Network Security: TLS/SSL

Tuomas Aura

T-110.5240 Network security
Aalto University, Nov-Dec 2010
Outline

1. Diffie-Hellman
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
Key exchange based on commutative public-key operations
**Diffie-Hellman**

Key exchange based on commutative public-key operations
- Each party has its own secret exponent $x$, $y$
- Each party sends or publishes its own public DH key
- Both compute the same shared secret or key material

Public-key notations: $g^x$, $g^y$, $DH_A$, $DH_B$, $DH-A$, $DH-B$, $PK_B$, $PK_B$

Shared secret notations: $g^{xy}$, $SK$, $K_{AB}$, $K_{DH}$

Needs authentication!
Impersonation attack

- Diffie-Hellman is secure against **passive** attackers
  - Not possible to discover the shared secret by sniffing the network
- Vulnerable to an **active** attack:
  - To A, the attacker can pretend to be B
  - To B, the attacker can pretend to be A

\[
K_{AB} := (g^z)^x
\]

\[
K_{AB} = g^{xz}
\]
Man-in-the-middle (MitM) attack

Attacker pretends to be A to B and B to A

Attacker shares session keys with both A and B and can translate session data between the two “secure” sessions

\[
\begin{align*}
K_{AB} &= (g^z)^x \\
K_{AB} &= g^{xz} \\
K_{AT} &= g^{xz} \\
K_{TB} &= g^{yz} \\
K_{AB} &= (g^z)^{yz} \\
K_{AB} &= g^{yz}
\end{align*}
\]
Alice and Bob notation

Common informal notation for cryptographic protocols
- Alice A, Bob B, Carol C, Trent T, Client C, Server S, Initiator I, Responder R, etc.

Insecure Diffie-Hellman:
- $A \to B$: $A, g^x$
- $B \to A$: $B, g^y$
- $SK = h(g^{xy})$

Man-in-the-middle attack:
- $A \to T(B)$: $A, g^x$  // Trent intercepts the message
- $T(A) \to B$: $A, g^z$  // Trent spoofs the message
- $B \to T(A)$: $B, g^y$  // Trent intercepts the message
- $T(B) \to A$: $B, g^z$  // Trent spoofs the message
Signed Diffie-Hellman

MitM attack can be prevented with authentication of the DH public keys

Signed Diffie-Hellman (insecure):
A → B: A, gx, SA(A, gx), CertA
B → A: B, gy, SB(B, gy), CertB
CertA is a standard public-key certificate, e.g. X.509, where the subject key is A’s public signature key

But what about freshness?
Signed Diffie-Hellman with time stamps (a bit better):
A → B: A, TA, gx, SA(A, TA, gx), CertA
B → A: B, TB, gy, SB(B, TB, gy), CertB
SK = h(TA, TB, gxgy)

Two sources of freshness:
- Time stamps are fresh
- DH exponents may be fresh
Security properties

Signed Diffie-Hellman with time stamps:

A → B: A, T_A, g^x, S_A(A, T_A, g^x), Cert_A
B → A: B, T_B, g^y, S_B(B, T_B, g^y), Cert_B

SK = h(T_A, T_B, g^{xy})

Properties:
- Secret, fresh session key known only by A and B
- Mutual authentication of session key
- Separation between long-term secrets (private signature keys) and short-term secrets (DH exponents and session key)
- Contributory: both contribute to SK and neither side alone can decide SK value

Missing properties:
- Only partial entity authentication: each party knows that the other is alive but not that the other intended to participate in a key exchange between A and B
- No key confirmation: neither party knows whether the other computed the same SK
- Partial non-repudiation of participation: signature proves participation in a key exchange at a certain time, but not with whom
- No identity protection
- No protection against DoS by flood of spoofed first messages
Signed Diffie-Hellman with nonces:

A → B: A, B, N_A, g^x, S_A(A, B, N_A, g^x), Cert_A

B → A: A, B, N_A, N_B, g^x, g^y, S_B(A, B, N_A, N_B, g^x, g^y), Cert_B

A → B: A, B, MAC_{SK}(A, B, “Done.”)

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

Mutual entity authentication and key confirmation requires (at least) three messages

Real protocols are even more complex:

- Version and algorithm negotiation
- DoS protection
- Identity protection
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
- (Not clear why it is called “perfect”)

Common way to implement PFS is ephemeral DH: both sides use a new DH exponents in every key exchange

Cost of forward secrecy:
- Random-number generation for new exponents
- Computation of new DH public keys

Typically DH exponents are replaced every day or hour → PFS only after deleting exponents and session keys
→ nonces required for freshness

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

Signed Diffie-Hellman with nonces:

\[
\begin{align*}
A \to B: & \quad A, B, N_A, g^x, S_A(A, B, N_A, g^x), \text{Cert}_A \\
B \to A: & \quad A, B, N_A, N_B, g^x, g^y, S_B(A, B, N_A, N_B, g^x, g^y), \text{Cert}_B \\
A \to B: & \quad A, B, \text{MAC}_{SK}(A, B, \text{“Done.”}) \\
\end{align*}
\]

\[SK = h(N_A, N_B, g^{xy})\]

Properties:

- Secret, fresh session key known only by A and B
- Mutual authentication of session key
- Separation between long-term secrets (private signature keys) and short-term secrets (DH exponents and session key)
- Mutual entity authentication and key confirmation (requires subtle reasoning!)
- Contributory: both contribute to SK and neither side alone can decide SK value
- Can provide perfect forward secrecy if the DH exponents are not reused

Missing properties:

- Partial non-repudiation: signature proves participation in a key exchange between A and B, but not when
- No identity protection
- No protection against DoS by flood of spoofed first messages
Key exchange using public-key encryption
Public-key encryption of the session key (insecure):

A → B: A, PKₐ
B → A: B, Eₐ(SK)

SK = session key
Eₐ(…) = encryption with A’s public key

Man-in-the-middle attack:

A → T(B): A, PKₐ // Trent intercepts the message
T(A) → B: A, PKₜ // Trent spoofs the message
B → T(A): B, Eₜ(SK) // Trent intercepts the message
T(B) → A: B, Eₐ(SK) // Trent spoofs the message
Authenticated key exchange

(Somewhat realistic protocol)

Public-key encryption of the session key:

A → B: A, B, N_A, Cert_A

B → A: A, B, N_A, N_B, E_A(KM), S_B(A, B, N_A, N_B, E_A(KM)), Cert_B

A → B: A, B, MAC_{SK}(A, B, “Done.”)

SK = h(N_A, N_B, KM)  (why not SK = KM?)

KM = random key material generated by B

Cert_A = certificate for A’s public encryption key

E_A(…) = encryption with A’s public key

Cert_B = certificate for B’s public signature key

S_B(…) = B’s signature

Typically RSA encryption and signatures
Security properties

Public-key encryption of the session key:
A → B: A,B, N_A, Cert_A
B → A: A,B,N_A,N_B,E_A(KM),S_B(A,B, N_A,N_B, E_A(KM)), Cert_B
A → B: A,B, MAC_{SK}(A,B, “Done.”)
SK = h(N_A,N_B,KM)

Properties:
- Secret, fresh session key known only by A and B
- Mutual authentication of session key
- Mutual entity authentication
- Mutual key confirmation (requires subtle reasoning!)
- Separation between long-term secrets (private RSA keys) and short-term secrets (KM, SK)
- Contributory: neither side can decide SK alone

Missing properties:
- Partial non-repudiation of participation of B: signature proves participation in a key exchange with A, but not when; plausible deniability for A
- No perfect forward secrecy
- No identity protection
- No protection against DoS by flood of spoofed first messages
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

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

Optional properties:
- **Entity authentication**: each participant know that the other is online and participated in the protocol
- **Key confirmation**: each participant knows that the other knows the session key
- **Separation of long-term and short-term secrets**: long term secrets such as private keys or shared master keys are not compromised even if session keys are
- **Perfect forward secrecy**: compromise of current secrets should not compromise past session keys
- **Contributory**: 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 fall-back attacks

- **Identity protection:**
  - Outsiders 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
  - The protocol cannot be used to flood third parties with data
  - It is not easy to prevent the participants from completing the protocol
TLS/SSL
Originally Secure Sockets Layer (SSLv3) by Netscape in 1995

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

Early resistance, especially in the IETF:
- IPSec will eventually replace TLS/SSL
- TLS/SSL is bad because it slows 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 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 key exchange replaced with 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**
  - Often only server 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.

- E.g. client D-H key
- Signature
- Key confirmation

Protocol versions, client nonce, cipher suites

Protocol version, server nonce, cipher suite

E.g. server D-H key and signature

Key confirmation

Encrypted and MAC’ed session data

Application data
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 for bidirectional authentication
TLS_DHE_DSS handshake

1. C → S: Versions, $N_C$, SessionId, CipherSuites
2. S → C: Version, $N_S$, SessionId, CipherSuite
   CertChain$_S$
   $g$, $n$, $g^y$, 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$)
- Finished messages are already protected by the new session keys
### TLS_DHE_DSS handshake

1. \( C \rightarrow S: \) Versions, \( N_C \), SessionId, CipherSuites

2. \( S \rightarrow C: \) 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. \( C \rightarrow S: \) [ 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. \( S \rightarrow C: \) ChangeCipherSpec
   - MAC\(_{SK}\) ("server finished", all previous messages)

\[ \text{pre_master_secret} = g^{xy} \]
\[ \text{master_secret} = SK = h(\text{pre_master_secret}, \text{"master secret"}, N_C, N_S) \]

*Finished* messages are already protected by the new session keys

Secret and fresh session key?  
Mutual authentication?  
Entity authentication?  
Key confirmation?  
Perfect forward secrecy?  
Contributory key exchange?  
Non-repudiation or plausible deniability?  
Integrity check for negotiation?  
Identity protection?  
DoS protection?
**TLS_RSA handshake**

1. **C → S:**  
   Versions, $N_C$, SessionId, CipherSuites

2. **S → C:**  
   Version, $N_S$, SessionId, CipherSuite  
   \[ \text{CertChain}_S \]  
   \[ \text{[ Root CAs ]} \]

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

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

- $E_S = \text{RSA encryption (PKCS #1 v1.5) with } S's \text{ public key from } \text{CertChain}_S$
- $\text{pre_master_secret} = \text{random number chosen by } C$
- $\text{master_secret} = SK = h(\text{pre_master_secret, “master secret”, } N_C, N_S)$
- *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**
# TLS_RSA handshake

1. **C → S:** Versions, $N_C$, SessionId, CipherSuites

2. **S → C:** Version, $N_S$, SessionId, CipherSuite
   - CertChain$_S$
     - [ Root CAs ]

3. **C → S:**
   - [ CertChain$_C$ ]
   - $E_S$(pre_master_secret),
   - [ 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)

- $E_S$ = 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 = $h(g^{xy}$, “master secret”, $N_C$, $N_S$)
- *Finished* messages are already protected by the new session keys
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 cache the session state and key
  - Client sends the SessionId of a cached session in Client Hello, otherwise zero
  - Server responds with the same SessionId if found in cache, otherwise with a fresh value

- New master_secret calculated for each connection from old master_secret and fresh nonces
- Change of IP address does not invalidate cached sessions
TLS record protocol
TLS record protocol

For write (sending):
1. Take arbitrary-length data blocks from upper layer
2. **Fragment** to blocks of \( \leq 4096 \) 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
TLS record protocol - abstraction

Abstract view:
\[ E_{K_1}(\text{data}, \text{HMAC}_{K_2}(\text{SN}, \text{content type}, \text{length}, \text{data})) \]

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
TSL trust model
Typical TLS Trust Model

- Trust root:
  - web browsers come with a pre-configured list of root CAs (e.g. Verisign)
    - Users can add or remove root CAs — which do you accept?
  - Root-CA public keys are stored in self-signed certificates
    - Not really a certificate; just a way of storing the CA public keys
- Users usually do not have client certificates
  - Businesses pay a CA to issue a server certificate; client users do not want to pay for certificates
  - Server authentication is typically done with passwords over the server-authenticated HTTPS channel (e.g. web form or HTTP basic access authentication)
Example of a TLS certificate chain: Nationwide (a building society in the UK)

Issuer: VeriSign Class 3 Public Primary CA
Subject: VeriSign Class 3 Public Primary CA

Self-signed certificate in my list of trusted root CAs

Issuer: VeriSign Class 3 Public Primary CA
Subject: CPS Incorp/VeriSign

Certificate chain received in TLS handshake

Issuer: CPS Incorp/VeriSign
Subject: olb2.nationet.com

But how do I know that olb2.nationet.com is the Nationwide online banking site?
TLS Applications

Originally designed for web browsing

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

The web-browser trust model is usually not suitable for the new applications!
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 changes 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? (The server name extension (RFC 4366) provides some help.)
Related reading


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

Dieter Gollmann. Computer Security, 2nd ed.: chapter 13.4