Transport Layer Security

“TLS is a cryptographic protocol that provides communication security over the Internet.”

- Provides confidentiality, integrity and server authentication
- The most successful and widely used cryptographic protocol (!!!)
- Any application protocol can be encapsulated in TLS
TLS version history

- SSL 1.0 – never publicly released
- SSL 2.0 – Netscape (1995)
- SSL 3.0 – Netscape (1996)
- TLS 1.0 (SSL 3.1) – RFC 2246 (1999)
- TLS 1.1 – RFC 4346 (2006)
- **TLS 1.2** – RFC 5246 (2008)
- TLS 1.3 – RFC 8446 (2018)
TLS handshake

• Client verifies server’s X.509 certificate
• Client extracts from the certificate server’s public key
• Client encrypts a random symmetric key using server’s public key
• Only the server can decrypt the symmetric key
• Now the client and server share the same symmetric key
• Symmetric key is used for the actual data encryption/authentication
TLS session resumption

- Resumed TLS connections share the same “master secret”
- Several TLS connections can belong to the same TLS session
- If TLS connection fails, TLS session becomes non-resumable
- Abbreviated handshake improves performance, saving:
  - 1 round-trip time across the network
  - 1 asymmetric crypto operation
TLS Record Layer

[Type][Version][Length][Data]

- **Type**: type of encapsulated data:
  - Handshake message (0x16)
  - Change Cipher Spec message (0x14)
  - Alert message (0x15)
  - Application data (0x17)
- **Protocol version**: 0x0303 (for TLS v1.2)
- **Length**: length of the data (2 bytes)
- **Data**: encapsulated data
  - Can contain several same type messages

TLS record header is never encrypted!
Dissecting TLS with Wireshark
Alert message

Signals about TLS related issues to other party

[Level] [Description]

- Level (1 byte):
  - Warning (0x01)
  - Fatal (0x02)

- Description (1 byte):
  close_notify(0),
  unexpected_message(10),
  bad_record_mac(20),
  decryption_failed(21),
  handshake_failure(40),
  bad_certificate(42),
  unsupported_certificate(43),
  certificate_revoked(44),
  certificate_expired(45),
  illegal_parameter(47),
  unknown_ca(48),
  access_denied(49),
  decrypt_error(51),
  user_canceled(90),
  ...
Change Cipher Spec message

Signals to other party that from now on the negotiated cipher suite will be used to protect outgoing messages

[0x01]
Application data

Contains (most likely encrypted) application data in a form as required by the application protocol (e.g., HTTP request/response etc.)

[Application Data]

```
TLSv1.1 Record Layer: Application Data Protocol: http
Content Type: Application Data (23)
Version: TLS 1.1 (6x0302)
Length: 448
Encrypted Application Data: 3a37312ac35ea3809f392b2b76174849218d83f179d6d305...
```
Handshake message

Contains protocol handshake parameters

[Type] [Length] [Body]

- **Type**: message type:
  
  - `hello_request(0), client_hello(1), server_hello(2),
  certificate(11), server_key_exchange(12),
  certificate_request(13), server_hello_done(14),
  certificate_verify(15), client_key_exchange(16),
  finished(20)

- **Length**: length of the body (3 bytes)

- **Body**: message body
  
  - Can be split over several TLS records
Handshake message: ClientHello

- The highest TLS version supported (2 bytes)
- Client randomness (32 bytes)
  - Timestamp in first 4 bytes
- Session ID length (1 byte) + session ID
- Cipher suites length (2 bytes)
- List of cipher suites supported:
  - 0x0005 – TLS_RSA_WITH_RC4_128_SHA
  - 0x002f – TLS_RSA_WITH_AES_128_CBC_SHA
  - 0x0035 – TLS_RSA_WITH_AES_256_CBC_SHA
  - 0x0039 – TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- Compression methods length (1 byte)
- List of compression methods supported:
  - 0x00 – null (mandatory)
  - 0x01 – DEFLATE (gzip)
- Extensions (optional)
Handshake message: ServerHello

- TLS version selected (2 bytes)
- Server randomness (32 bytes)
  - Timestamp in first 4 bytes
- Session ID length (1 byte) + session ID
- Cipher suite selected (2 bytes)
- Compression method selected (1 byte)
- Extensions (optional)
Handshake message: Certificate

- Length of certificate list (3 bytes)
- List of certificates
  - Certificate length (3 bytes)
  - DER encoded certificate
- The first is server’s certificate
- Other certificates are optional
  - Usually intermediate CA certificates
Handshake message: ServerHelloDone

- Empty message body
- Tells that there will be no more messages from the server in this protocol round
Handshake message: ClientKeyExchange

Contains (two-byte length-prefixed) encrypted 48-byte random “pre-master secret”

- Encrypted using the public key from the server’s certificate
- Encrypted according to PKCS#1 v1.5
- The first two bytes in the pre-master secret contain the TLS version
  - Must be checked by the server
  - Prevents some attacks (?)
- Next 46 bytes are truly random bytes
Handshake message: Finished

• The first encrypted message
• Serves to verify if encryption works
• Contains hash of concatenation of all previous handshake messages (excluding the TLS record header)
  • Must be verified by other party to detect downgrade attacks

```plaintext
TLSv1.1 Record Layer: Handshake Protocol: Encrypted Handshake Message
  Content Type: Handshake (22)
  Version: TLS 1.1 (0x0302)
  Length: 64
  Handshake Protocol: Encrypted Handshake Message
```
How many symmetric keys are needed?
- MAC & encrypt (+ IV for block ciphers)
- Separate keys for each direction

How to derive these keys from the 48-byte pre-master secret?
Key derivation

- TLS defines PRF() (pseudo-random function)
  - Uses SHA256
  - Produces infinitely long pseudo-random output

- From the 48-byte “pre-master secret” a 48-byte “master secret” is derived:
  \[
  \text{PRF}(\text{premaster } + \text{’master secret’ } + \text{client_random } + \text{server_random}, 48)
  \]

- From the “master secret” is derived a key block in the size needed:
  \[
  \text{PRF}(\text{master_secret } + \text{’key expansion’ } + \text{server_random } + \text{client_random}, 136)
  \]

- The key block is split into the keys needed:
  
  ```python
  client_mac_key = key_block[:20]
  server_mac_key = key_block[20:40]
  client_enc_key = key_block[40:56]
  server_enc_key = key_block[56:72]
  client_iv = ...
  ...
  ```
MAC calculation

\[ \text{HMAC\textunderscore hash}(\text{key}, \text{seq} + \text{type} + \text{version} + \text{length} + \text{data}) \]

- **hash**: hash algorithm from the negotiated cipher suite
- **key**: client/server MAC key
- **seq**: client/server sequence number (8 bytes)
  - Starts from 0
  - Incremented for every TLS record sent
- **type**: TLS record type
- **version**: TLS protocol version (2 bytes)
- **length**: length of the data (2 bytes)
- **data**: TLS record payload
Task: TLS getcert – 4p

Implement TLS v1.2 client that can retrieve server’s certificate:

```
$ ./tls_getcert.py https://www.eesti.ee/ --certificate server.pem
--> ClientHello()
<--- Handshake()
     <---- ServerHello()
          [+] server randomness: 0F0F1A925B17BD55B165A30259AD7BFFF4514788382741D6E521A16C7DAC83D76
          [+] server timestamp: 1978-01-03 07:01:22
          [+] TLS session ID: EF82F90493D73D79776B3A4F9B3E852130727353190FAF379428CC22AB0E1570
          [+] Cipher suite: TLS_RSA_WITH_AES_256_CBC_SHA
<--- Handshake()
     <---- Certificate()
          [+] Server certificate length: 1882
          [+ ]Server certificate saved in: server.pem
<--- Handshake()
     <---- ServerHelloDone()
--> Alert()
     [+] Closing TCP connection!
```

```
$ openssl x509 -in server.pem -text | grep 'Subject:'

Subject: C = EE, L = Tallinn, O = Estonian Information System Authority, CN = *.eesti.ee
```
Task: TLS getcert

$ ./tls_getcert.py https://www.twitter.com/
---> ClientHello()
<--- Handshake()
    <--- ServerHello()
       [+] server randomness: 68167B6301F828E2D57C3A41D39DD5B67D9085959FAE7E72A4A5E11EFDF84E83
       [+] server timestamp: 2025-05-03 23:24:03
       [+] TLS session ID:
       [+] Cipher suite: TLS_RSA_WITH_AES_128_CBC_SHA
    <--- Handshake()
    <--- Certificate()
       [+] Server certificate length: 1652
    <--- Handshake()
    <--- ServerHelloDone()
---> Alert()
[+] Closing TCP connection!

$ ./tls_getcert.py https://www.live.com/
---> ClientHello()
<--- Handshake()
    <--- ServerHello()
       [+] server randomness: 5E9D052EFD33DC0F286721B02E731E9798F9E531693F3F6FB429B4561D8647CA
       [+] server timestamp: 2020-04-20 05:13:02
       [+] TLS session ID: 061600007D202EE4C11CAD43A98AE9F860C4FF4C41F9FDC48C5083FB5D9A35F7
       [+] Cipher suite: TLS_RSA_WITH_AES_256_CBC_SHA
    <--- Certificate()
       [+] Server certificate length: 2015
    <--- ServerHelloDone()
---> Alert()
[+] Closing TCP connection!
Task: TLS getcert

- Use Wireshark to see what bytes are actually sent out on the wire
  - Use capture filters 'host www.twitter.com and port 443'
- NB! One TLS record can contain several handshake messages
- Unix timestamp can be obtained using `int(time.time())`
- Unix timestamp can be printed using:
  ```python
datetime.datetime.fromtimestamp(int(time.time())).strftime('%Y-%m-%d %H:%M:%S')
```