MTAT.07.017
Applied Cryptography

Transport Layer Security (TLS)
Advanced Features

University of Tartu

Spring 2014
Client usually is authenticated in application level by some shared secret (e.g., password). This can fail:

- Server can be impersonated
- Password can be reused in another service
- Password can be guessed
- Password can be phished
Client Certificate Authentication

- CertificateVerify – signature over all handshake messages
- Can CertificateVerify be reused in another handshake?
- Why CertificateVerify is after ClientKeyExchange?
- Client's Certificate sent before ChangeCipherSpec
- Client proves his identity by signing and not by decrypting
Session Resumption

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

- Any party can initiate negotiation of a new TLS session:
  - Client by sending ClientHello
  - Server by sending HelloRequest
- Handshake messages of the new TLS session are protected by cipher suite negotiated in previous TLS session
- Used by server to renegotiate stronger cipher suite or to request client certificate authentication if on application level client tries to access resources that require such security measures
Certificate request on renegotiation

ClientHello
ServerHello, Certificate, ServerHelloDone
ClientKeyExchange
[ChangeCipherSpec], Finished
[ChangeCipherSpec], Finished
Application Data (GET /auth HTTP/1.1)
HelloRequest
ClientHello
ServerHello, Certificate, CertificateRequest, ServerHelloDone
Certificate, ClientKeyExchange, CertificateVerify
...
Diffie-Hellman Key Exchange

\[ \text{common number} = 2 \]

\[ \text{random number} = 4 \]
\[ 2^4 = 16 \]
\[ 32 \]
\[ 32^4 = 1,048,576 \]

\[ \text{random number} = 5 \]
\[ 2^5 = 32 \]
\[ 16 \]
\[ 16^5 = 1,048,576 \]

- \( (2^5)^4 = 2^{5\cdot4} = (2^4)^5 \)
- In practice is used multiplicative group of integers modulo \( p \)
- Discrete logarithm problem
  - hard to find \( x \), given \( 2^x = 32 \mod p \)
- What to do with key established?
- Secure against passive eavesdropping
- ElGamal, DSA, Elliptic Curve DSA based on DH
**Diffie-Hellman Key Exchange**

- **ServerKeyExchange** contains DH group, server’s DH public key and server’s RSA signature over DH public key, client randomness and server randomness
- **ClientKeyExchange** contains client’s DH public key
- How is “pre-master secret” calculated?
- Used by TLS_(EC)DHE_RSA_WITH_* cipher suites
- Why not to use RSA key exchange directly?
  - Perfect Forward Secrecy
Perfect Forward Secrecy

Benefits:

• Attacker who has compromised RSA private key cannot decrypt previous TLS traffic
• Attacker who has compromised RSA private key has to execute active MITM attack
• Attacker has to crack $x$ keys to decrypt $x$ sessions made to the server

How to apply PFS for eID case?
Extensions

- ClientHello can contain length prefixed extensions
- ServerHello will contain response to client’s extensions
- Most popular extensions:
  - Server Name Indication extension (RFC 3546)
    ```
    Extension: server_name
    Type: server_name (0x0000)
    Length: 17
    Server Name Indication extension
    Server Name list length: 15
    Server Name Type: host_name (0)
    Server Name length: 12
    Server Name: www.eesti.ee
    ```
  - TLS Session Tickets (RFC 5077)
    ```
    Extension: SessionTicket TLS
    Type: SessionTicket TLS (0x0023)
    Length: 180
    Data (180 bytes)
    ```
  - Elliptic Curves (RFC 4492)
    ```
    Extension: elliptic_curves
    Type: elliptic_curves (0x000a)
    Length: 8
    Elliptic Curves Length: 6
    Elliptic curves (3 curves)
    Elliptic curve: secp256r1 (0x0017)
    Elliptic curve: secp384r1 (0x0018)
    Elliptic curve: secp521r1 (0x0019)
    ```
  - Heartbeat (RFC 6520)
Other authentication methods

- **TLS-PSK (RFC 4279)** – pre-shared key
  - PSK identities up to 128 octets in length
  - PSKs up to 64 octets in length
  - \texttt{TLS\_PSK\_WITH\_*}
  - \texttt{TLS\_RSA\_PSK\_WITH\_*}
  - \texttt{TLS\_DHE\_PSK\_WITH\_*}

- **TLS-SRP (RFC 5054)** – low-entropy password
  - Uses discrete logarithms
  - Prevents off-line brute force attacks
  - Password does not have to be stored on server in plaintext
  - User name appended to ClientHello in SRP extension
  - \texttt{TLS\_SRP\_SHA\_WITH\_*}
  - \texttt{TLS\_SRP\_SHA\_RSA\_WITH\_*}
  - \texttt{TLS\_SRP\_SHA\_DSS\_WITH\_*}

- **DH\_anon** – both client and server remain anonymous
  - \texttt{TLS\_DH\_anon\_WITH\_*}
  - No Certificate messages allowed
  - opportunistic encryption (HTTP/2.0)
Task (7 points)

Implement TLS 1.0 client that can obtaining HTTP GET response.

$ python tls_client.py https://127.0.0.1:4433/

--> client_hello()

<--- handshake()

<--- server_hello()

[+] server randomness: 53724A4CEE946A8517927EEF1BB4B2E90B1A0DD4D83BB5B9


[+] TLS session ID:

[+] Cipher suite: TLS_RSA_WITH_RC4_128_SHA

<--- handshake()

<--- certificate()

[+] Server certificate length: 554

<--- handshake()

<--- server_hello_done()

--> client_key_exchange()

--> change_cipher_spec()

--> finished()

<--- change_cipher_spec()

<--- handshake()

<--- finished()

--> application_data()

GET / HTTP/1.0

<--- application_data()

HTTP/1.0 200 OK

Hello!

[+] Closing TCP connection!
Task

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 you provide correct inputs to these functions (!!!)

- Your code should work on eesti.ee and nordea.ee

- Grading:
  - 2 points if a server accepts your ClientKeyExchange message
  - 2 points if a server accepts your Finished message
  - 1 point if your code verifies server’s Finished message
  - 2 points if your code can show HTTP response

- You can use s_server for development (listens on port 4433):
  $ openssl s_server -cert cert.pem -key priv.pem -debug -msg -www

- Or tls_server.py (provides more verbose information)

- Wireshark “Decode As” – “TCP Destination 4433” – “SSL”
$ python tls_server.py --port 4433
[+] Connection from 127.0.0.1:38452

---- client_hello() ----

[+] version: 0301
[+] client randomness: 53723DC0A35CF83D35EB0AD0E37AFD4D9506C4A8040E0BF9EAA3A76185B36F17
[+] TLS session ID:
[+] Cipher suites:
  TLS_RSA_WITH_RC4_128_SHA
[+] Compression methods:
  null
[+] Extensions length: 0

--> server_hello()

[+] server randomness: 53723DC0C05AE242A42175FA40B0ACBE95B8C4116A948776824B3153382BD38
[+] server timestamp: 2014-05-13 18:44:00
[+] TLS session ID:
[+] Cipher suite: TLS_RSA_WITH_RC4_128_SHA

--> certificate()

[+] Server certificate length: 554

--> server_hello_done()

---- client_key_exchange() ----

[+] PreMaster length: 128
[+] PreMaster (encrypted): ... 80c92715401efa4c865bfa99141478f281c4cd2f8488c72760a412f5febf9e510e4bd8f7cd06b18513de93bab67ab56cc07c97dafec17d5be8b0edc1f1
[+] PreMaster: 03019b365e7bd59dca66da37277da790ade7642cec4f643bb89a7a9a8b89eab90f6788b51cd85ab50c3816a436b1e4

---- change_cipher_spec() ----

[+] Applying cipher suite:
  [+] master_secret = PRF(03019b365e7bd59dca66da37277da790ade7642cec4f643bb89a7a9a8b89eab90f6788b51cd85ab50c3816a436b1e4
  [+] master_secret: bbfeb4a6ae093ed38171f980f08a7d840dad77777c8827167335ce724218607f0a03e9a663c250f903e391ed47e4
  [+] client_mac_key: 9980e4d6ea7e433ad4e2be75645318903d942af
  [+] server_mac_key: fcf8e9b9e361b0cc0c547313f53ab9e5f709a
  [+] client_enc_key: 1d4cf122a63930b347547dcb516b21f4
  [+] server_enc_key: 8bc00ba328a5a0df56784ff5bd7f63

---- handshake() ----

[+] finished()

[+] client_verify (received): 8232f05b92f01eb7d8138859
[+] client_verify (calculated): 8232f05b92f01eb7d8138859

--> change_cipher_spec()

[+] finished()

---- application_data() ----

GET / HTTP/1.0

HTTP/1.0 200 OK

Hello!
[+] Closing TCP connection!