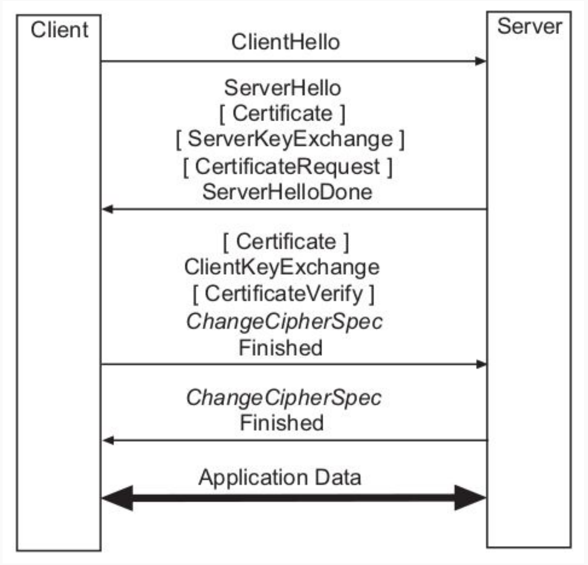


- **Multiple** messages can be sent in the same fragment, **as long as they belong to the same type** (*i.e.*, the **Type** field has the same value.

SSL/TLS Handshake Protocol: HelloRequestMessage

- Allows the **server** to ask a **client** to restart the **SSL/TLS Handshake** negotiation.
- Why?
  - Negotiate new keys if an **TLS Connection** has been used for too long



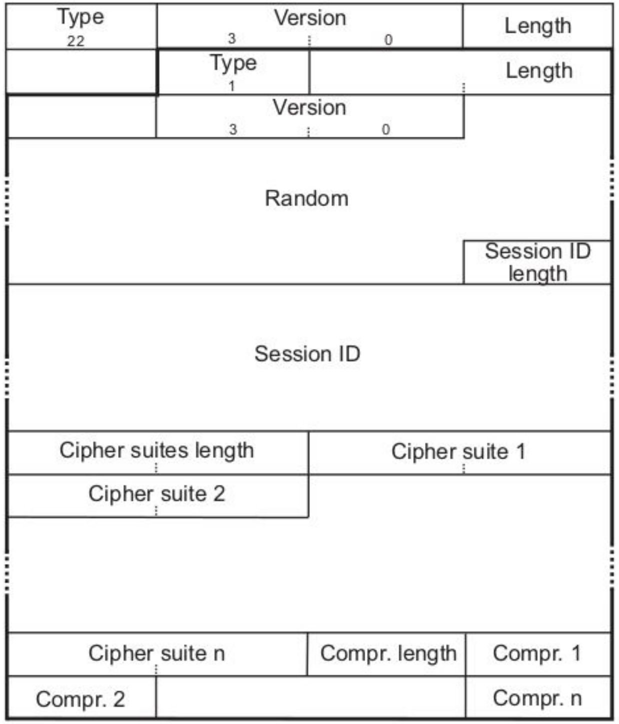


SSL/TLS Handshake Protocol: ClientHello

- Message that usually begins the **Handshake**
- Fields:
  - **Version** - highest SSL/TLS version supported
  - **Random** - 32-byte client-generated random value (4 bytes date and time + 28 pseudorandom)
  - May include a **Session ID** (to resume a session)
  - **Cipher Suite 1, Cipher Suite 2, …** - list of cipher suites Supported by the client, ordered by preference (those things are just 2-byte codes)
  - **Compr. 1, Compr. 2, …** - supported compression methods

- **IMPORTANT:** after **Compr. N**, extra data can be included
  - It can be ignored
  - This is how the **TLS Extensions** are implemented

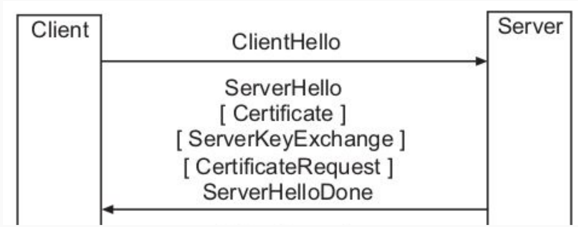
Client
ClientHello
Server

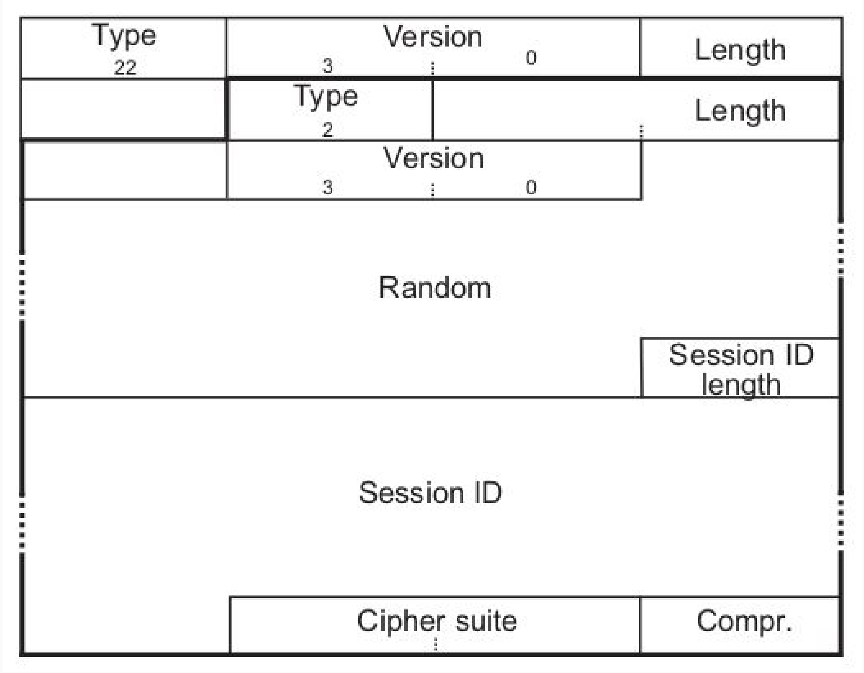


SSL/TLS Handshake Protocol: ServerHello

- **Version** - server chooses the version to be used
- **Random** - 32-byte random server value
- **SessionID** - non-empty **if a session is being resumed**
- **Cipher Suite** - server chooses the cipher suite to use
- **Compr.** - server chooses the compression method to use

NOTE: only one **Cipher Suite** and only one **Compr.**



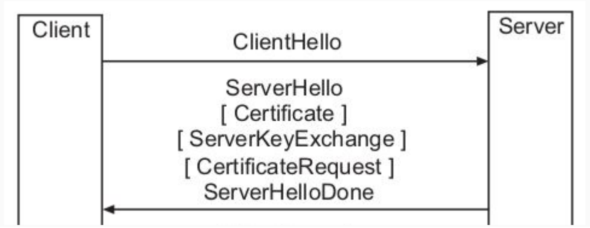


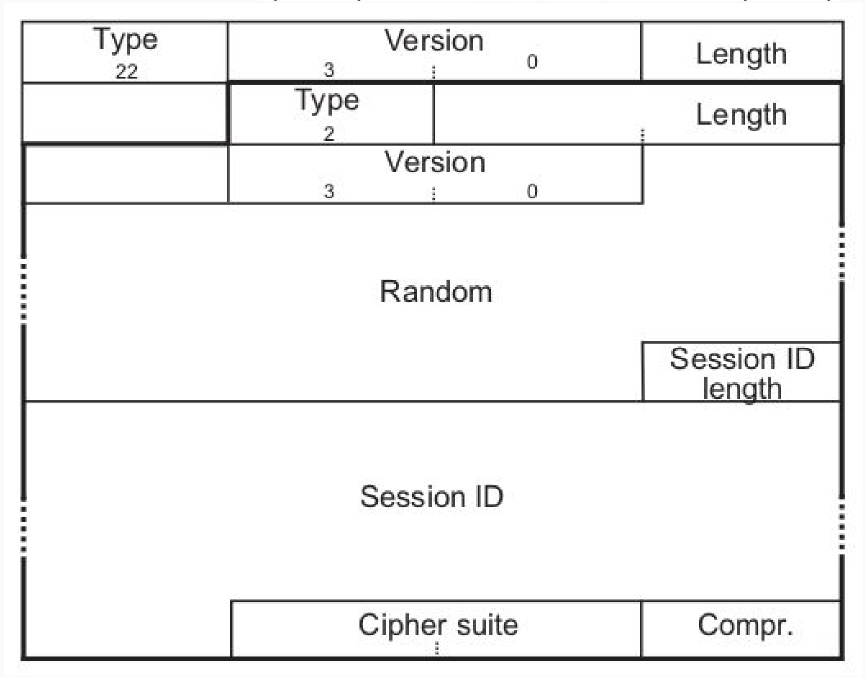
SSL/TLS Handshake Protocol: ServerHello

- **Version** - server chooses the version to be used
- **Random** - 32-byte random server value
- **SessionID** - non-empty **if a session is being resumed**
- **Cipher Suite** - server chooses the cipher suite to use
- **Compr.** - server chooses the compression method to use

NOTE: only one **Cipher Suite** and only one **Compr.**

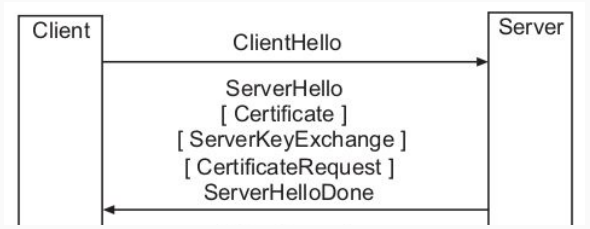
- After the server has sent the ServerHello message, it's assumed that **the client and the server have a common understanding** of the SSL/TLS version to use, the session ID to use and the cipher suites to use.

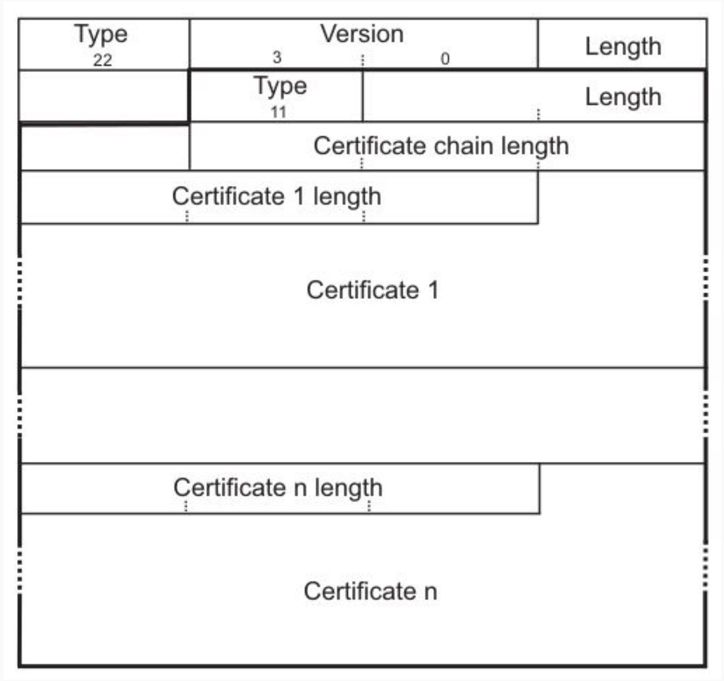




SSL/TLS Handshake Protocol: CertificateMessage

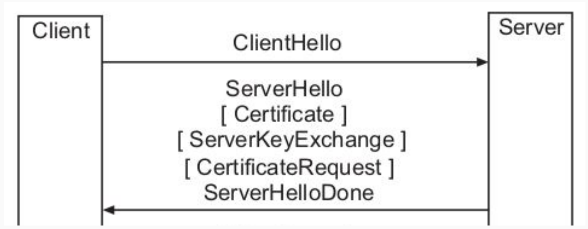
- A **chain** of **Public Key Certificates** is sent
- It’s up to the client to validate it

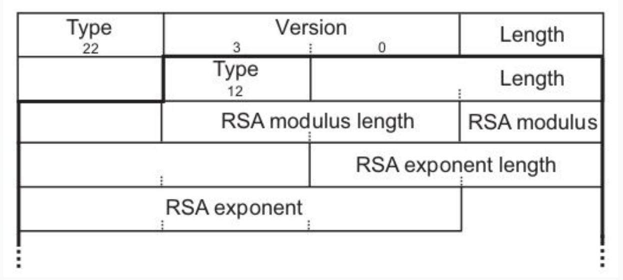


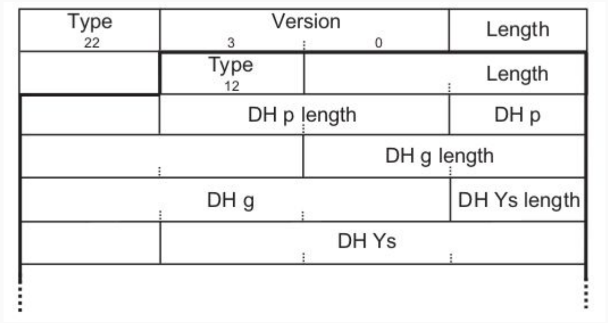


SSL/TLS Handshake Protocol: ServerKeyExchange

- Used to **generate and exchange** the **premaster secret.**
- If RSA is used:
  - Client generates the **premaster secret**, signs it with server's Public Key and sends it to it (**NO PFS**).
- If Diffie-Hellman is used:
  - Fixed Diffie-Hellman Key Exchange (DH)- DH parameters are **fixed** and part of the respective public key certificate **(NO PFS)**
  - Ephemeral Diffie-Hellman Key Exchange (DHE) - a Diffie-Hellman exchange is performed. Diffie-Hellman parameters are **generated** and **authenticated** in some way (usually digitally signed using the sender's **private key** [RSA or DSA]).
    - Only the server authenticates his public Diffie-Hellman value

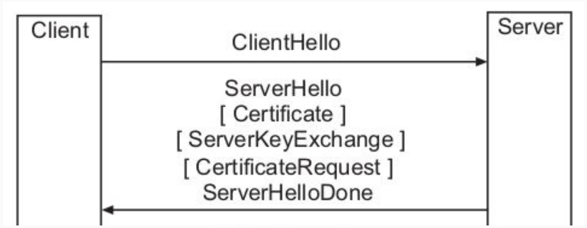


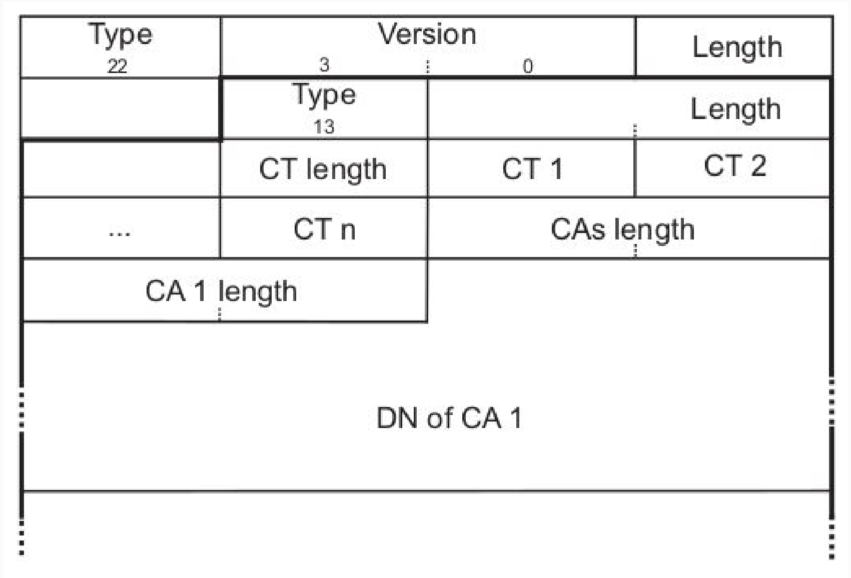




SSL/TLS Handshake Protocol: CertificateRequest

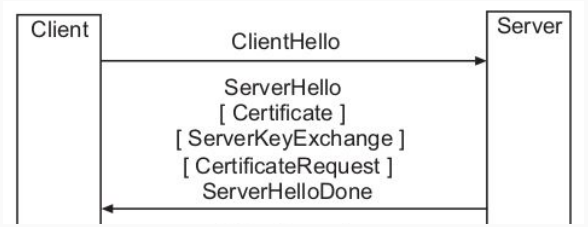
- Server may request a certificate from the client
- Server specifies the **CA**s that it considers valid

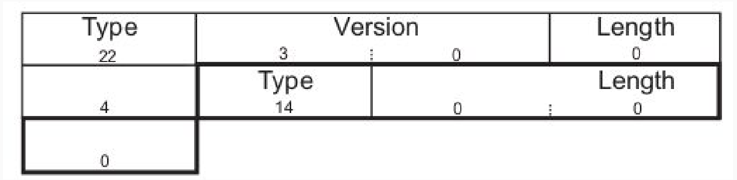




SSL/TLS Handshake Protocol: ServerHelloDone

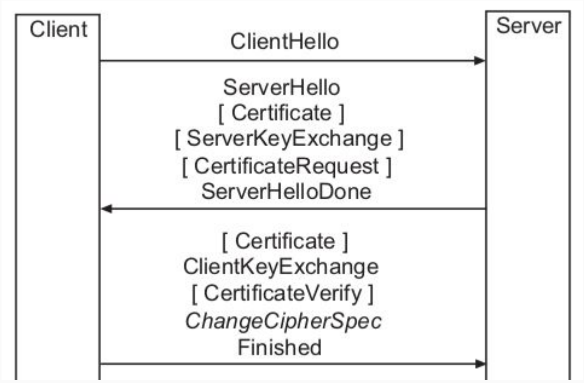
- Indicates the end of **ServerHello** and associated messages

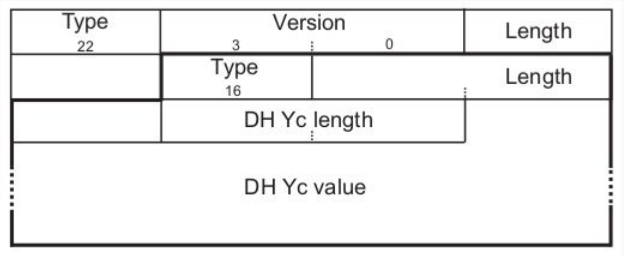


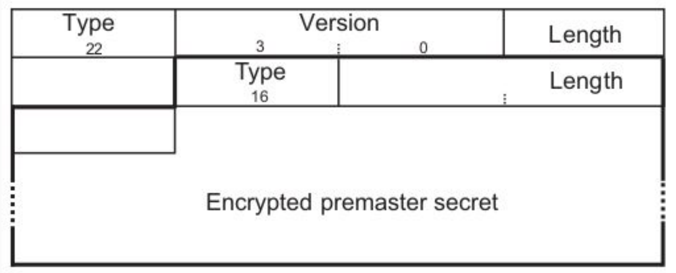


SSL/TLS Handshake Protocol: ClientKeyExchange

- One of the **MOST IMPORTANT** messages
- Client sends the encrypted **48-byte premaster secret** it generated to the
server, encrypted with the server's public key if **RSA** is used or it sends its **public Diffie-Hellman parameter** if **Diffie-Hellman** is used.
    - In the case of **RSA**, the server decrypts the **premaster secret**
    - In the case of **Diffie-Hellman**, the server generates the **premaster secret** (which will be the same as the one on the client side)
- In the case of **RSA** the first 2-bytes of the **48-byte** premaster secret refer to the SSL/TLS version chosen by the client in ClientHello and the server should check that (to prevent **version rollback attacks)**

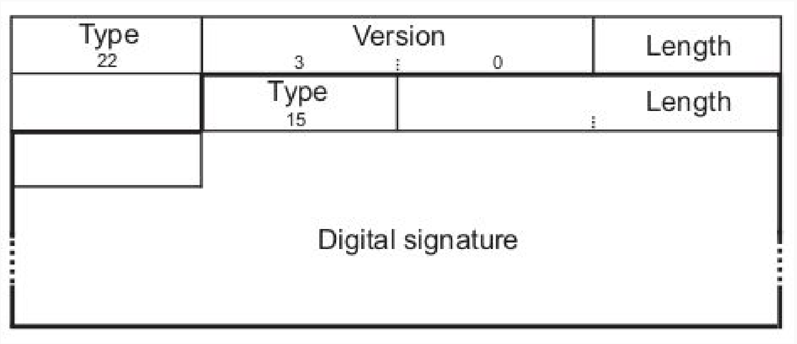


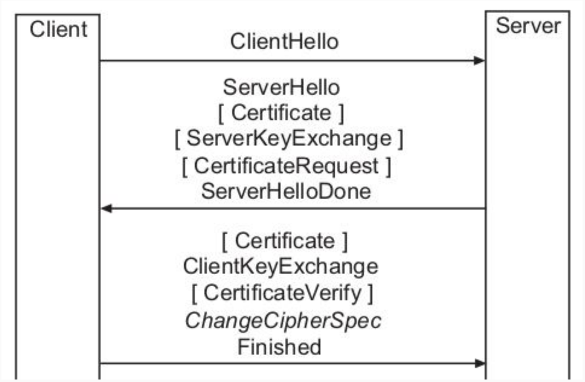




SSL/TLS Handshake Protocol: CertificateVerify

- If a **client certificate is used**, it must prove that it owns the certificate, by sending a value signed with its private key.

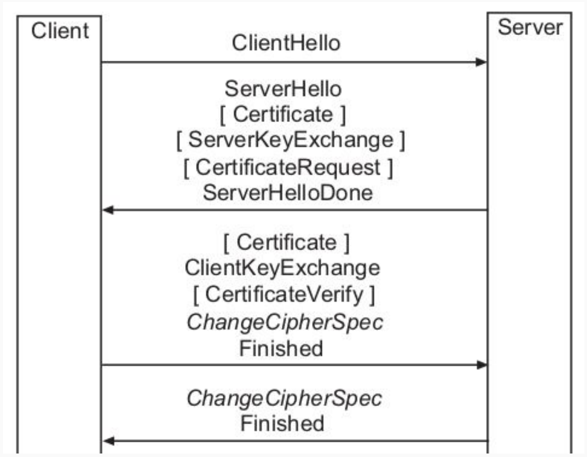


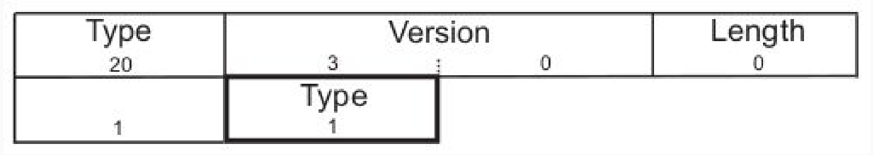


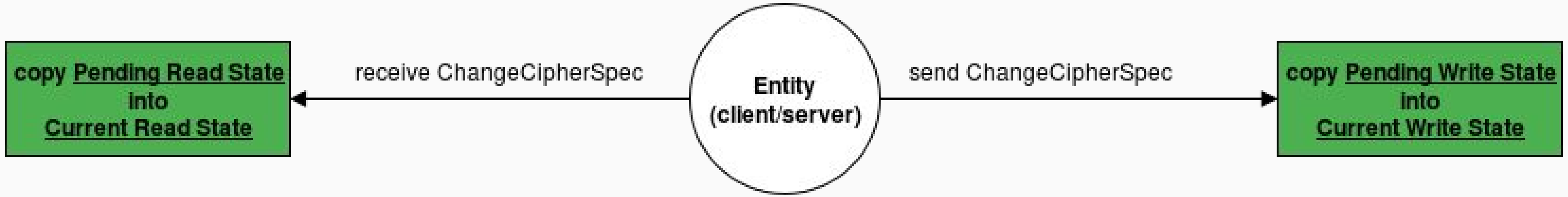
SSL/TLS ChangeCipherSpec Protocol

- used to change or set the initial encryption settings, such as the algorithms used and the keys.
- The SSL/TLS Handshake Protocol is used to negotiate the security parameters, **but those parameters only start being used after a ChangeCipherSpec message.**

- **States** contain data such as:
  - CipherSuite in use
  - Secret Keys in use







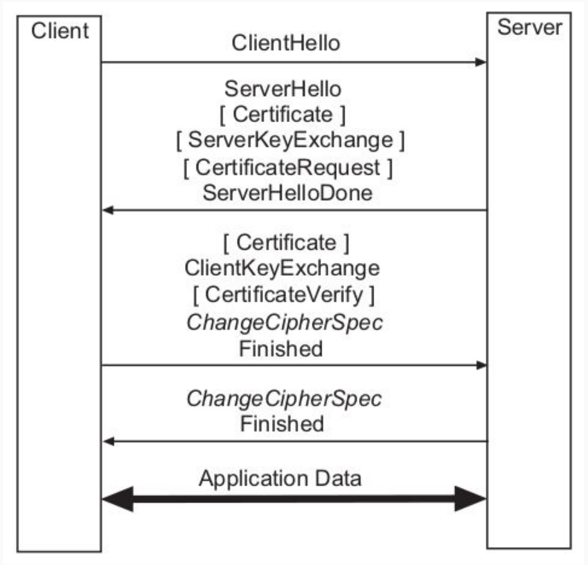
copy Pending Read State into Current Read State
receive ChangeCipherSpec
Entity (client/server)
send ChangeCipherSpec
copy Pending Write State into Current Write State

SSL/TLS Handshake Protocol: Finished Message

- Always sent immediately after the **ChangeCipherSpec** message, to **verify that the key exchange and authentication process has been successful.**
- First message that's **encrypted with the negotiated keys.**

- Structure of the message:

    -
      ```
      PRF(master_secret, finished_label, Hash(handshake_messages))
         [0..verify_data_length-1];
      ```
    - **finished_label** is "client finished"/"server finished"
    - **Hash** must the the same function as the one used in the base **PRF**.
    - **handshake_messages - ** All of the data from all messages in this handshake (not including any HelloRequest messages) up to, but not including, this message. This is only data visible at the handshake layer does not include record layer headers.

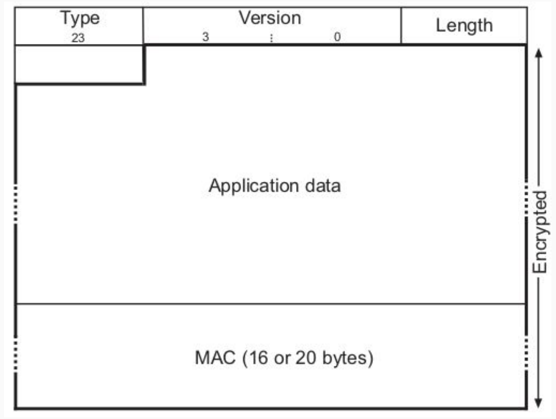


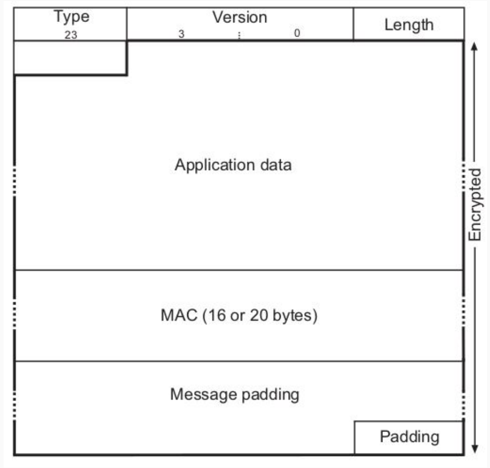
SSL/TLS Alert Protocol

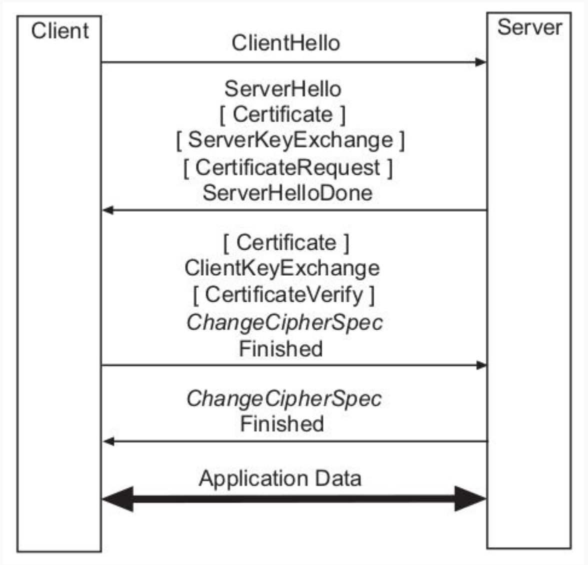
- Used to signal **errors** and **warnings**

SSL/TLS Application Data Protocol

- Allows the communicating peers to exchange the application data, using the negotiated cipher suite and keys.





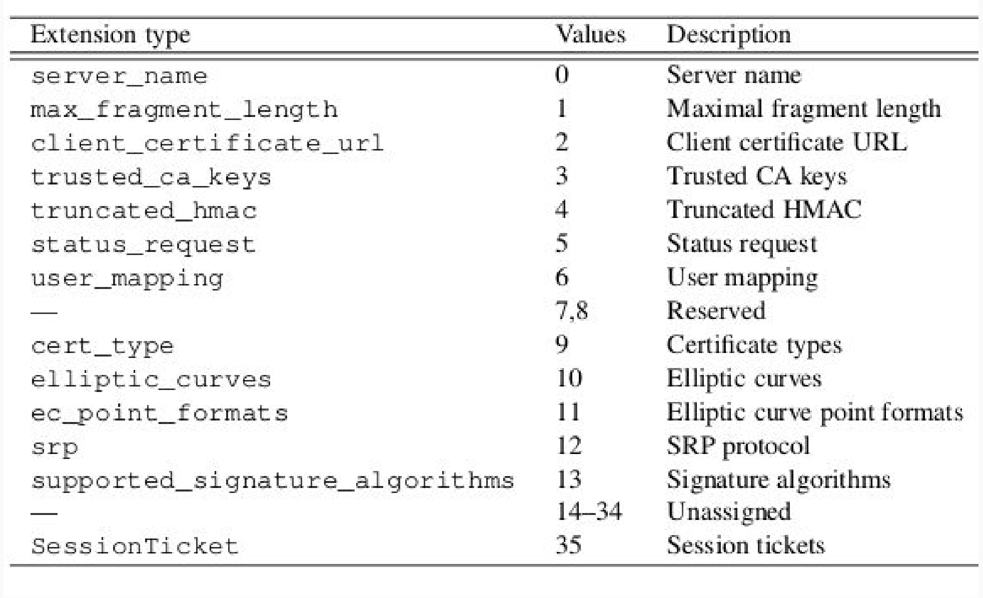


Block cipher **Stream Cipher**

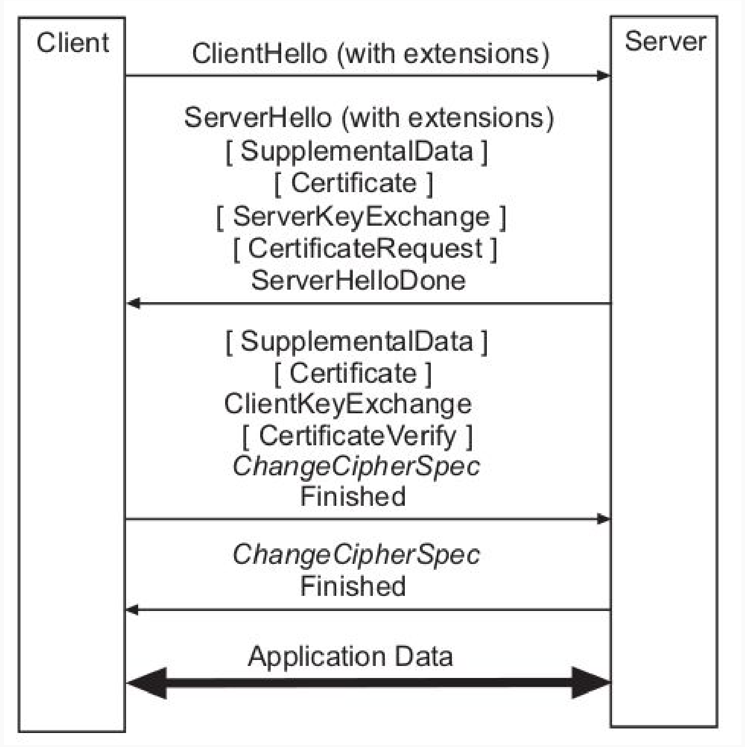
NOTE: in TLS IVs are sent explicitly and those value are missing in the image above (the image depicts SSL 3.0 Records).

TLS Extensions

- Since TLS 1.2, Extensions are now part of the base spec
- **Backwards compatible:** server only sends back the extended ServerHello if it understands the extention(s), otherwise, it ignores them and sends a “normal” ServerHello

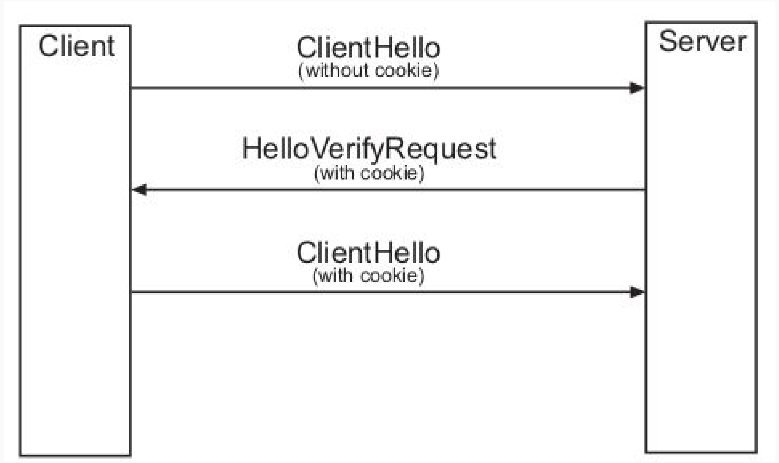


Extension type	Values	Description
server_name	0	Server name
max_fragment_length	1	Maximal fragment length
client_certificate_url	2	Client certificate URL
trusted_ca_keys	3	Trusted CA keys
truncated_hmac	4	Truncated HMAC
status_request	5	Status request
user_mapping	6	User mapping
—	7,8	Reserved
cert_type	9	Certificate types
elliptic_curves	10	Elliptic curves
ec_point_formats	11	Elliptic curve point formats
srp	12	SRP protocol
supported_signature_algorithms	13	Signature algorithms
—	14–34	Unassigned
SessionTicket	35	Session tickets



DTLS

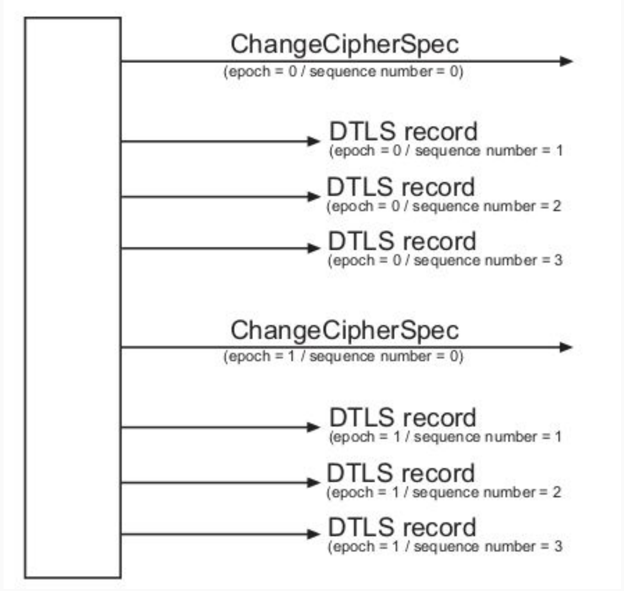
- Major differences from **TLS:**
  - **DTLS Record** has 2 new fields
  - **DTLS Handshake:**
    - 1. Modified to handle message loss, rendering and fragmentation
    - 2. Can retransmit a message if its lost (or a timeout occurs)
    - 3. **Stateless Cookie** is needed for DoS protection.



DTLS

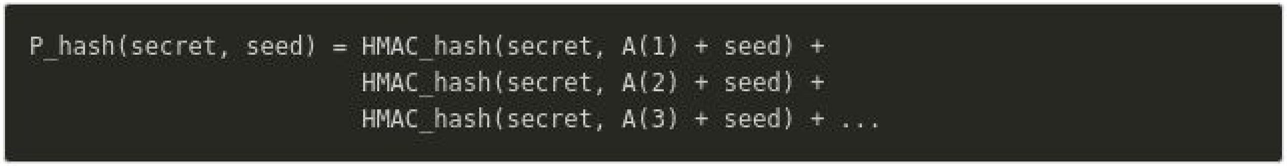
- **TLS** over a best-effort delivery datagram protocol like **UDP**.
- One **DTLS Record** is independent from another and can be decrypted separate from others.
- **DTLS Record** fragmentation should be avoided (use PMTU)
  - Separate a **DTLS Message** into multiple **DTLS Records** if needed (ex: split **handshake message** into multiple records)
- **DTLS Record Header** has 2 additional fields:
  - **2-byte epoch**, incremented on every cipher state change
  - **6-byte sequence number** that is incremented on every **DTLS Record** sent in a given cipher state.
    - Set to **0** after every **ChangeCipherSpec** message.
    - Can be used to provide replay protection, using a **sliding receiving window**
    - This sequence number yields independency between each record

- Unlike in **TLS**, **DTLS MAC** errors do not necessarily result in connection termination. Functionality as normal may proceed, since all **DTLS Records are independent one from another**.

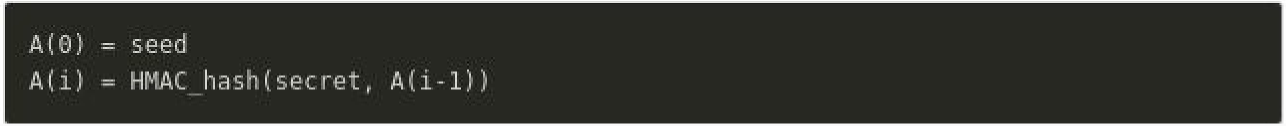

ChangeCipherSpec
(epoch = 0 / sequence number = 0)
DTLS record
(epoch = 0 / sequence number = 1
DTLS record
(epoch = 0 / sequence number = 2
DTLS record
(epoch = 0 / sequence number = 3
ChangeCipherSpec
(epoch = 1 / sequence number = 0)
DTLS record
(epoch = 1 / sequence number = 1
DTLS record
(epoch = 1 / sequence number = 2
DTLS record
(epoch = 1 / sequence number = 3

Keying Material Generation: PRF

- **TLS** uses a Pseudorandom Function (PRF) to generate the keying material:
    - **PRF(secret, label, seed) = P_hash(secret, label + seed)**



```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...
```

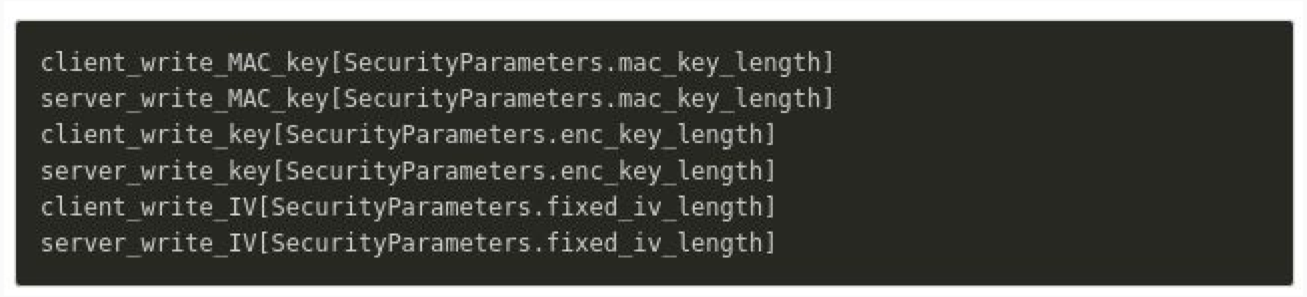


```
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))
```

- For all of the cipher suites defined in the RFC 5246, specifying TLS 1.2 and in TLS documents published prior to the publication of the RFC 5246 when TLS 1.2 is negotiated, PRF with the **SHA-256** hash function MUST be used.
- New cipher suites specified in the future **must explicitly specify a hash function to be used in PRF** and **that hash function should be SHA-256 or stronger**.

Keying Material Generation

- master secret = PRF(premaster secret, "master secret", ClientHello.random + ServerHello.random)[0..47]
- **master secret** is then **used as a source of entropy** to generate the rest of the keys
- key_block = PRF(master secret, "key expansion", ServerHello.random + ClientHello.random)
- The key block is then partitioned into the following items:



```
client_write_MAC_key[SecurityParameters.mac_key_length]
server_write_MAC_key[SecurityParameters.mac_key_length]
client_write_key[SecurityParameters.enc_key_length]
server_write_key[SecurityParameters.enc_key_length]
client_write_IV[SecurityParameters.fixed_iv_length]
server_write_IV[SecurityParameters.fixed_iv_length]
```

The End