Network Security:
TLS 1.3 handshake

STUDENTS-HUB.com
Uploaded By: anonymous

Handshake and session protocol

Network security protocols have two parts:

- Handshake = authenticated key exchange that creates symmetric session keys
- Session protocol = encryption and authentication of the session data with the session keys

- Handshake needs a root of trust: PKI (CAs), pre-distributed public keys, or shared master key

TLS 1.3 full handshake

```
Client                                                        Server

ClientHello
+ key_share*
+ signature_algorithms*
+ supported_groups*
+ server_name*
+ certificate_authorities*     -------------->
                                                         ServerHello
                                                         + key_share*
                                                {EncryptedExtensions}
                                                {CertificateRequest*}
                                                       {Certificate*}
                                                 {CertificateVerify*}
                                                           {Finished}
                               <--------------     [ApplicationData*]
{Certificate*}
{CertificateVerify*}
{Finished}                     -------------->
[Application data]             <------------->     [Application data]
```

{encrypted}
[encrypted]
+ extension
* Optional

1. Parameter negotiation

2. DHE or ECDHE key exchange

3. Server authentication

4. Client authentication (typically omitted)

5. Key confirmation

6. Protected session data

STUDENTS-HUB.com

Uploaded By: anonymous

TLS 1.3 full handshake

1. C → S: N_C , supported_versions, supported_groups, signature_algorithms, cipher_suites, server_name, certificate_authorities, g^x

2. S → C: N_S , version, cipher_suite, g^y
   EncryptedExtensions
   $Cert_S$, $Sign_S(TH)$
   $HMAC_{Kfks}(TH)$

   encrypted with K_{shts}

   K_{shts} : server_handshake_traffic_secret

3. C → S: $Cert_C$, $Sign_C(TH)$
   $HMAC_{Kfkc}(TH)$

   encrypted with K_{chts}

K_{chts}: client_handshake_traffic_secret

N_C , N_S = client and server random = nonces

$Cert_C$, $Cert_S$ = certificate chains

TH = transcript hash, i.e., hash of all previous messages

Exchange keys K_{chts}, K_{shts}, K_{fkc}, K_{fks} and session keys K_{cats}, K_{sats} are derived from g^{xy} and TH

K_{cats}: client_application_traffic_secret_N

STUDENTS-HUB.com

Uploaded By: anonymous

TLS 1.3 algorithms

- Small number of modern cipher suites
- AEAD ciphers: encryption and authentication always together
- Perfect forward secrecy required
  - Only ephemeral key exchanges: DHE or ECDHE
  - Old RSA handshake is not supported

STUDENTS-HUB.com

Uploaded By: anonymous

1-RTT handshake

Client

ClientHello
+ key_share*
+ signature_algorithms*
+ supported_groups*
+ server_name*
+ certificate_authorities* --------------->

Client does not know which groups the server supports but makes a guess

Server

ServerHello
+ key_share*
{EncryptedExtensions}
{CertificateRequest*}
{Certificate*}
{CertificateVerify*}
{Finished}
<--------------- [ApplicationData*]

{Certificate*}
{CertificateVerify*}
{Finished} --------------->
[Application data] <--------------> [Application data]

STUDENTS-HUB.com

Uploaded By: anonymous

1-RTT handshake

- TLS 1.3 handshake causes only one round-trip delay
  - Client can send HTTP request (application data) right after client Finished
  - TLS 1.2 and most other key-exchange protocols require **two RTT**
  - Important for page load times in web browsing
- However, TCP + TLS 1.3 together cause 2-RTT latency
  - QUIC avoids this because it runs over UDP
- Sometimes TLS 1.3 handshake takes two RTT:
  - If server does not support the group of `key_share` in `ClientHello`, server sends `HelloRetryRequest` to ask for a different curve
  - DTLS server under DoS attack can send a Cookie in HelloRetryRequest

Key derivation

Inputs to key derivation:

1. PSK (external PSK or resumption PSK)
2. DHE/ECDHE secret
3. Transcript of handshake messages, up to the point where the key is derived

one or both, as available

Keys:

- client_early_traffic_secret → used to derive AEAD keys for early data in 0-RTT (...)
- client/server_handshake_traffic_secret → used to derive AEAD keys for handshake messages {...} and Finished HMAC keys
- client/server_application_traffic_secret_N → used to derive AEAD encryption keys for post-handshake application data and messages [...]
- resumption_master_secret and ticket_nonce → derive resumption PSK
- exporter_master_secret → used to create keys for the application layer

STUDENTS-HUB.com

Uploaded By: anonymous

References

- TLS 1.3, RFC 8446
- The New Illustrated TLS Connection, https://tls13.ulfheim.net/
- **A Readable Specification of TLS 1.3**
  https://www.davidwong.fr/tls13/

Uploaded By: anonymous

Exercises

- Use a network sniffer (e.g., tcpdump, Wireshark) to look at TLS handshakes. Can you spot a full handshake and session resumption? Can you see the plaintext server name indication (SNI)?
- Compare TLS 1.3 and TLS 1.2 handshakes in network trace: Can you see the difference is round-trips, identity protection?
- How would you modify the TLS 1.3 handshake to improve identity protection? Learn about Protected Extensible Authentication Protocol (PEAP). How does PEAP protect the client identity?
- Consider removing different message fields from the handshake. How does each message field contribute to security?
- Why have the supported and mandatory-to-implement cipher suites in TLS changed over time?
- Why did most web servers for a long time prefer the RSA handshake?
- One reason why the RSA handshake it is no longer supported in TLS 1.3 is that it does not provide PFS. Is it possible to implement PFS without Diffie-Hellman?
- Finds applications that could benefit significantly from the 0-RTT handshake. Is there any cost to deploying it?
- What problems arise if you want to set up multiple secure (HTTPS) web sites behind a NAT or on virtual servers that share one IP address? How to TLS 1.3 and TLS 1.2 solve this issue?
- If an online service (e.g., webmail) uses TLS with server-only authentication to protect passwords, is the system vulnerable to offline password cracking?

STUDENTS-HUB.com

Uploaded By: anonymous

TLS 1.3 full handshake

1. C → S: N_C , supported_versions, supported_groups, signature_algorithms, cipher_suites, server_name, certificate_authorities, g^x

2. S → C: N_S , version, cipher_suite, g^y
EncryptedExtensions
$Cert_S$, $Sign_S(TH)$
$HMAC_{Kfks}(TH)$
} encrypted with K_{shts}

3. C → S: $Cert_C$, $Sign_C(TH)$
$HMAC_{Kfkc}(TH)$
} encrypted with K_{chts}

Which security properties?

- Secret, fresh session key
- Mutual or one-way authentication
- Entity authentication, key confirmation
- Perfect forward secrecy (PFS)
- Contributory key exchange
- Downgrading protection
- Identity protection
- Non-repudiation
- Plausible deniability
- DoS resistance

$Cert_C$, $Cert_S$ = certificate chain
TH = transcript hash i.e. hash of all previous messagas
Exchange keys K_{chts}, K_{shts}, K_{fkc}, K_{fks} session keys K_{cats}, K_{sats} derived from g^{xy} and TH

STUDENTS-HUB.com

Uploaded By: anonymous

Identity protection?

- Client sends server name indication (SNI) and CAs in plaintext
  – SNI needed to have multiple server names at one IP address
- Server certificates are encrypted against passive sniffing
  – However, anyone can get them from server by connecting to it and sending the right SNI
- Client certificates (if used) are encrypted
  – Protected also against server impersonation

Summary: server identity leaked; client identity well protected

Network Security:
TLS 1.3 PSK and session resumption

STUDENTS-HUB.com
Uploaded By: anonymous

Outline

- Recall TLS 1.3 full handshake
- Pre-shared key (PSK) mode
- Session resumption

STUDENTS-HUB.com

Uploaded By: anonymous

TLS 1.3 full handshake

```
Client                                                      Server

ClientHello
+ key_share*
+ signature_algorithms*
+ supported_groups*
+ server_name*
+ certificate_authorities*    -------------->
                                                       ServerHello
                                                      + key_share*
                                             {EncryptedExtensions}
                                             {CertificateRequest*}
                                                    {Certificate*}
                                              {CertificateVerify*}
                                                        {Finished}
                              <--------------  [ApplicationData*]
{Certificate*}
{CertificateVerify*}
{Finished}                    -------------->
[Application data]            <------------->   [Application data]
```

{encrypted}
[encrypted]
+ extension
* Optional

1. Parameter negotiation

2. DHE or ECDHE key exchange

3. Server authentication

4. Client authentication (typically omitted)

5. Key confirmation

6. Protected session data

The slides from CS-E4300 - Network Security at Aalto University

STUDENTS-HUB.com

Uploaded By: anonymous

Pre-shared key (PSK) mode

```
Client                                                Server
ClientHello
+ key_share*
+ pre_shared_key
(Application Data*)        -------------->
                                                 ServerHello
                                            + pre_shared_key
                                                + key_share*
                                       {EncryptedExtensions}
                                                  {Finished}
                           <--------------  [ApplicationData*]
(EndOfEarlyData)
{Finished}                 -------------->
[Application data]         <--------------  [Application data]
```

Client identities

Selected client identity

Recommended:
DHE or ECDHE for PFS

Authentication
based on PSK

STUDENTS-HUB.com

Uploaded By: anonymous

Pre-shared key (PSK) mode

1. C → S: N_C , g^x, ClientIdentity
2. S → C: N_S , g^y, $HMAC_{Kfks}(TH)$, early data
3. C → S: $HMAC_{Kfkc}(TH)$

- Mutual authentication based on a pre-established identity and session key (*external PSK*)
  - PSK = pre-established shared key between C and S
  - HMAC keys K_{fks} and K_{fkc} in for the Finished message are derived from PSK, g^{xy} and TH; and so are the session keys

TLS 1.3 session resumption (1)

```
Client                                                    Server

ClientHello
+ key_share*
+ signature_algorithms*
+ supported_groups*
+ server_name*
+ certificate_authorities*    -------------->        ServerHello
                                                     + key_share*
                                            {EncryptedExtensions}
                                            {CertificateRequest*}
                                                   {Certificate*}
                                             {CertificateVerify*}
                                                       {Finished}
                              <--------------  [ApplicationData*]
{Certificate*}
{CertificateVerify*}
{Finished}                    -------------->
                              <--------------  [NewSessionTicket]
[Application data]            <------------->  [Application data]
```

Server packages the session state into an encrypted data called session ticket and sends it to the client

STUDENTS-HUB.com

Uploaded By: anonymous

TLS 1.3 session resumption (2)

Client identities =
one or more session tickets

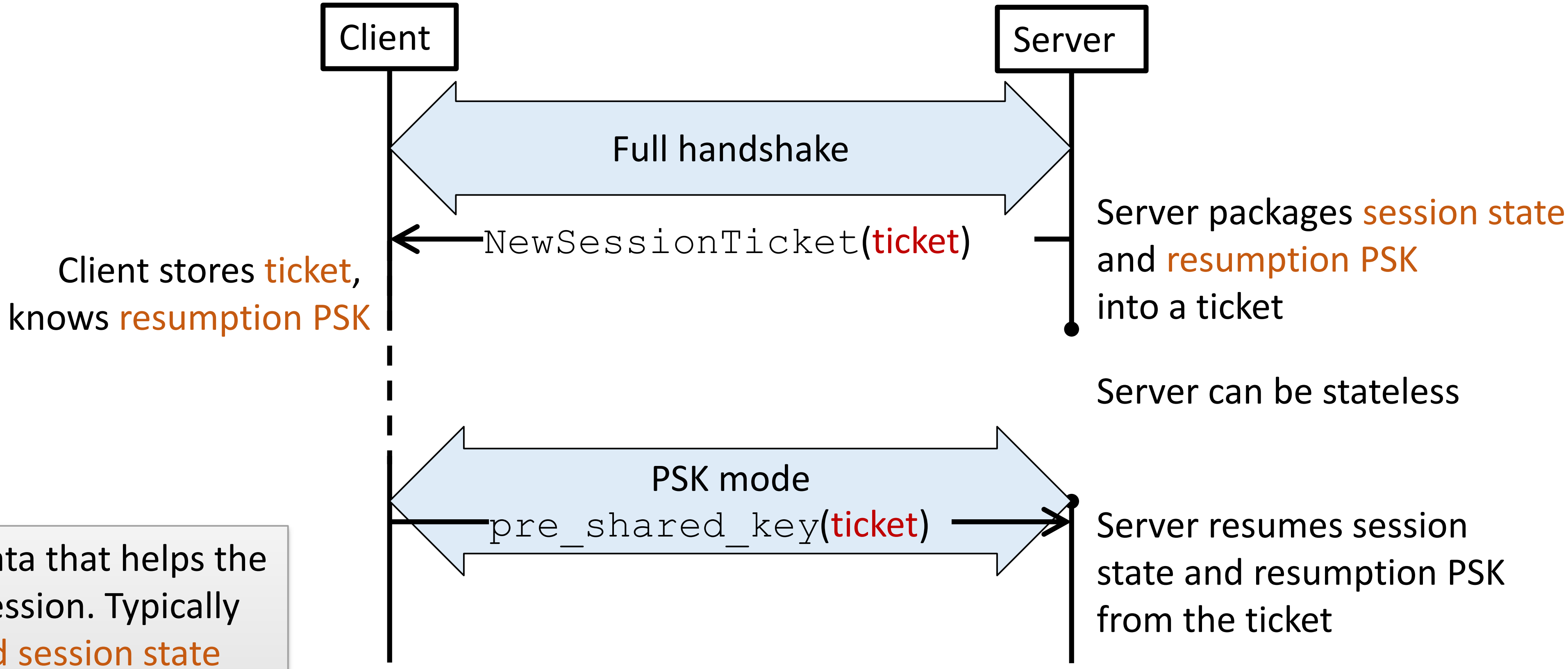
Selected session ticket

Recommended:
DHE or ECDHE for PFS

```
Client                                                      Server
ClientHello
+ key_share*
+ pre_shared_key
(Application Data*)          -------------->
                                                       ServerHello
                                                  + pre_shared_key
                                                      + key_share*
                                             {EncryptedExtensions}
                                                        {Finished}
                             <--------------     [ApplicationData*]
(EndOfEarlyData)
{Finished}                   -------------->
[Application data]           <--------------     [Application data]
```

STUDENTS-HUB.com
The slides from CS-E4300 - Network Security at Aalto University
Uploaded By: anonymous

TLS 1.3 session resumption timeline



Ticket = opaque data that helps the server recall the session. Typically contains encrypted session state and resumption PSK. Only the server itself can decrypt the tickets that has created

STUDENTS-HUB.com

Uploaded By: anonymous

TLS 1.3 session resumption uses

- TLS 1.3 session resumption = PSK mode handshake with ticket as client identity and resumption key as the PSK
  - Currently the main purpose of the PSK mode
- When useful?
  - Server does not want to store the TLS sessions over idle periods
  - If client is authenticated with smartcard, avoids repeated user action
  - Mobile clients keep changing their IP address and need frequent reconnection
  - Resume the session with a different server instance in the cloud

STUDENTS-HUB.com

Uploaded By: anonymous

Key derivation

Inputs to key derivation:

1. PSK (external PSK or resumption PSK)
2. DHE/ECDHE secret
3. Transcript of handshake messages, up to the point where the key is derived

one or both, as available

Keys:

- client_early_traffic_secret → used to derive AEAD keys for early data in 0-RTT (…)
- client/server_handshake_traffic_secret → used to derive AEAD keys for handshake messages {…} and Finished HMAC keys
- client/server_application_traffic_secret_N → used to derive AEAD encryption keys for post-handshake application data and messages […]
- resumption_master_secret and ticket_nonce → derive resumption PSK
- exporter_master_secret → used to create keys for the application layer

STUDENTS-HUB.com

Uploaded By: anonymous

TLS 1.3 session resumption and identity

```
Client                                                  Server
ClientHello
+ key_share*
+ pre_shared_key
(Application Data*)          -------------->
                                                   ServerHello
                                              + pre_shared_key
                                                  + key_share*
                                         {EncryptedExtensions}
                                                    {Finished}
                             <--------------   [ApplicationData*]
(EndOfEarlyData)
{Finished}                   -------------->
                             <--------------   [NewSessionTicket]
[Application data]           <-------------->  [Application data]
```

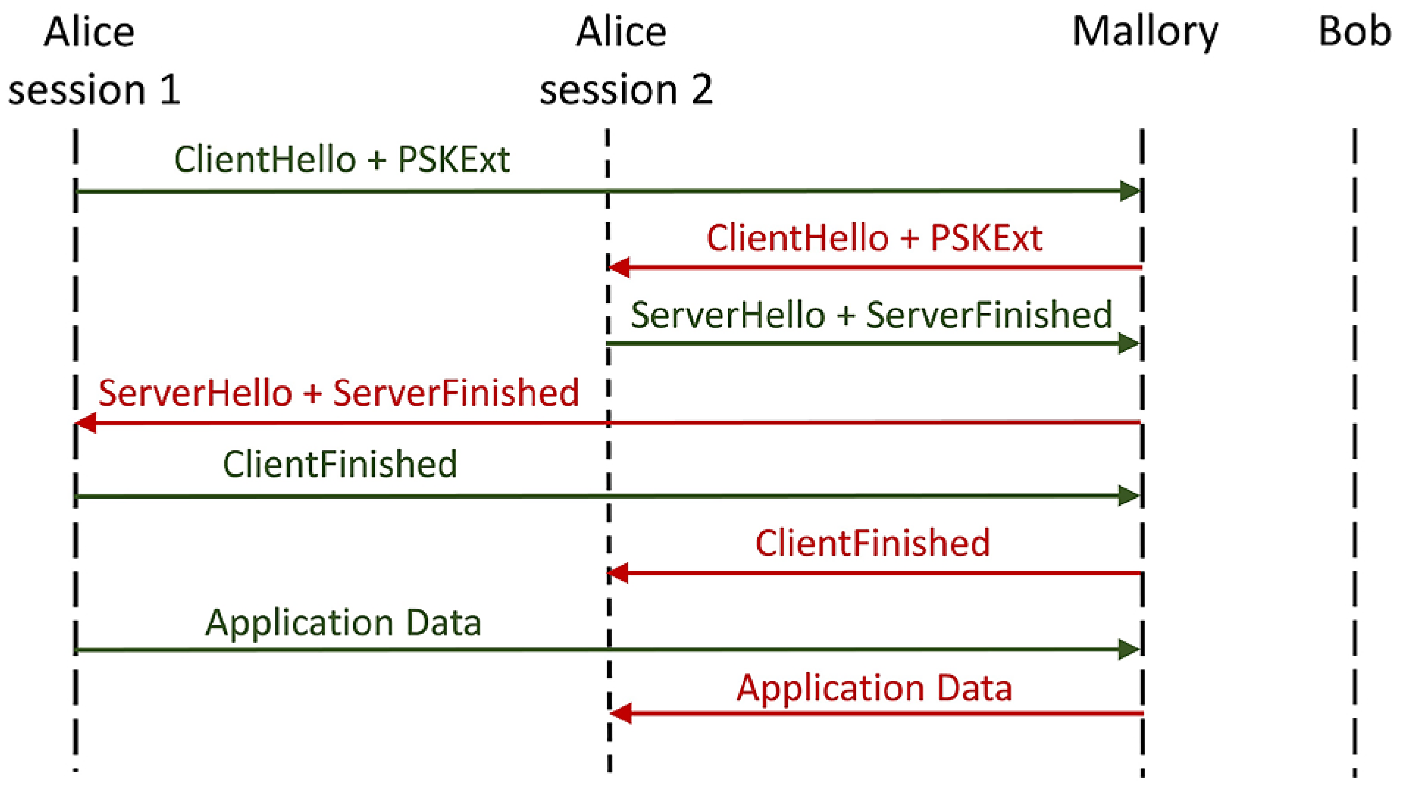
Server can refresh the ticket for PFS and for protecting client identity

STUDENTS-HUB.com

Uploaded By: anonymous

“Selfie attack”

- Reflection attack against external (out-of-band) PSK
  - Trick the client to connect to itself
  - Assumes the same entity can be both client and server
- PSK used mistakenly as a group key for two parties
  - Group key only authenticates the group, not the individual
- Solution: Use different PSK for each direction
  - For each PSK, Alice is either the client or server, never both for the same PSK



[Nir Drucker & Shay Gueron, Selfie: reflections on TLS 1.3 with PSK, 2019]

STUDENTS-HUB.com

Uploaded By: anonymous

Network Security:
TLS 1.3 0-RTT handshake

STUDENTS-HUB.com
Uploaded By: anonymous

TLS 1.3 session resumption

```
Client                                                Server
ClientHello
+ key_share*
+ pre_shared_key
(Application Data*)          -------------->
                                                 ServerHello
                                            + pre_shared_key
                                                + key_share*
                                       {EncryptedExtensions}
                                                  {Finished}
                             <--------------  [ApplicationData*]
(EndOfEarlyData)
{Finished}                   -------------->
[Application data]           <--------------  [Application data]
```

Client identities = one or more session tickets

Selected session ticket

Recommended: DHE or ECDHE for PFS

STUDENTS-HUB.com

Uploaded By: anonymous

0-RTT handshake

Client can send early data right after the first message

Example: HTTP GET request

Server can respond in the second message

```
Client                                              Server
ClientHello
+ key_share*
+ pre_shared_key
(Application Data*)          -------------->
                                                ServerHello
                                           + pre_shared_key
                                               + key_share*
                                      {EncryptedExtensions}
                                                 {Finished}
                             <--------------   [ApplicationData*]
(Application Data*)
(Application Data*)
(EndOfEarlyData)
{Finished}                   -------------->
[Application data]           <--------------   [Application data]
```

STUDENTS-HUB.com

Uploaded By: anonymous

Key derivation

Inputs to key derivation:

1. PSK (external PSK or resumption PSK)
2. DHE/ECDHE secret
3. Transcript of handshake messages, up to the point where the key is derived

} one or both, as available

Keys:

- client_early_traffic_secret → used to derive AEAD keys for early data in 0-RTT (…)
- client/server_handshake_traffic_secret → used to derive AEAD keys for handshake messages {…} and Finished HMAC keys
- client/server_application_traffic_secret_N → used to derive AEAD encryption keys for post-handshake application data and messages […]
- resumption_master_secret and ticket_nonce → derive resumption PSK
- exporter_master_secret → used to create keys for the application layer

STUDENTS-HUB.com

Uploaded By: anonymous

0-RTT handshake

- With session resumption or PSK, client can send application data (early data) right after ClientHello
  - Lower latency for web browsing and APIs. However, TCP handshake in the underlying transport layer still takes one RTT
- Serious security limitations:
  - Early data is vulnerable to replay attacks (no fresh server nonce yet)
  - No PFS for the early data
- Ok for idempotent requests (mainly HTTP GET) that do not require long-term secrecy
- Application must explicitly enable 0-RTT
  - TLS layer cannot decide when the lower security of 0-RTT is acceptable

STUDENTS-HUB.com

Uploaded By: anonymous

Network Security:
RSA handshake (TLS 1.2 and earlier)

STUDENTS-HUB.com
Uploaded By: anonymous

Public-key encryption of session key

- Public-key encryption of the session key:

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

PK_A = A's public key

SK = session key

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

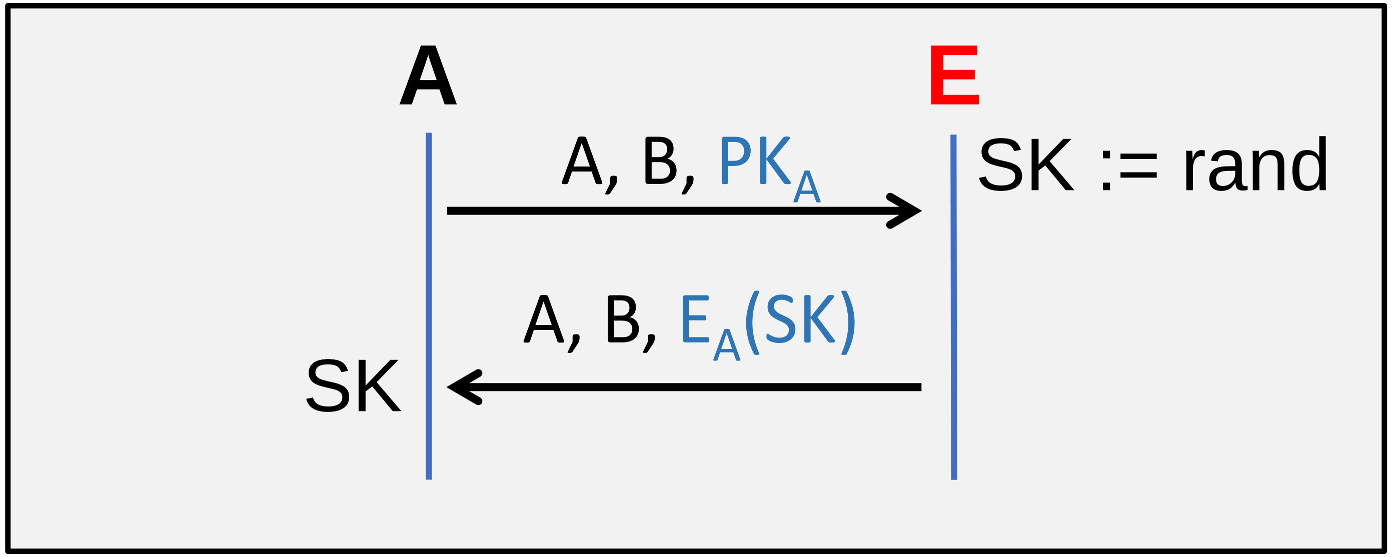
Note:
The protocol is not secure like this. Please read further.

STUDENTS-HUB.com

Uploaded By: anonymous

Impersonation and MitM attacks

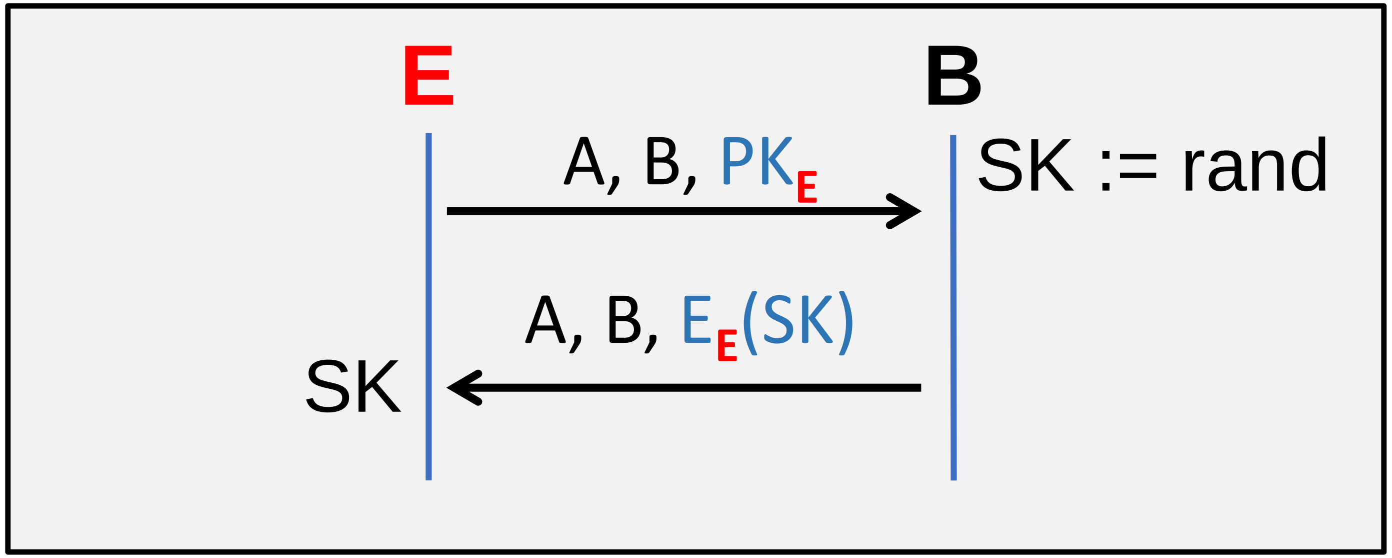
- Unauthenticated key exchange with public-key encryption suffers from the same impersonation and man-in-the-middle attacks as DH



- A has a shared secret, but with whom?

Impersonation and MitM attacks

- Impersonating A is similarly possible because B does not know whether the public key really belongs to A:



- B has a shared secret, but with whom?

Authenticated key exchange

- Authenticated key exchange with public-key encryption:

1. $A \rightarrow B$: $A, B, N_A, Cert_A$
2. $B \rightarrow A$: $A, B, N_B, E_A(KM), S_B(\text{"Msg2"}, A, B, N_A, N_B, E_A(KM)), Cert_B,$
   $MAC_{SK}(A, B, \text{"Responder done."})$
3. $A \rightarrow B$: $A, B, MAC_{SK}(A, B, \text{"Initiator done."})$

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

Why nonces and not SK = KM?

Somewhat realistic protocol (compare with TLS_RSA)

KM = random key material (random bits) generated by B
$Cert_A$, $E_A(\ldots)$ = A's certificate and public-key encryption to A
$Cert_B$, $S_B(\ldots)$ = B's certificate and signature
$MAC_{SK}(\ldots)$ = MAC with the session key

To match with the previous slide:
A = Server, B = Client

TLS_RSA handshake

```
Client                                                 Server
ClientHello
                          -------------->
        1. Parameter negotiation
                                                  ServerHello
                    2. Server certificate        Certificate*
                                           CertificateRequest*
                          <--------------      ServerHelloDone
Certificate*
ClientKeyExchange    3. RSA encryption of key material
CertificateVerify*   4. Client authentication with
ChangeCipherSpec        signature (typically omitted)
Finished
                          -------------->
                     5. Key confirmation
                                             ChangeCipherSpec
                          <--------------            Finished
[Application data]        <------------->  [Application data]
```

6. Protected session data

STUDENTS-HUB.com

Uploaded By: anonymous

TLS_RSA handshake

1. C → S: Versions, N_C, SessionId, CipherSuites
2. S → C: Version, N_S, SessionId, CipherSuite
   $Cert_S$, [Root CAs]
3. C → S: [$Cert_C$]
   E_S(pre_master_secret),
   [$Sign_C$(all previous messages including)]
   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 $Cert_S$
pre_master_secret = random byte string chosen by C
master_secret = h(pre_master_secret, "master secret", N_C, N_S)

STUDENTS-HUB.com

Uploaded By: anonymous

TLS_RSA handshake

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

Which security properties?
- Secret, fresh session key
- Mutual or one-way authentication
- Entity authentication, key confirmation
- Perfect forward secrecy (PFS)
- Contributory key exchange
- Downgrading protection
- Identity protection
- Non-repudiation
- Plausible deniability
- DoS resistance

E_S = RSA encryption (PKCS #1 v1.5) with S’s public key from $Cert_S$
pre_master_secret = random byte string chosen by C
master_secret = h(pre_master_secret, “master secret”, N_C, N_S)

STUDENTS-HUB.com

Uploaded By: anonymous