Network Security: TLS/SSL

Tuomas Aura
T-110.5241 Network security
Aalto University, Nov-Dec 2012
Outline

1. Diffie-Hellman key exchange (recall)
2. Key exchange using public-key encryption
3. Goals of authenticated key exchange
4. TLS/SSL
5. TLS handshake (!)
6. TLS record protocol
7. TLS trust model
Diffie-Hellman key exchange

Recall from previous lecture
Signed DH with nonces and key confirmation:

1. $A \rightarrow B$: $A, B, N_A, g^x, S_A(\text{“Msg1”}, A, B, N_A, g^x), \text{Cert}_A$
2. $B \rightarrow A$: $A, B, N_B, g^y, S_B(\text{“Msg2”}, A, B, N_B, g^y), \text{Cert}_B, \text{MAC}_{SK}(A, B, \text{“Responder done.”})$
3. $A \rightarrow B$: $A, B, \text{MAC}_{SK}(A, B, \text{“Initiator done.”})$

$SK = h(N_A, N_B, g^{xy})$

Ok to reuse $x$ and $y$
Ephemeral Diffie-Hellman

- Perfect forward secrecy (PFS): session keys and data from past sessions is safe even if the long-term secrets, such as private keys, are later compromised
  - Even participants themselves cannot recover old session keys
  - Called “perfect” for some historical reason; the word means nothing specific

- General principle of implementing PFS: create a new temporary public key pair for each key exchange and afterwards discard the private key

- Common way to implement PFS is ephemeral DH (DHE): both sides use a new DH exponents in every key exchange and delete the exponent values afterwards

- Cost of ephemeral Diffie-Hellman: random-number generation for new x and y, exponentiation for the DH public keys $g^x$ and $g^y$, and cannot cache and reuse previously computed $g^{xy}$

- Typical trade-off: replace DH exponents periodically, e.g. once in a day or hour, and use nonces to create a fresh session key: $SK = h(g^{xy}, N_A, N_B)$
Key exchange using public-key encryption
PK encryption of session key

Public-key encryption of the session key (insecure):

1. A $\rightarrow$ B: A, $PK_A$
2. B $\rightarrow$ A: B, $E_A(SK)$

$SK = $ session key

$E_A(...) = $ encryption with A’s public key
Impersonation and MitM attacks

The protocol again:

1. $A \rightarrow B$: $A, \text{PK}_A$
2. $B \rightarrow A$: $B, E_A(\text{SK})$

$SK = \text{session key}$

$E_A(...)$ = encryption with A’s public key

Same impersonation and man-in-the-middle attacks are possible as in unauthenticated Diffie-Hellman:

$A \rightarrow T(B)$: $A, \text{PK}_A$  // Trent intercepts the message

$T(B) \rightarrow A$: $B, E_A(\text{SK})$  // Trent spoofs the message
Authenticated key exchange with public-key encryption:

1. $A \rightarrow B$: $A, B, N_A, \text{Cert}_A$
2. $B \rightarrow A$: $A, B, N_B, E_A(KM), S_B(“Msg2”, A, B, N_A, N_B, E_A(KM)), \text{Cert}_B, MAC_{SK}(A, B, “Responder done.”)$
3. $A \rightarrow B$: $A, B, MAC_{SK}(A, B, “Initiator done.”)$

$SK = h(N_A, N_B, KM)$  // why not $SK = KM$?
$KM = \text{random key material (random bits) generated by B}$
$\text{Cert}_A = \text{certificate for A’s public encryption key}$
$E_A(...) = \text{encryption with A’s public key}$
$\text{Cert}_B = \text{certificate for B’s public signature key}$
$S_B(...) = \text{B’s signature}$
$MAC_{SK}(...) = \text{MAC with the session key}$

Typically RSA encryption and signatures
Goals of authenticated key exchange
Basic security goals

Create a good session key:
- **Secret** i.e. known only to the intended participants
- **Fresh** i.e. never used before
- **Separation of long-term and short-term secrets**: long-term secrets such as private keys or master keys are not compromised even if session keys are.

Authentication:
- **Mutual** i.e. bidirectional authentication: each party knows with whom it shares the session key
- Sometimes only **unidirectional** i.e. one-way authentication

Optional properties:
- **Entity authentication**: each participant knows that the other is online and participated in the protocol
- **Key confirmation**: each participant knows that the other knows the session key (implies entity authentication)
- **Perfect forward secrecy**: compromise of current secrets should not compromise past session keys
- **Contributory key exchange**: both parties contribute to the session key; neither can decide the session-key alone
Advanced security goals

- **Non-repudiation**: A party cannot later deny taking part (usually not an explicit goal)
- **Plausible deniability**: No evidence left of taking part (usually not an explicit goal either)
- **Integrity check for version and algorithm negotiation**: Increases difficulty of version roll-back attacks
- **Identity protection**: Sniffers cannot learn the identities of the protocol participants. Usually, one side must reveal its identity first, making it vulnerable to active attacks against identity protection
- **Denial-of-service resistance**: The protocol cannot be used to exhaust memory or CPU of the participants. It is not easy to spoof packets that prevent others from completing a key exchange. The protocol cannot be used to flood third parties with data
Protocol engineering

Network is a distributed system with many participants

Computer networking is about protocols
  - Protocol = distributed algorithm
  - Algorithm = stepwise instructions to achieve something

Security is just one requirement for network protocols
  - Cost, complexity, performance, deployability, time to market etc. may override perfect security

Like the design of cryptographic algorithms, security engineering requires experienced experts and peer scrutiny
  - Reuse well-understood solutions such as TLS; avoid designing your own protocols

The most difficult part is understanding the problem
  - Must understand both security and the application domain
  - When the problem is understood, potential solutions often become obvious
TLS/SSL
Originally Secure Sockets Layer (SSLv3) by Netscape in 1995

Originally created to facilitate web commerce:
- Encrypting credit card numbers and passwords on the web
- Fast adoption because built into web browsers

Early resistance, especially in the IETF:
- It was believed that IPSec will eventually replace TLS/SSL
- TLS/SSL was considered bad because it slowed down the adoption of IPSec

Now SSL/TLS is the dominant encryption standard

Standardized as Transport-Layer Security (TLS) by IETF [RFC 5246]
- Minimal changes to SSLv3 implementations but not interoperable
TLS/SSL architecture (1)

- Encryption and authentication layer added to the protocol stack **between TCP and applications**
- End-to-end security between client and server, usually web browser and web server
TLS/SSL architecture (2)

- **TLS Handshake Protocol** — authenticated key exchange
- **TLS Record Protocol** — block data delivery

General architecture of security protocols: authenticated key exchange + session protocol

TSL specifies separate minor “protocols”:
- Alert — error messages
- Change Cipher Spec — turn on encryption or update keys
Cryptography in TLS

Key-exchange mechanisms and algorithms organized in **cipher suites**
- Negotiated in the beginning of the handshake

**Example: TLS_RSA_WITH_3DES_EDE_CBC_SHA**
- **RSA** = handshake: RSA-based key exchange
- Key-exchange uses its own MAC composed of SHA-1 and MD5
- **3DES_EDE_CBC** = data encryption with 3DES block cipher in EDE and CBC mode
- **SHA** = data authentication with HMAC-SHA-1

**TLS mandatory cipher suites:**
- **TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA** (TLS 1.0)
  - **DHE_DSS** = handshake: ephemeral Diffie-Hellman key exchange authenticated with DSS* signatures
- **TLS_RSA_WITH_3DES_EDE_CBC_SHA** (TLS 1.1)
- **TLS_RSA_WITH_AES_128_CBC_SHA** (TLS 1.2)
- SHA-1 and MD5 in the handshake replaced with a negotiable MAC algorithm in TLS 1.2

**Insecure cipher suites:**
- **TLS_NULL_WITH_NULL_NULL**
- **TLS_RSA_EXPORT_WITH_DES40_CBC_SHA**
TLS handshake
**TLS handshake protocol**

- Runs on top of TLS record protocol
- Negotiates protocol version and cipher suite (i.e. cryptographic algorithms)
  - Protocol versions: 3.0 = SSLv3, 3.1 = TLSv1
  - Cipher suite e.g. DHE_RSA_WITH_3DES_EDE_CBC_SHA
- Performs **authenticated key exchange**
  - Several different key exchange mechanisms supported (typically RSA and DHE)
  - Often only the server is authenticated
TLS Handshake (DH)

1. Negotiation
2. Authentication
3. Key exchange
4. Key confirmation
5. Start session

**Client**
- ClientHello
- Certificate*
- ServerKeyExchange*
- CertificateRequest*
- Certificate
- ClientKeyExchange
- CertificateVerify*
- ChangeCipherSpec
- Finished

**Server**
- ServerHello
- Certificate*
- ServerKeyExchange*
- CertificateRequest*
- ServerHelloDone
- ChangeCipherSpec
- Finished
- Application data

Optional, client typically unauthenticated.

Optional signature

Key confirmation

Client D-H key etc.

Encrypted and MAC’ed session data

Server certificate

Server D-H key and signature etc.

Protocol versions, client nonce, cipher suites

Protocol version, server nonce, cipher suite

Protocol versions, client nonce, cipher suites
TLS handshake

1. C → S: ClientHello
2. S → C: ServerHello, Certificate, [ServerKeyExchange], [CertificateRequest], ServerHelloDone
3. C → S: [Certificate], ClientKeyExchange, [CertificateVerify], ChangeCipherSpec, Finished
4. S → C: ChangeCipherSpec, Finished

[Brackets] indicate fields needed only for bidirectional authentication
**TLS_DHE_DSS handshake**

1. **C → S:** Versions, $N_C$, SessionId, CipherSuites
2. **S → C:** Version, $N_S$, SessionId, CipherSuite
   - CertChain$_S$
   - $g, n, g^y, \text{Sign}_S(N_C, N_S, g, n, g^y)$
   - [ Root CAs ]

3. **C → S:** [ CertChain$_C$ ]
   - $g^x$
   - [ Sign$_C$(all previous messages including $N_C, N_S, g, n, g^y, g^x$) ]
   - ChangeCipherSpec
   - MAC$_{SK}$ ("client finished", all previous messages)

4. **S → C:** ChangeCipherSpec
   - MAC$_{SK}$("server finished“, all previous messages)

- `pre_master_secret = g^{xy}`
- `master_secret = SK = h(g^{xy}, “master secret”, N_C, N_S)`

The Finished messages are already protected by the new session keys
**TLS Handshake (RSA)**

1. **Negotiation**
2. **Authentication**
3. **Key exchange**
4. **Key confirmation**
5. **Start session**

**Client**
- ClientHello
- Certificate*
- ClientKeyExchange
- CertificateVerify*
- ChangeCipherSpec
- Finished
- Application data

**Server**
- ServerHello
- ServerHelloDone
- Certificate*
- ServerKeyExchange*
- CertificateRequest*
- ChangeCipherSpec
- Finished
- Application data

Optional, client typically unauthenticated.

Key material encrypted to server

Optional signature

Key confirmation

Protocol version, client nonce, cipher suites

Server certificate (RSA)

Not needed

Encrypted and MAC’ed session data

Protocol versions, server nonce, cipher suite

Signature

Key confirmation

Optional, client typically unauthenticated.
TLS_RSA handshake

1. C → S: Versions, $N_C$, SessionId, CipherSuites
2. S → C: Version, $N_S$, SessionId, CipherSuite
   
   CertChains
   
   [ Root CAs ]

3. C → S: [ CertChain$_C$ ]
   
   $E_S$(pre_master_secret),
   
   [ Sign$_C$(all previous messages including $N_C$, $N_S$, $E_S(\ldots)$) ]
   
   ChangeCipherSpec
   
   MAC$_{SK}$ (“client finished”, all previous messages)

4. S → C: ChangeCipherSpec
   
   MAC$_{SK}$("server finished“, all previous messages)

- $E_S$ = RSA encryption (PKCS #1 v1.5) with S’s public key from CertChain$_S$
- pre_master_secret = random number chosen by C
- master_secret = SK = $h$(pre_master_secret, “master secret”, $N_C$, $N_S$)
- Finished messages are already protected by the new session keys

1. Negotiation
2. RSA
3. Nonces
4. Signature
5. Certificates
6. Key confirmation and negotiation integrity check
Nonces in TLS

Client and Server Random are nonces

Concatenation of a real-time clock value and random number:

```c
struct {
    uint32 gmt_unix_time;
    opaque random_bytes[28];
} Random;
```
**Session vs. connection**

- **TLS session can span multiple connections**
  - Client and server may cache the session state and master_secret
  - Client sends the SessionId of a cached session in Client Hello; otherwise zero
  - Server responds with the same SessionId if session found in server cache; otherwise with a fresh value

- When session is reused, new session keys derived from old master_secret and new nonces

- Change of IP address does not invalidate cached sessions

- **Session tickets [RFC 5077]**:
  - Server can send the session state data (ticket) to the client
  - Client sends the ticket back to the server when reconnecting
  - Ticket is encrypted and authenticated with a secret key know only to the server → client cannot modify data
  - Not implemented by all browsers
TLS renegotiation attack

TLS client or server can initiate “renegotiation handshake” i.e. new handshake to refresh session keys

In 2009, a protocol flaw was found:

1. MitM attacker intercepts a TLS connection from honest client; lets the client in the beginning of the handshake
2. Attacker executes a TLS handshake with the honest server and sends some data to the server over this TLS connection
3. Attacker forwards the (still unauthenticated) connection from the client to the server over its own TLS connection
4. Honest client executes the TLS handshake with the server; server thinks it is renegotiation; attacker loses ability to decrypt the data after ChangeCipherSpec, but continues to forward the connection

What did the attacker achieve? Attacker inserted its own data to the beginning of the connection

- E.g. consider applications where the client sends commands first and finally authorizes them by entering its credentials

Solved by RFC 5746

Surprising because TLS/SSL is one of the most analyzed protocols and no major protocol-level flaws have been found since 1995
TLS record protocol
For write (sending):
1. Take arbitrary-length data blocks from upper layer
2. **Fragment** to blocks of ≤ 4096 bytes
3. **Compress** the data (optional)
4. Apply a **MAC**
5. **Encrypt**
6. Add fragment header (SN, content type, length)
7. Transmit over **TCP server port 443 (https)**

For read (receiving):
- Receive, decrypt, verify MAC, decompress, defragment, deliver to upper layer
Abstract view:
\[ E_{K_1}(\text{data}, \text{HMAC}_{K_2}(\text{SN, content type, length, data})) \]

Different encryption and MAC keys in each direction
- All keys and IVs are derived from the master_secret

TLS record protocol uses 64-bit unsigned integers starting from zero for each connection
- TLS works over TCP, which is reliable and preserves order. Thus, sequence numbers must be received in exact order
TLS Applications

Originally designed for web browsing
- Client typically unauthenticated

New applications:
- Any TCP connection can be protected with TLS
- The SOAP remote procedure call (SOAP RPC) protocol uses HTTP as its transport protocol. Thus, SOAP can be protected with TLS
- TLS-based VPNs
- EAP-TLS authentication and key exchange in wireless LANs and elsewhere

Many of the new applications require mutual authentication
Related reading

- Kaufmann, Perlman, Speciner. Network security, 2nd ed.: chapters 11, 19
Exercises

- Use a network sniffer (e.g. Netmon, Ethereal) to look at TLS/SSL handshakes. Can you spot a full handshake and session reuse? Can you see the lack of identity protection?
- What factors mitigate the lack of identity protection in TLS?
- How would you modify the TLS handshake to improve identity protection? Remember that the certificates are sent as plaintext and SessionId is also a traceable identifier.
- Why do most web servers prefer the RSA handshake?
- Consider removing fields from the TLS DHE and RSA key exchanges. How does each field contribute to security?
- How to implement perfect forward secrecy with RSA?
- Why have the mandatory-to-implement cipher suites in TLS changed over time?
- How many round trips between client and server do the TLS DHE and RSA key exchanges require? Consider also the TCP handshake and that certificates may not fit into one IP packet.
- Why is the front page of a web site often insecure (HTTP) even if the password entry and/or later data access are secure (HTTPS)? What security problems can this cause?
- What problems arise if you want to set up multiple secure (HTTPS) web sites behind a NAT or on a virtual servers that share only one IP address? How does the server name extension (RFC 6066) help?
- If an online service (e.g. webmail) uses TLS with server-only authentication to protect passwords, is the system vulnerable to offline password cracking?