MTAT.07.017
Applied Cryptography

Transport Layer Security (TLS)

University of Tartu

Fall 2022
Transport Layer Security (TLS)

“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 the server’s public key from the certificate
- Client encrypts a random symmetric key using the 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 a TLS connection fails, the TLS session becomes non-resumable
- Abbreviated handshake improves performance, saving:
  - 1 round-trip time across the network
  - 1 asymmetric crypto operation
TLS Record Layer

\[[\text{Type}] [\text{Version}] [\text{Length}] [\text{Data}]\]

- **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

Encapsulated frame:

Ethernet II, Src: 00:8c:fd:9d:1f:fc, Dst: 08:00:20:63:63:63
Internet Protocol Version 4, Src: 172.17.37.100, Dst: 172.17.37.100

Layer: TLSv1.2 Record Layer: Handshake Protocol: Client Hello
- Content Type: Handshake (22)
- Version: TLS 1.2 (0x0303)
- Length: 45

   Handshake Protocol: Client Hello
   - Handshake Type: Client Hello (1)
     - Length: 45
     - Version: TLS 1.2 (0x0303)
Alert message

Signals about TLS related issues to the 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.)

[TLSv1.2 Record Layer: Application Data Protocol: http-over-tls]
Content Type: Application Data (23)
Version: TLS 1.2 (0x0303)
Length: 38

Encrypted Application Data: f48f13030665f43f3d9a07ba27dccc824c4dc8dfbe9e16
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
- Indicates that there will be no more messages from the server in this protocol round

```
TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
  Content Type: Handshake (22)
  Version: TLS 1.2 (0x0303)
  Length: 4
- Handshake Protocol: Server Hello Done
  Handshake Type: Server Hello Done (14)
  Length: 0
```
Handshake message: ClientKeyExchange

Contains an (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 whether encryption works
- Contains a hash of the concatenation of all previous handshake messages (excluding the TLS record header)
  - Must be verified by other party to detect downgrade attacks
• How many symmetric keys are needed?
  • MAC & encrypt (+ IV for block ciphers)
  • Separate keys for each direction
• How do we 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:
  \[
  \text{master\_secret} = \text{PRF(premaster + 'master\_secret' + client\_random + server\_random, 48)}
  \]

- From the “master secret” a key block in the size needed is derived:
  \[
  \text{key\_block} = \text{PRF(master\_secret + 'key\_expansion' + server\_random + client\_random, 136)}
  \]

- The key block is split into the keys needed:
  \[
  \text{client\_mac\_key} = \text{key\_block[:20]}
  \text{server\_mac\_key} = \text{key\_block[20:40]}
  \text{client\_enc\_key} = \text{key\_block[40:56]}
  \text{server\_enc\_key} = \text{key\_block[56:72]}
  \text{client\_iv} = \ldots
  \]
  \[
  \ldots
  \]
MAC calculation

HMAC_hash(key, seq + type + version + length + 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**

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

```bash
$ ./tls_getcert.py https://facebook.com/ --certificate server.pem
```

--> ClientHello()

<---- Handshake()

<---- ServerHello()

[+] server randomness: C5BAC25C2C7E1EADF306D88CBF1A77FBAF5BB06C661304C9DC2195FD08E7C7

[+] server timestamp: 2075-02-14 08:23:24

[+] TLS session ID: A6D312741036DE3A304A65829119C5685CEA2082EB85E82DE31846CEA8964809

[+] Cipher suite: TLS_RSA_WITH_AES_128_CBC_SHA

<---- Handshake()

<---- Certificate()

[+] Server certificate length: 1885

[+] Server certificate saved in: server.pem

<---- Handshake()

<---- ServerHelloDone()

-- Alert()

[+] Closing TCP connection!

```
4 0.038276899 172.17.37.100 157.248.221.35 TLSv1.2 120 Client Hello
5 0.67631047 157.240.221.35 172.17.37.100 TCP 66 443 - 42240 [ACK] Seq=1 Ack=55 Win=65536 Len=0 TSval=2489148823 TSecr=4247465819
6 0.07764049 157.240.221.35 172.17.37.100 TLSv1.2 2026 Server Hello
7 0.07764098 157.240.221.35 172.17.37.100 TCP 66 42240 - 443 [ACK] Seq=55 Ack=2761 Win=63480 Len=0 TSval=4247465858 TSecr=2489148823
8 0.07764098 157.240.221.35 172.17.37.100 TLSv1.2 325 Certificate, Server Hello Done
9 0.07764098 157.240.221.35 172.17.37.100 TCP 66 42240 - 443 [ACK] Seq=55 Ack=3080 Win=63328 Len=0 TSval=4247465858 TSecr=2489148823
10 0.07764098 157.240.221.35 172.17.37.100 TLSv1.2 73 Alert (Level: Fatal, Description: Certificate Unknown)
```

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

```bash
Subject: C = US, ST = California, L = Menlo Park, O = "Facebook, Inc.", CN = *.facebook.com
```
Task: TLS getcert

$ ./tls_getcert.py https://amazon.com/
--> ClientHello()
<--- Handshake()
    <--- ServerHello()
        [+] server randomness: 3DA2A106FD0E04AA35F1D380C75F3D82587D9701F18409C9EF8EFE7BC16717FC
        [+] server timestamp: 2002-10-08 12:10:30
        [+] TLS session ID: 6A084FD6533A993DA25612AE4AF6F2B55902D0A0D6239C22BF910F3E72F9018E
        [+ ]Cipher suite: TLS_RSA_WITH_AES_128_CBC_SHA
    <--- Handshake()
    <--- Certificate()
        [+ ]Server certificate length: 2247
    <--- Handshake()
    <--- ServerHelloDone()
--> Alert()
 [+ ] Closing TCP connection!

$ ./tls_getcert.py https://live.com/
--> ClientHello()
<--- Handshake()
    <--- ServerHello()
        [+ ]server randomness: 636A50D428AB87AE9FEBB30EB60F64900E166F72E28560419FD6CD06652ACBD9
        [+ ]server timestamp: 2022-11-08 14:51:32
        [+ ]TLS session ID: 722F00005CECE308E04C05AD094DDD2DA610A7347FA3F8AC93B97A812006D697
        [+ ]Cipher suite: TLS_RSA_WITH_AES_256_CBC_SHA
    <--- Certificate()
        [+ ]Server certificate length: 1989
    <--- ServerHelloDone()
--> Alert()
 [+ ] Closing TCP connection!
Task: TLS getcert

- Use Wireshark to see what bytes are actually sent out over the network
  - Use capture filters 'host amazon.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')
  ```
Task: TLS client (next homework)

Implement a TLS v1.2 client that can obtain an HTTP GET response:

```
$ ./tls_client.py https://127.0.0.1:4433/
--> ClientHello()
<--- Handshake()
    <--- ServerHello()
        [+] server randomness: 60828BA632D28C1E49A5532585E14F6A46390AC448E0F4F0AF99654F3D979BC9
        [+] server timestamp: 2021-04-23 11:56:06
        [+] TLS session ID:
        [+] Cipher suite: TLS_RSA_WITH_RC4_128_SHA
    <--- Handshake()
    <--- Certificate()
        [+] Server certificate length: 554
    <--- Handshake()
        <--- ServerHelloDone()
--> ClientKeyExchange()
--> ChangeCipherSpec()
--> Finished()
<--- ChangeCipherSpec()
<--- Handshake()
    <--- Finished()
--> Application_data()
GET / HTTP/1.0
<--- Application_data()
HTTP/1.0 200 OK
Content-Length: 6

Hello!
[+] Closing TCP connection!
```
Task: TLS client

- Client has to support TLS_RSA_WITH_RC4_128_SHA cipher suite
- Template contains fully implemented PRF(), derive_master_secret(), derive_keys(), encrypt(), decrypt() and client/server Finished hash calculation code
  - Make sure that the correct inputs are provided to these functions (!!!)
- Grading:
  - 2 points if the server accepts your ClientKeyExchange message
  - 2 points if the server accepts your Finished message
  - 2 points if your code can show the HTTP response
- tls_server binary can be used for development (port 4433)
- Wireshark: “Decode As” → “TCP Destination 4433” → “TLS”
$ ./tls_server --port 4433
[+] Connection from 127.0.0.1:36098

--- Handshake()
  ---- ClientHello()
   +[+] version: 0303
   +[+] client random: 60828BA6C11163C5841E3DB1E47178B3F12ED088844DEB424A6DBB2C3415DD4
   +[+] client timestamp: 2021-04-23 11:56:06
   +[+] TLS session ID:
   +[+] Cipher suites:
     [**TLS_RSA_WITH_RC4_128_SHA**]
   +[+] Compression methods:
     null
   +[+] Extensions length: 0

  --> ServerHello()
   +[+] server random: 60828BA632D2B8C1E49A5532585E14F6A46390AC448E0F4F0AF99654F3D979BC9
   +[+] server timestamp: 2021-04-23 11:56:06
   +[+] TLS session ID:
   +[+] Cipher suite: TLS_RSA_WITH_RC4_128_SHA

  --> Certificate()
   +[+] Server certificate length: 554

  --> ServerHelloDone()
  ---- Handshake()
   ---- ClientKeyExchange()
   +[+] PreMaster length: 128
   +[+] PreMaster (encrypted): 61897cea2123d2a50b4811da54f134e54ba12a9fa0ae42cf6c9c6a4cc8406151e6ff3683738522695e37be0574937d472666e9be500a5c97d569a00bd7a9
   +[+] PreMaster: 03039e1742df8e8ab14cbe4fffd13aef7e43a73dab09a6bfc38ec4a3700b2044f50984a56f1005d34325e3473db4b98

  ---- ChangeCipherSpec()
   +[+] Applying cipher suite:
     [+ master_secret = PRF(03039e1742df8e8ab14cbe4fffd13aef7e43a73dab09a6bfc38ec4a3700b2044f50984a56f1005d34325e3473db4b98, "master secret" + 60828BA6C11163C5841E3DB1E47178B3F12ED088844DEB424A6DBB2C3415DD4]
     [+ client_mac_key: 6f30e10a5d2e9128571d46a4f573768282927c
     [+ server_mac_key: e864f54ae6eb29d8167d68af4d49b9d00ee
     [+ client_enc_key: 912da8d4db049e4eed21c3aae43a04
     [+ server_enc_key: b2a9d385372dd9e253c34020c345731d

  ---- Handshake()
   ---- Finished()

  ---- ChangeCipherSpec()

  ---- Finished()

--- Application_data()
GET / HTTP/1.0

--- Application_data()
$ ./tls_client.py https://facebook.com/
--> ClientHello()
<--- Alert()
[-] fatal: 40

$ ./tls_client.py https://twitter.com/
--> ClientHello()
<--- Alert()
[-] fatal: 40

$ ./tls_client.py https://baidu.com/
--> ClientHello()
<--- Handshake()
<--- ServerHello()
[+] server randomness: 91F35EC232AAB56732AED74D4CD27E243E78946392D44F95BD8919B615C14C17
[+] server timestamp: 2047-08-05 22:23:46
[+] TLS session ID: D9C239963DB4654D5444952D44F95BD8919B615C14C17
[+] Cipher suite: TLS_RSA_WITH_RC4_128_SHA
<--- Handshake()
<--- Certificate()
[+] Server certificate length: 1872
<--- Handshake()
<--- ServerHelloDone()
--> ClientKeyExchange()
--> ChangeCipherSpec()
--> Finished()
<--- ChangeCipherSpec()
<--- Handshake()
<--- Finished()
--> Application_data()
GET / HTTP/1.0
<--- Alert()
[-] warning: 0
[+] Closing TCP connection!
Most common pitfalls

• Server fails to verify MAC of client’s Finished message
  • Make sure client’s Finished message is encrypted using the correct keys. Compare keys – if they are different make sure the key derivation receives the correct premaster secret and client and server randomness values.
  • Make sure that MAC is calculated using the TLS record type and not the handshake message type

• Server fails to verify hash in client’s Finished message
  • Make sure all handshake messages sent and received are appended to the handshake_messages variable

• Client fails to verify hash in server’s Finished message
  • Plaintext version of client’s Finished message must be appended to the handshake_messages

• Server returns fatal Alert “decryption failed” after receiving client’s Finished message
  • Make sure the server did not choose a non-RC4 cipher suite