TLS Introduction

Cryptology (1)

Martin Stanek

2025

KI FMFI UK Bratislava

SSL/TLS History

- SSL – Secure Socket Layer
- TLS – Transport Layer Security
- History:
  - 1995 SSL 2.0 (Netscape Communications)
  - 1996 SSL 3.0 (Netscape Communications)
  - 1999 TLS 1.0 (RFC 2246, “SSL 3.1”)
  - 2006 TLS 1.1 (RFC 4346)
  - 2008 **TLS 1.2 (RFC 5246)**, updated by 10 other RFCs
  - 2018 TLS 1.3 (RFC 8446)

Goals of TLS

- According to TLS 1.2 (prioritized):
  1. Cryptographic security – to establish a secure connection between two parties (data confidentiality and integrity/authenticity)
  2. Interoperability
  3. Extensibility – to provide a framework into which new public key and bulk encryption methods can be incorporated as necessary
  4. Relative efficiency – optional session caching scheme, reducing network activity
- basic cryptographic components:
  - key agreement schemes (DH, RSA)
  - server authentication (certificates), client authentication optional
  - symmetric encryption: block/stream ciphers
  - authenticating data: HMAC, AEAD (authenticated encryption with additional data)
  - PRF (pseudorandom function)
  - PRNG (pseudorandom number generator)

Support: browsers and servers

- Browsers – default settings:
  - Chrome (142), Firefox (145): TLS 1.2, 1.3
  - removed/disabled by default – TLS 1.0 and 1.1
- Servers:

	XII/2017	X/2020	VI/2025
TLS 1.0	91.0%	51.5%	23.5%
TLS 1.1	84.9%	58.5%	25.2%
TLS 1.2	89.4%	99.0%	100%
TLS 1.3		39.8%	75.3%

source: SSL Pulse

TLS applications

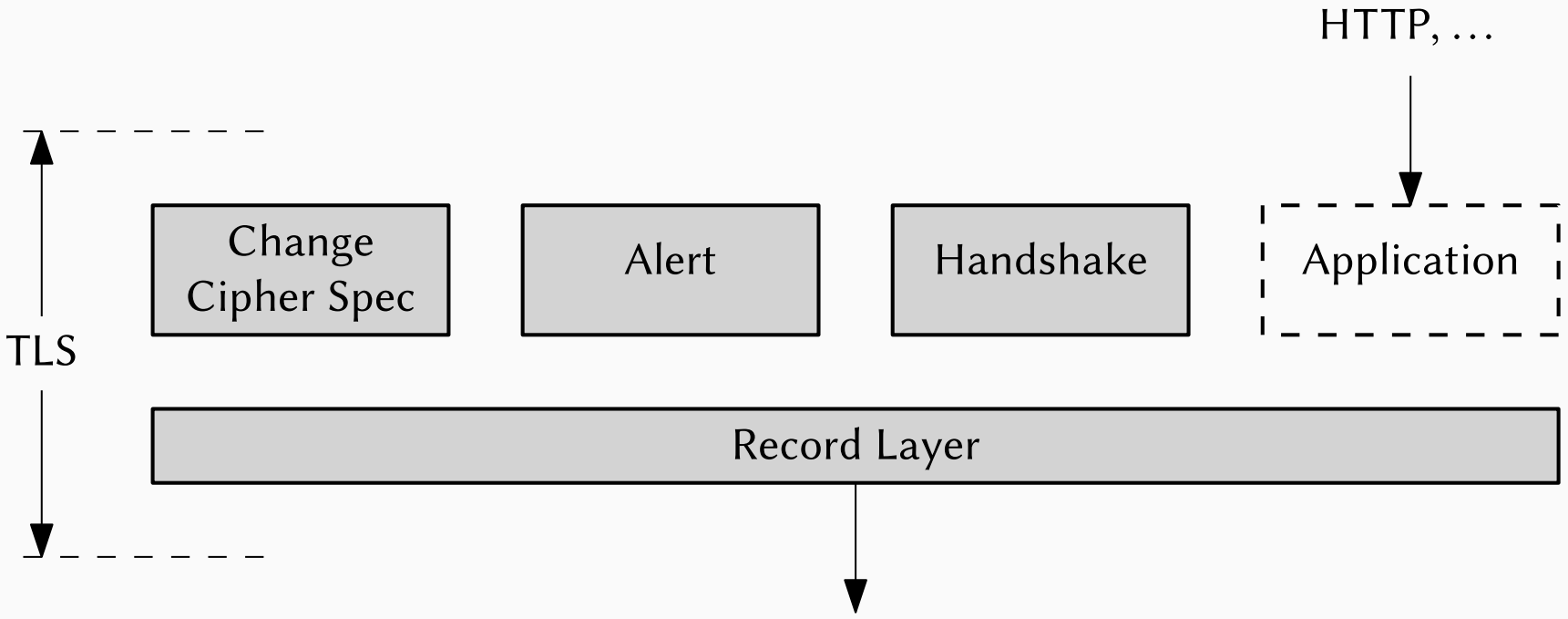
- TLS requires a reliable transport protocol (e.g. TCP)
  - see DTLS (RFC 6347) for using TLS with datagram protocols
- almost transparent to higher level protocols
- various applications:
  - web: HTTPS ≈ HTTP + TLS (the most frequently used application), QUIC
  - accessing mail: IMAP/POP3 + TLS
  - transferring mail: SMTP + TLS
  - building VPN over TLS, etc.

Limitations of TLS

- no data non-repudiation
- depends on PKI
  - certificate management (trust, distribution, revocation, etc.)
- TLS does not provide solution for web application vulnerabilities
  - SQL injection, XSS, CSRF, etc.
- TLS does not provide solution for weaknesses on user's side
  - weak passwords, accepting suspicious certificates, etc.

Structure

- client ⟷ server (asymmetric communication)
- two layers, subprotocols



TLS Connection State (1)

- client and server maintain/update their connection states
  - connection end (client/server)
  - encryption algorithm (block, stream, AEAD)
  - MAC algorithm
  - compression algorithm
  - PRF function
  - master secret (shared secret, 48 B)
  - client random (32 B)
  - server random (32 B)
  - sequence number (starting at 0, less than 2^{64}, does not wrap, incremented after each record)
- other data required for the state:
  - compression state, cipher state (e.g. scheduled key /stream cipher's state)

TLS Connection State (2)

- all required keys and initialization vectors are derived from master secret, client random and server random values
  - client write [MAC key | encryption key | IV]
  - server write [MAC key | encryption key | IV]
- 4 states for each connection end:
  - current [read | write] state
  - pending [read | write] state
- initial state: ciphersuite `TLS_NULL_WITH_NULL_NULL`
  - transformation of data ≈ identity (no MAC, no encryption, no compression)

TLS Record Protocol

- record layer processes data from higher layers:
  - fragmentation ($\leq 2^{14}$ bytes)
  - compression (NULL)
  - MAC computation and encryption, or AEAD encryption
- content: [type, version, length, fragment data]
  - type – 20 (ChangeCipher), 21 (Alert), 22 (Handshake), 23 (Application)
  - version – 3.0 (SSL 3.0), 3.1 (TLS 1.0), 3.2 (TLS 1.1), 3.3 (TLS 1.2)
  - length – length of the fragment data
  - fragment data – processed data (MAC and encryption, or AEAD)
- MAC-then-Encrypt
  - MAC is computed from concatenated sequence number, type, version, length and data

Application Data

- processed transparently by record layer (fragmented, encrypted etc.)
- processing based on the connection state

Change Cipher Spec Protocol

- single message (single byte containing value 1)
- signals a change in cryptographic state
- switch to pending write state immediately after sending
- switch to pending read state immediately after receiving

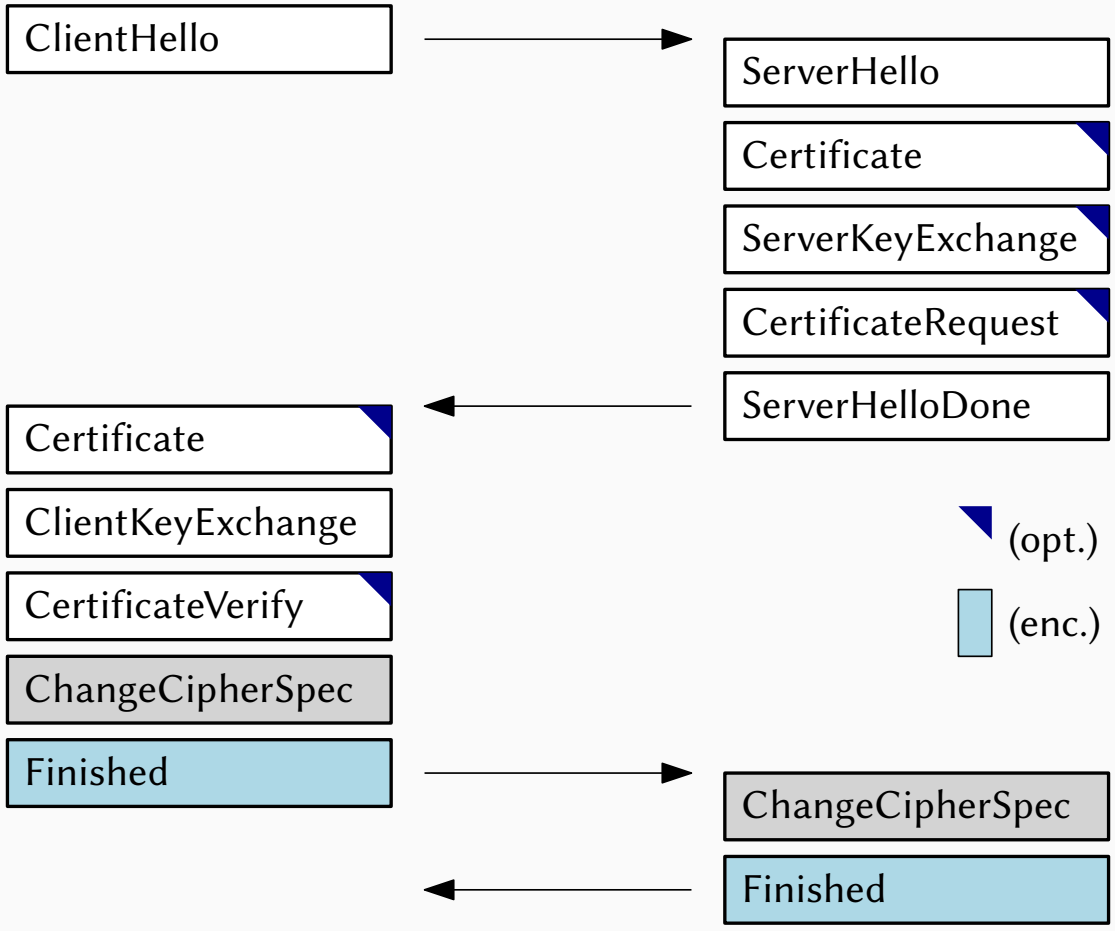
Alert Protocol

- information about error state, connection closure
- message (2 bytes):
  - level – 01 (warning), 02 (fatal)
  - code (25 overall) – close notify, bad record MAC, unknown_ca, record_overflow, protocol_version etc.
- fatal $\Longrightarrow$ terminate the connection immediately

Handshake Protocol – overview (1)

1. Exchange hello messages, agree on algorithms, exchange random values (nonces), check for session resumption.
2. Exchange certificates to authenticate server (mandatory) and client (optional).
3. Exchange parameters and values to agree on a pre-master secret.
4. Calculate master secret from the pre-master secret and random values. Calculate necessary keys and other parameters.
5. Switch to agreed algorithms and keys.
6. Verify that the other communication end calculated the same parameters.

Handshake Protocol – overview (2)



ClientHello

- structure:
  - TLS version
  - client_random (4B seconds from 1.1.1970; 28B random bytes)
  - session ID: allows reusing the parameters from previous or simultaneous connection
  - supported cipher suites (sorted by client's preference)
  - compression methods
  - extensions (optional)
- extension examples:
  - server_name (SNI): which hostname the client is attempting to connect to
  - elliptic_curves: set of elliptic curves supported by the client
  - TLS session ticket: encrypted session state sent to client (later used for session resumption)
  - signature_algorithms: indicates supported combinations of algorithm and hash function for digital signatures

List of supported cipher suites – example

User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:145.0) Gecko/20100101 Firefox/145.0

client-preferred order:

```
TLS_AES_128_GCM_SHA256
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
```

List of supported signatures and curves – example

Firefox/145.0

- signature algorithms:
  SHA256/ECDSA, SHA384/ECDSA, SHA512/ECDSA, RSA_PSS_SHA256, RSA_PSS_SHA384, RSA_PSS_SHA512, SHA256/RSA, SHA384/RSA, SHA512/RSA, SHA1/ECDSA, SHA1/RSA, SHA384/RSA, SHA512/RSA, SHA1/ECDSA, SHA1/RSA
  (*) PSS schemes defined in TLS 1.3
- named groups:
  X25519MLKEM768, x25519, secp256r1, secp384r1, secp521r1, ffdhe2048, ffdhe3072
- X25519MLKEM768: combining X25519 ECDH with ML-KEM-768

ServerHello

- structure:
  - TLS version
  - server_random (4B seconds from 1.1.1970; 28B random bytes)
  - session ID: identification of the session
    session ID from ClientHello found in session cache ⇒ session resumption, proceed to Finished message
    non-empty (different value): new session ID
    empty: session will not be cached
  - selected cipher suite (from the client's list)
  - selected compression method
  - extensions (optional, subset of extensions offered by client)

(Server) Certificate

- server's certificate chain (X.509v3 certificates)
- self-signed certificate of root CA distributed independently
- required if key exchange methods use it for authentication (all except DH_anon)
- server's certificate type must by suitable for selected key exchange method, e.g.
  - RSA method requires RSA public key certificate that allows the key to be used for encryption
  - ECDHE_RSA method requires RSA public key that allows selected digital signature scheme and hash algorithm

Key exchange methods

- RSA
  - client generates a pre-master secret (48B)
  - client encrypts the pre-master secret using RSA public key of the server
  - server decrypts using its private key
  - remark: RSA encryption – PKCS#1 v 1.5 (no RSA-OAEP for TLS 1.2)
- Diffie-Hellman protocol
  - fixed DH – public parameters are part of the server's certificate
  - ephemeral DH – public parameters specified by the server, signed (RSA, DSA, ECDSA) and sent to the client in a message
  - anonymous DH – no authentication, MITM possible

ServerKeyExchange

- if server needs to send parameters required for key exchange method
  - typical use cases: (EC)DHE_[DSS|RSA]
  - DH_anon
- DH parameters:
  - DHE: p, g, server's public "key" and their signature
  - ECDHE: usually ID of a named curve (e.g. 0x0017 – P-256, generator is then fixed implicitly), public "key" and their signature
- signatures:
  - client used the signature_algorithm extension ⇒ server selects accordingly
  - client did not use the extension ⇒ server uses appropriate default (depending on cipher suite)
  - remark: RSA signatures – PKCS#1 v 1.5 (no RSA-PSS for TLS 1.2)

List of supported cipher suits – example

www.uniba.sk (November 2025), server-preferred order for TLS 1.2

 # TLS 1.2 (suites in server-preferred order)

TLS_RSA_WITH_AES_128_CBC_SHA (0x2f)			**WEAK**
TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x33)	DH 2048 bits	FS	**WEAK**
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45)	DH 2048 bits	FS	**WEAK**

CertificateRequest and ServerHelloDone

- if client authentication is required (rarely)
  - only non-anonymous server can request client authentication
- structure:
  - list of accepted certificate types (such as rsa_sign, dss_sign)
  - list of supported signature and hash algorithm pairs
  - list of distinguished names of acceptable CAs
- ServerHelloDone
  - signaling the end of server's messages

ClientKeyExchange

- structure and content depend on key exchange method
- RSA:
  - client generates pre-master_secret (48B):
    preMS = TLS version from ClientHello || random value (46B)
  - client encrypts preMS using server's RSA public key
- DH:
  - client's public "key" (not signed)
  - empty content if static DH exponent (in certificate) is used
    preMS = key obtained from DH exchange

Computing keys from pre-master secret

- computing master_secret (length 48B):
  MS = PRF(pre-master secret, master secret'', client_random || server_random)
- key material computed in defined order by partitioning sufficiently long output from PRF(MS, "key expansion", client_random || server_random)

  order (remark: IV values are used only for AEAD modes):
  - client_write_MAC_key
  - server_write_MAC_key
  - client_write_key
  - server_write_key
  - client_write_IV
  - server_write_IV

CertificateVerify, ChangeCipherSpec

- explicit verification of a client certificate
  - client's certificate must be suitable for digital signatures
- content: digital signature of all handshake messages sent and received up to this point
- ChangeCipherSpec – switch pending write state
  - all subsequent messages/data are protected

Finished

- transmitted after ChangeCipherSpec
- verification that key exchange and authentication were successful
- content (length 12B, if cipher suite does not specify longer):
  PRF(MS, label, H(msgs))
  - label = “client finished” / “server finished”
  - H – hash function used in PRF construction
  - msgs = all messages in handshake protocol up to this point
- server and client verify Finished messages once received
  - Handshake protocols ends successfully only after verifying these messages

Some cryptographic details

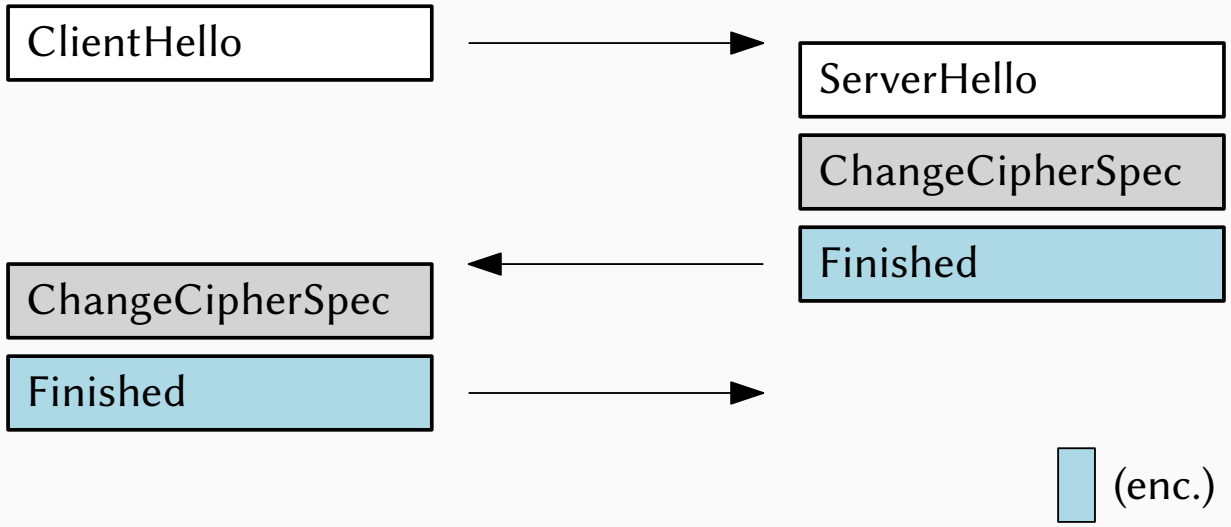
- MAC (in case of non-AEAD cipher): HMAC based on selected hash function
  - other MAC construction can be used, e.g., Poly1305 (using stream cipher ChaCha20 inside, RFC 7539)
- Mandatory implemented cipher suite: TLS_RSA_WITH_AES_128_CBC_SHA
- CBC mode for block ciphers:
  - explicit IV, (should be) random, (must be) unpredictable
  - padding examples: (incomplete last block || 00), (incomplete last block || 02 02 02)

PRF construction

- computing PRF(secret, label, seed) = P_hash(secret, label || seed)
- hash is SHA256 for TLS 1.2 (default)
- P_hash(secret, seed) – data expansion function:

$$
\begin{aligned}
\text{P\_hash}(\text{secret}, \text{seed}) = {} & \text{HMAC\_hash}(\text{secret}, \text{A}(1) \,||\, \text{seed}) \,||\\
& \text{HMAC\_hash}(\text{secret}, \text{A}(2) \,||\, \text{seed}) \,||\\
& \text{HMAC\_hash}(\text{secret}, \text{A}(3) \,||\, \text{seed}) \,||\, \ldots\\
\text{A}(0) = {} & \text{seed}\\
\text{A}(i) = {} & \text{HMAC\_hash}(\text{secret}, \text{A}(i\text{-}1))
\end{aligned}
$$

Handshake Protocol – session resumption



- Session ID, state stored (cached) by client and server
- alternative: Session tickets (state stored by client)

Forward Secrecy (FS)

- FS: previous session keys are not compromised even if the long term keys are
  - desirable property of key agreement/distribution protocols
- TLS 1.2:
  - RSA: obtaining server's RSA private key reveals all previous and future pre-master secrets (all keys can be recomputed from pre-master secret)
  - ephemeral non-anonymous DH (DHE, ECDHE): FS

Exercises

1. *Choose a public web server and analyze the results provided by SSL Server Test (what information is available by this test).*
2. *Try testssl.sh tool for TLS server assessment and analyze the results.*
3. *Find what extensions are included in ClientHello message sent by your web browser (use wireshark or other network protocol analyzer). Compare with `curl` command.*