MTAT.07.017
Applied Cryptography

Transport Layer Security (TLS)

University of Tartu

Spring 2021
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)
• 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 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

[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

Frame 4: 120 bytes on wire (960 bits), 120 bytes captured (960 bits) on interface enp0s31f6, id 0
Ethernet II, Src: HewlettP 9d:1f:f0 (a0:8c:fd:9d:1f:f0), Dst: Cisco ff:fc:8c (00:68:e3:ff:fc:8c)
Transport Layer Security
TLSv1.2 Record Layer: Handshake Protocol: Client Hello
  Content Type: Handshake (22)
  Version: TLS 1.2 (6x0303)
  Length: 45
  Handshake Type: Client Hello (1)
  Length: 45
  Version: TLS 1.2 (6x0303)
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]

- TLSv1.2 Record Layer: Application Data Protocol: http-over-tls
- Content Type: Application Data (23)
- Version: TLS 1.2 (0x0303)
- Length: 38
- Encrypted Application Data: f48f13030665f43f3d9a07ba27dcc824c4dc8dfbe9e10
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

TLsv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
Content Type: Handshake (22)
Version: TLS 1.2 (0x0303)
Length: 36
Handshake Protocol: Encrypted Handshake Message
• 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:
  \[
  \text{master\_secret} = \text{PRF}(\text{premaster} + \text{'master secret'} + \text{client\_random} + \text{server\_random}, 48)
  \]

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

• The key block is split into the keys needed:
  \[
  \begin{align*}
  \text{client\_mac\_key} & = \text{key\_block}[:20] \\
  \text{server\_mac\_key} & = \text{key\_block}[20:40] \\
  \text{client\_enc\_key} & = \text{key\_block}[40:56] \\
  \text{server\_enc\_key} & = \text{key\_block}[56:72] \\
  \text{client\_iv} & = \ldots \\
  \ldots
  \end{align*}
  \]
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
Implement a TLS v1.2 client that can retrieve a server’s certificate:

```
$ ./tls_getcert.py https://facebook.com/ --certificate server.pem
--> ClientHello()
<--- Handshake()
    <--- ServerHello()
       [+] server randomness: 14CEF697777BB51D0AFA367E78689AF86152BDF47D8D7C71B0BB6C2AD279FE32
       [+] TLS session ID: 2AF350EFDBC0FEC91D55E68B837C58E86FEA4140632508D7B6C8217F43765757
       [+] Cipher suite: TLS_RSA_WITH_AES_128_CBC_SHA
<--- Handshake()
    <--- Certificate()
       [+] Server certificate length: 1768
       [+] Server certificate saved in: server.pem
<--- Handshake()
    <--- ServerHelloDone()
--> Alert()
[+] Closing TCP connection!
```

```
$ openssl x509 -in server.pem -text | grep 'Subject:'
Subject: C = US, ST = California, L = Menlo Park, O = "Facebook, Inc.", CN = *.facebook.com
```
Task: TLS getcert

$ ./tls_getcert.py https://twitter.com/
--> ClientHello()
<--- Handshake()
    <---- ServerHello()
        [+] server randomness: B5319D1EAF32F2E74B7867D6A871B062FBC5A78DE7E4551E6076A128B30BA7B6
        [+] server timestamp: 2066-05-01 11:19:42
        [+] TLS session ID:
        [+] Cipher suite: TLS_RSA_WITH_AES_128_CBC_SHA
    <--- Handshake()
    <---- Certificate()
        [+] Server certificate length: 1606
    <--- Handshake()
    <---- ServerHelloDone()
--> Alert()
[+] Closing TCP connection!

$ ./tls_getcert.py https://live.com/
--> ClientHello()
<--- Handshake()
    <---- ServerHello()
        [+] server randomness: 608023ADFD820BF1FEEDC57B939263DB5BF2A3F3A4BD2589DEC1FFAB64EDF075
        [+] server timestamp: 2021-04-21 16:07:57
        [+] TLS session ID: D73D00004387C0F7403F8D97241972CDDF7E108D61A1B8E24F48A8FE538E187B
        [+] 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 over the network
  - Use capture filters 'host 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')
  ```
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:
  - 3 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 randomness: 60828BA6C11163C5841E3DB1DE4717B83F12ED088844DEB424A6DBB2C3415DD4
    [*] 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 randomness: 60828BA632D28C1E49A5532585E14F6A46390AC4480F4F0AF99654F3D979BC9
    [*] server timestamp: 2021-04-23 11:56:06
    [*] TLS session ID:
    [*] Cipher suite: TLS_RSA_WITH_RC4_128_SHA

  --> Certificate()
  --> ServerHelloDone()

--- Handshake()
  <--- ClientKeyExchange()
    [*] PreMaster length: 128
    [*] PreMaster (encrypted): ...cd49749acf52739e45ff723a93731d146a2025385e5f61a82f02e655ee8a34fddcf145aa053ca798b3a0cae10427de5e607a9a46ee6c9294c197e8039
    [*] PreMaster: 03039e1742dfc8a134be4f0d13aef7ea43a73db09a6bfc38ec4a3470bo204f4f50984a56f6105d34325e3473db4b98a

--- ChangeCipherSpec()
    [*] Applying cipher suite:
      [*] master_secret = ... + 60828BA6C11163C5841E3DB1DE4717B83F12ED088844DEB424A6DBB2C3415DD4, "master secret" + 60828BA6C11163C5841E3DB1DE4717B83F12ED088844DEB424A6DBB2C3415DD4
      [*] master_secret: ... + 60828BA6C11163C5841E3DB1DE4717B83F12ED088844DEB424A6DBB2C3415DD4
      [*] client_mac_key: 6f0c0e10a15e2e912b571d146a457a5762882827c
      [*] server_mac_key: e854294f546ae69eb29d81617d68af4d459d00ee
      [*] client_enc_key: 912d2a8d6db049e4eed821234b334a04
      [*] server_enc_key: b2a9d386372dd9e253c260e0c589731d

--- Handshake()
  <--- Finished()
    [*] client_verify (received): 6f0c0e10a15e2e912b571d146a457a5762882827c
    [*] client_verify (calculated): 04b3cb0ef05733b0da3c27d6

--- ChangeCipherSpec()
--- Finished()
--- Application_data()
GET / HTTP/1.0
--- Application_data()
RC4 (TLS_RSA_WITH_RC4_128_SHA)

$ ./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: 91F35EC232AAB567:
        [+] server timestamp: 2047-08-05 22:23:46:
        [+] TLS session ID: D9C239963DB4654D544:
        [+] Cipher suite: TLS_RSA_WITH_RC4_128_SHA:
<--- Handshake()
    <--- Certificate()
        [+] Server certificate length: 1753
<--- 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