Transport-Layer Security (TLS)

T-110.4206

2009-10-07

Tuomas Aura
Helsinki University of Technology

Outline

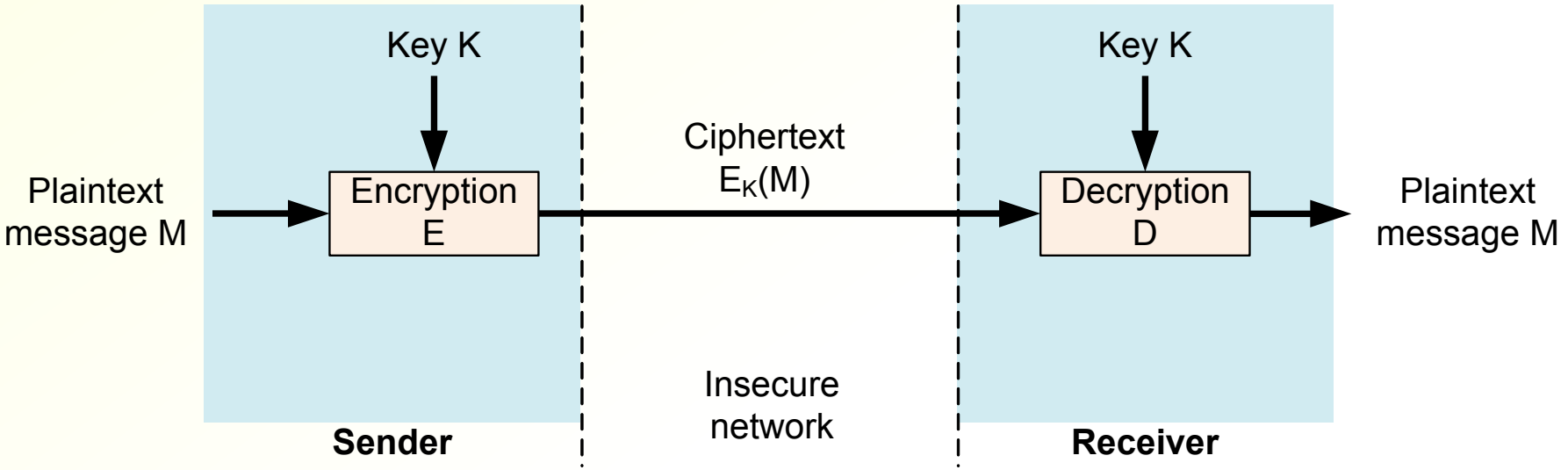
1. Network security protocols
2. Essential cryptography
3. Authenticated key exchange
4. Key exchange using RSA encryption
5. TLS/SSL
6. TLS handshake with RSA
7. TLS record protocol
8. Session reuse
9. Trust model

Network security protocols

- Typical goals for security protocols:
  encryption and authentication of data
- Typical protocol architecture:
  1. Autheticated key exchange produces a session key
  2. Session protocol uses the session key to protect data
- Examples: SSL, TLS, IPsec, HIP, Kerberos, 3G AKA

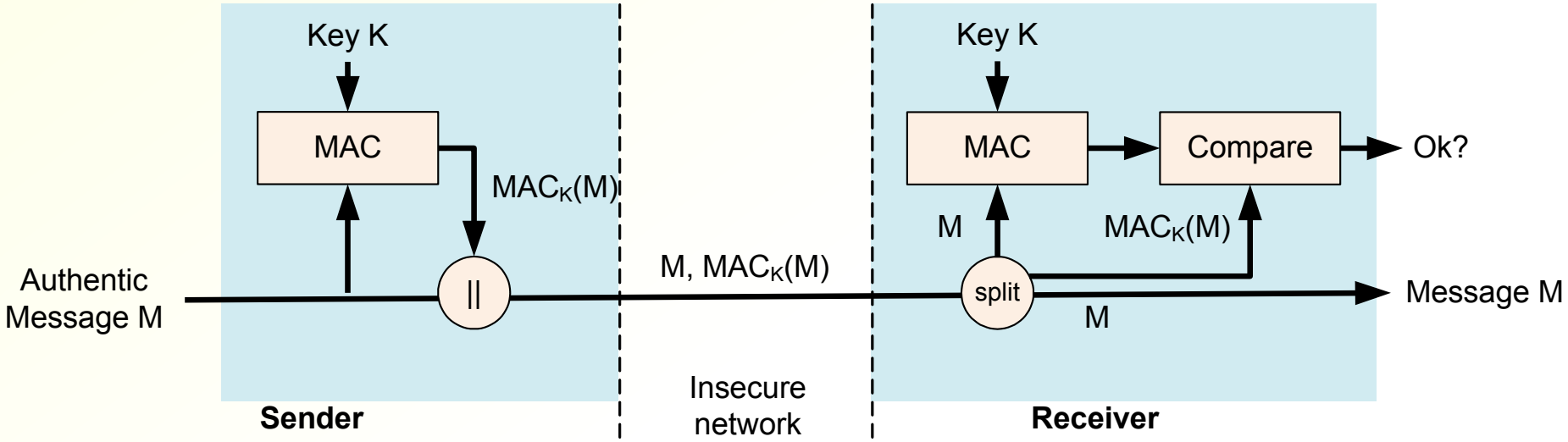
Essential cryptography

Encryption



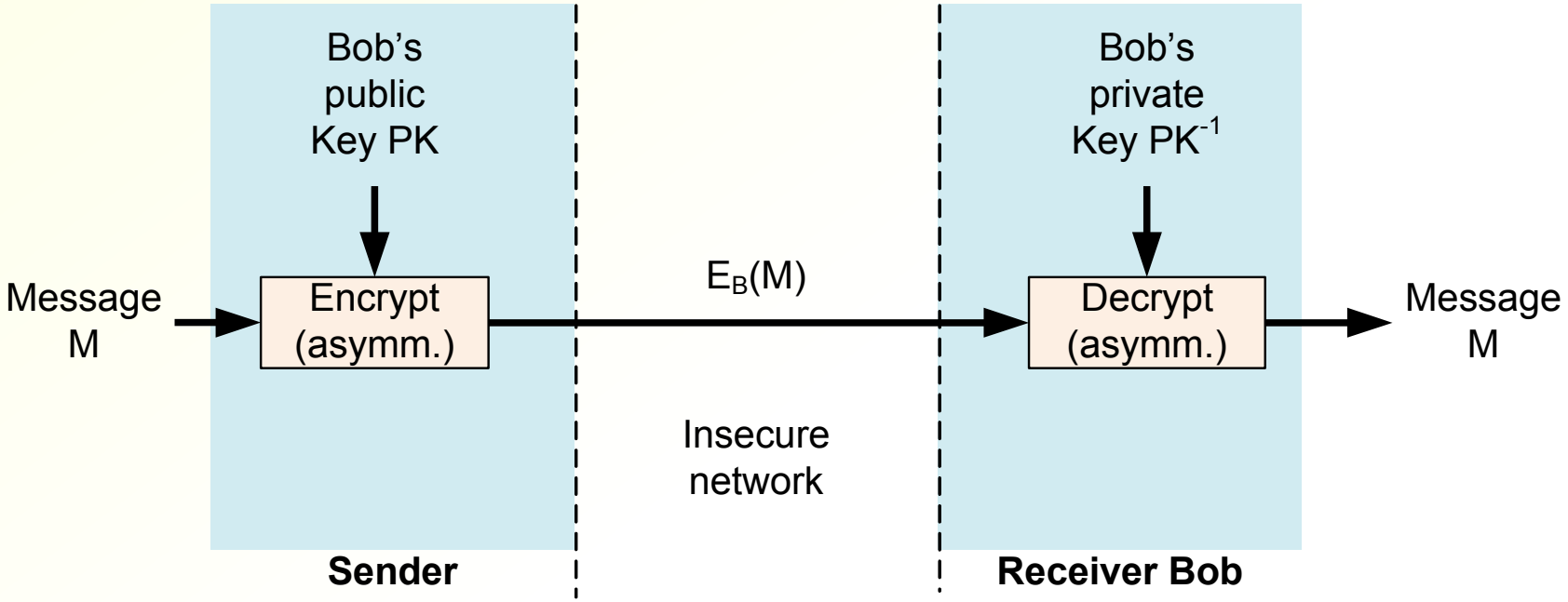
- Message encryption based on symmetric cryptography
  - Endpoints share a secret key K
  - Protects confidentiality of data M

Message authentication code (MAC)



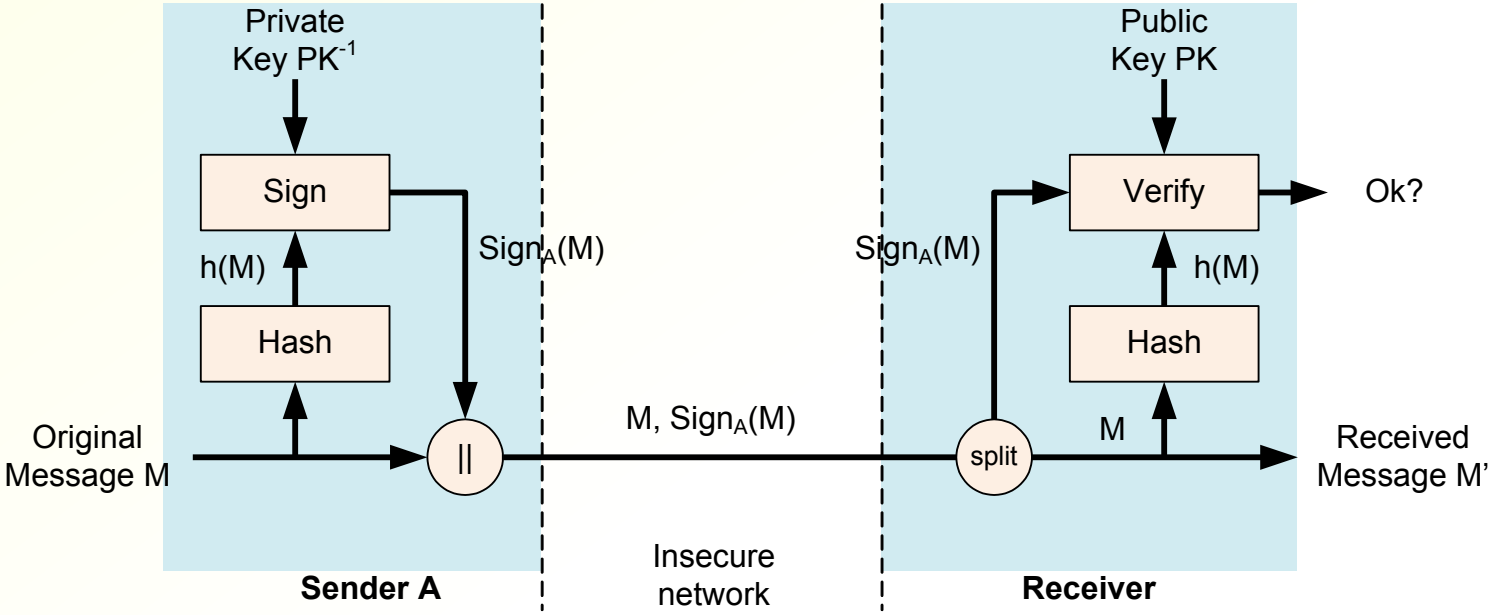
- Message authentication and integrity protection based on symmetric cryptography
  - Endpoints share a secret key K
  - MAC appended to the original message M
  - Examples: HMAC-SHA1, AES CBC-MAC

Public-key encryption



- Message encryption based on asymmetric crypto
  - Key pair: public key and private key
  - Example: RSA

Digital signature (1)



- Message authentication and integrity protection with public-key crypto
  - Verifier has a public key PK ; signer has the private key PK^{-1}
  - Messages are first hashed with a cryptographic hash function and then signed
  - Examples: DSS, RSA + SHA-256

Authenticated key exchange

Basic goals for key exchange

- 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 who it shares the key with
  - One-way i.e. unidirectional authentication: only one party verifies who the other one is

Basic goals for key exchange

- Other good properties, mostly optional:
  - Protection of long-term secrets: long term secrets such as private keys or shared master keys are not compromised even if session keys are
  - 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 (implies entity authentication)
  - Forward and backward secrecy: compromise of all current secrets does not compromise past session keys, and compromise of past session keys does not compromise future session keys (this terminology can used in conflicting ways)
  - Contributory: both parties contribute to the session key; neither can decide the session-key value alone
  - Identity protection: passive observers (sometime also active attackers) cannot learn names of the protocol participants

Key exchange using RSA encryption

Attempt at key exchange v.1

- Public-key encryption of the session key:

  $A \rightarrow B$: A, PK_A

  $B \rightarrow A$: B, $E_A(SK)$

  A,B = names or other identifiers

  PK_A = A's public encryption key

  SK = random session key

  $E_A(\ldots)$ = encryption with A's public key

- Anything wrong?

Man in the middle attack

- The protocol again:

  $A \rightarrow B$: A, PK_A

  $B \rightarrow A$: B, $E_A(SK)$

- Lack of authentication! Man-in-the-middle attack:

  $A \rightarrow T(B)$: A, PK_A // Attacker intercepts the message

  $T(A) \rightarrow B$: A, PK_T // Attacker spoofs the message

  $B \rightarrow T(A)$: B, $E_T(SK)$ // Attacker intercepts the message

  $T(B) \rightarrow A$: B, $E_A(SK)$ // Attacker spoofs the message

Attempt at key exchange v.2

- Authenticated key exchange:

  $A \rightarrow B$: A,B, $Cert_A$

  $B \rightarrow A$: A,B, $E_A(SK)$, $S_B(A,B, E_A(SK))$, $Cert_B$

  SK = random session key

  $Cert_A$ = certificate for A's public encryption key

  $E_A(\ldots)$ = encryption with A's public key

  $Cert_B$ = certificate for B's public signature key

  $S_B(\ldots)$ = B's signature

- Typically implemented with RSA encryption and signatures: the same public works for either
- Man in the middle attack prevented.
  Anything still wrong?

Replay of old session keys

- The protocol again:

  $A \rightarrow B$: $A,B, Cert_A$

  $B \rightarrow A$: $A,B, E_A(SK), S_B(A,B, E_A(SK)), Cert_B$
- Replay attack! Session keys not fresh:

  $A \rightarrow B$: $A,B, Cert_A$

  $B \rightarrow A$: $A,B, E_A(SK), S_B(A,B, E_A(SK)), Cert_B$ // Sniff

  ... // Later

  $A \rightarrow B$: $A,B, Cert_A$

  $T(B) \rightarrow A$: $A,B, E_A(SK), S_B(A,B, E_A(SK)), Cert_B$ // Replay
- Attacker tricks B into accepting the old session key
- We usually assume the attacker may compromise old session keys

Attempt at key exchange v.3

- Authenticated key exchange with freshness:

  $A \rightarrow B$: $A,B, N_A, Cert_A$

  $B \rightarrow A$: $A,B,N_A,N_B,E_A(KM), S_B(A,B, N_A,N_B, E_A(KM)), Cert_B$

  $SK = h(KM|N_A|N_B)$

  KM = random key material

  N_A = random nonce generated by A

  N_B = random nonce generated by B

- Anything still wrong?

Attempt at key exchange v.4

- Authenticated key exchange with freshness and key confirmation:
  $A \rightarrow B$: $A,B, N_A, Cert_A$
  $B \rightarrow A$: $A,B, N_A,N_B, E_A(KM), S_B(A,B, N_A,N_B, E_A(KM)), Cert_B$
  $A \rightarrow B$: $A,B, MAC_{SK}(A,B, \text{“Done.”})$
  $SK = h(KM \mid N_A \mid N_B)$
  KM = random key material generated by B
  N_A = random nonce generated by A
  N_B = random nonce generated by B
  $Cert_A$ = certificate for A’s public encryption key
  $E_A(\ldots)$ = encryption with A’s public key
  $Cert_B$ = certificate for B’s public signature key
  $S_B(\ldots)$ = B’s signature
  $MAC_{SK}(\ldots)$ = message authentication code computed with session key
- Typically implemented with RSA

TLS/SSL

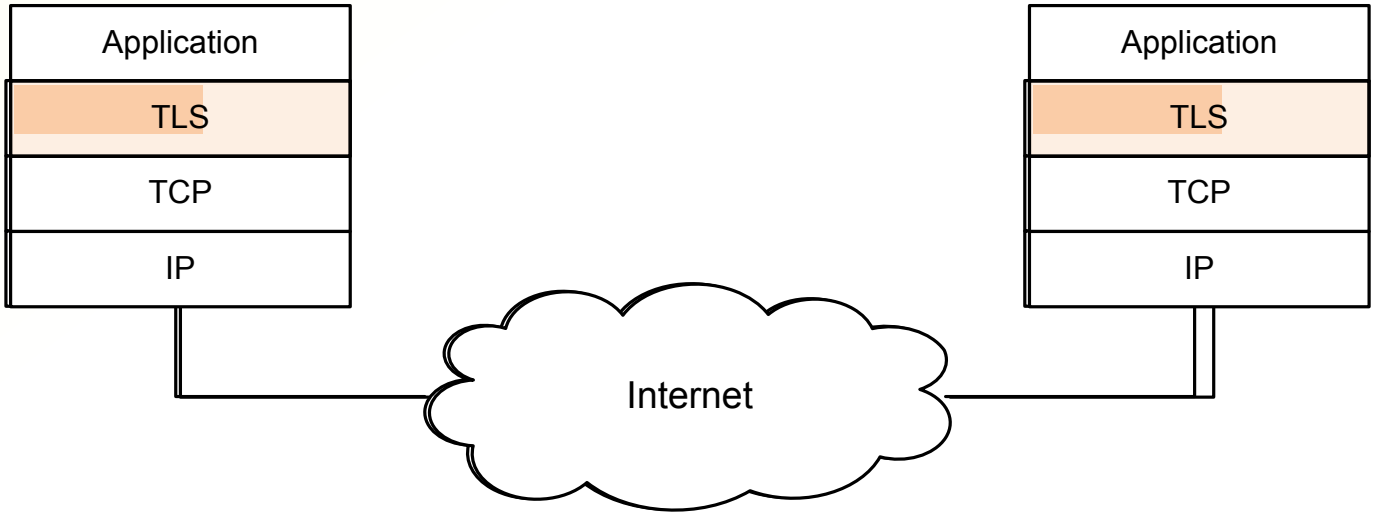
TLS/SSL

- 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 attitudes, especially in the IETF:
  - IPSec will eventually replace SSL
  - SSL is bad because it slows the adoption of IPSec

  Now the dominant encryption standard
- Standardized as Transport-Layer Security (TLSv1) by IETF RFC2246
  - Minimal changes to SSLv3 implementations but not interoperable
  - Latest version TLS 1.2, RFC 5246

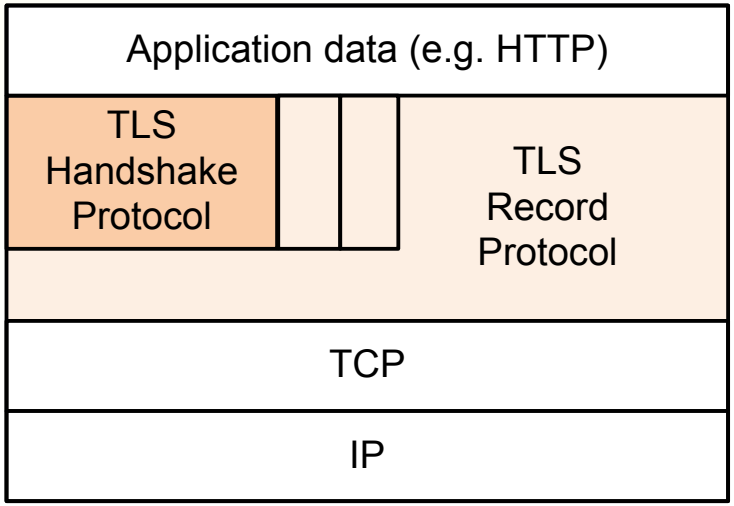
TLS/SSL architecture (1)

- 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.
- Applications use a new TLS API instead of the normal TCP socket API



TLS/SSL architecture (2)

- TLS Handshake Protocol — authenticated key exchange
- TLS Record Protocol — session protocol for protecting data
- Small sub-protocols: Alert (error messages) and Change Cipher Spec



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

Cryptography in TLS

- Many key-exchange mechanisms and algorithm suites defined
- Most widely deployed cipher suite, default in TLS 1.1:
  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 mode and CBC
  - SHA = data authentication with HMAC-SHA-1
- Default cipher suite in TLS 1.0, rarely used in practice:
  TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
  - DHE_DSS = handshake: ephemeral Diffie-Hellman key exchange authenticated with DSS signatures *
- Examples of other cipher suites:

  TLS_NULL_WITH_NULL_NULL
  TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
  TLS_DHE_DSS_WITH_AES_256_CBC_SHA [RFC3269]

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 (one-way i.e. unilateral authentication)

TLS Handshake (RSA)

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

Protocol versions, client nonce, cipher suites

Client | **Server**

ClientHello →

Protocol version, server nonce, cipher suite

← ServerHello
← Certificate*
← ServerKeyExchange*
← CertificateRequest*
← ServerHelloDone

Server certificate

Optional, client typically unauthenticated

Certificate*
ClientKeyExchange
CertificateVerify*
ChangeCipherSpec
Finished →

Encrypted key material

← ChangeCipherSpec
← Finished

Application data ↔ Application data

Encrypted and MAC'ed session 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 optional fields

TLS_RSA handshake

1. C → S: Versions, N_C, SessionId, CipherSuites
2. S → C: Version, N_S, SessionId, CipherSuite
   $CertChain_S$
   [Accepted root CAs]
3. C → S: [$CertChain_C$]
   E_S(pre_master_secret),
   [$Sign_C$(all previous messages including N_C, N_S, E_S(…))]
   ChangeCipherSpec
   MAC_{SK} (“client finished”, all previous messages)
4. S → C: ChangeCipherSpec
   MAC_{SK}("server finished“, all previous messages)

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

- 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

TLS_RSA handshake

1. C → S: Versions, N_C, SessionId, CipherSuites
2. S → C: Version, N_S, SessionId, CipherSuite
   $CertChain_S$
   [Accepted root CAs]
3. C → S: [$CertChain_C$]
   E_S(pre_master_secret),
   [$Sign_C$(all previous messages including N_C, N_S, E_S(...))]
   ChangeCipherSpec
   MAC_{SK} ("client finished", all previous messages)
4. S → C: ChangeCipherSpec
   MAC_{SK}("server finished", all previous messages)

Secret session key?
Fresh session key?
Mutual authentication?
Protection of long-term secrets?
Forward and vackward secrecy?
Entity authentication?
Key confirmation?
Contributory?
Identity protection?

- 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

Nonces in TLS

- Nonces N_C, N_S (client and server random)
- Concatenation of a real-time clock value and random number:

```
struct {
        uint32 gmt_unix_time;
        opaque random_bytes[28];
} Random;
```

TLS record protocol

TLS record protocol

- To write (sending):
  1. Take arbitrary-length data blocks from upper layer
  2. Fragment to blocks of ≤ 4096 bytes
  3. Compress the data (optional)
  4. Append a message authentication code MAC computed with the session key
  5. Encrypt with session key
  6. Add fragment header (sequence number, type, length)
  7. Transmit over TCP server port 443 (https)
- To read (receiving):
  - Receive, decrypt, verify MAC, decompress, defragment, deliver to upper layer

TLS record protocol - abstraction

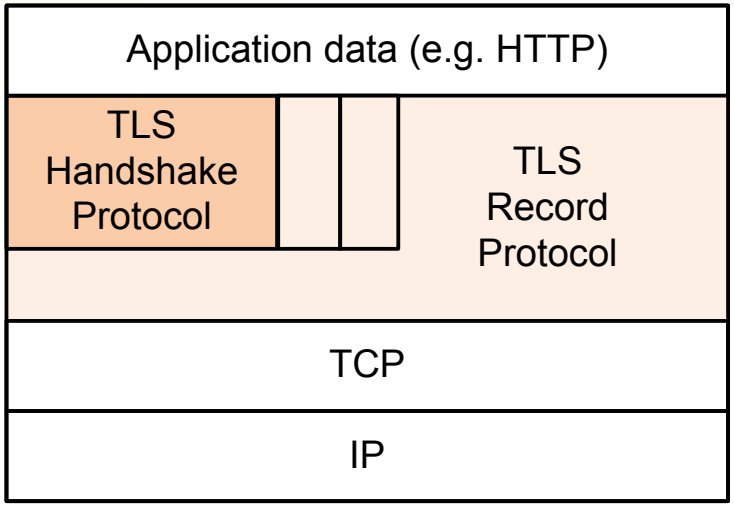
- Abstract view:

  E_{K1} (seq. number, type, length, data,
  $HMAC_{K2}$(seq. number, type, length, data))
- Different encryption and MAC keys in each direction
  - All keys and initialization vectors are derived from the master_secret
- TLS record protocol uses 64-bit sequence numbers 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

Session reuse

Session vs. connection

- TLS works over TCP
- TLS session not bound to IP address or TCP connection; session can span multiple TCP connections
- TCP connection breaks when the client moves and its IP address changes, but TLS session may survive



Session reuse

- TLS session can span multiple connections
  - Client and server cache the session state and master_secret
  - Client sends the SessionId of a cached session in Client Hello; zero if no session
  - Server responds with the same SessionId if found in cache; otherwise with a fresh value
- New master_secret calculated with new nonces for each connection
- Change of IP address does not invalidate cached sessions
- Try which servers support TLS/SSL session reuse: connect to a server with HTTPS, enter your password, log in using password, move to a different IP segment, connect to the same server again; do you need to re-enter the password?
  - Cookies are another way to manage sessions; delete cookies before reconnecting

Trust model

Typical TLS Trust Model

- Trust root: web browsers and operating systems come with a pre-configured list of root CAs (e.g. Verisign)
  - Which root CAs does your browser accept?
  - How do you know the list is not fake?
- Root-CA public keys are stored in self-signed certificates
  - Not really a certificate; just a way of storing the CA public key
- Users usually do not have client certificates
  - Businesses pay a top-level CA to issue a server certificate. Client users do not want to pay
  - Typically, password authentication of the user over server-authenticated TLS (HTTP basic access authentication, or password entered into a web form and POSTed to the server)

TLS Certificate Example

- Example of a TLS certificate chain: Nationwide (a building society in the UK)

Self-signed certificate in the list of trusted root CAs in the browser

Certificate chain received in TLS handshake

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

Trust chain

Root CA self-signed certificate (trust root)

➔ Certificate chain with possible sub-CAs

➔ The server certificate (final certificate in the chain) binds server name to server public key

- Name in the certificate must match the server name in the browser address bar or TLS API call

➔ Server public key public key is used in the authenticated key exchange to authenticate server

➔ Session key

➔ Encryption and authentication of data with the session protocol

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 often not suitable for the new applications!