GIS 05: SSH & TLS

Guillermo Navarro-Arribas

Dep. of Information and Communications Engineering
Universitat Autonoma de Barcelona

v. 20210419

Contents

This document is given as a teaching material for the subject *Garantia de la Informació i Seguretat* for the degree on Computer Engineering at the Universitat Autònoma de Barcelona.

The document focuses on the description of the main security features provided by the SSH and TLS protocols.

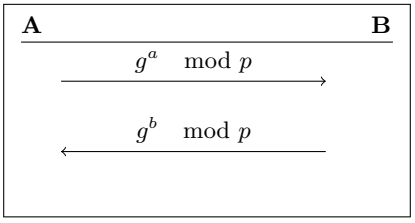
1 Preliminaries: Diffie-Hellman

Diffie-Hellman (DH) is a **key exchange** algorithm [12]. It allows two parties to establish a shared secret. It is based on the difficulty of the discrete logarithm problem.

Suppose A and B want to use DH to establish a shared secret to be used as a symmetric key.

- Let p be a prime, let g be a generator (for any $x \in \{1, 2, \ldots, p-1\}$ there is n such that $x = g^n \mod p$).
- A selects private value a.
- B selects private value b.

A and B exchange the following values:



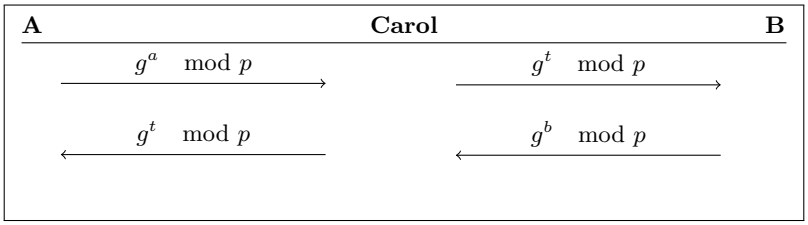
Both can then compute $K = g^{ab} \mod p$ as a shared secret. This shared secret is commonly used to derive session keys in most secure communication protocols.

Note that:

- An attacker can see $g^a \mod p$ and $g^b \mod p$ but cannot compute K. Note that $g^a g^b \mod p = g^{a+b} \mod p \neq g^{ab} \mod p$.
- If the attacker gets a or b he/she can get K.

1.1 Man-in-the-middle in DH

It is well known that DH is vulnerable to a man-in-the-middle (MITM) attack. Suppose Carol can place herself between A and B, and can capture messages from both. Carol can use a random value t and perform the following attack:



After that, A computes the secret as $K_A = g^{at} \mod p$, and B as $K_B = g^{bt} \mod p$. Carol can compute both values and thus perform a MITM attack on the communications, without A and B noticing her manipulation.

This is an very important issue to consider when using DH within general protocols. DH does not provide authentication and thus it is important to provide some additional mechanism when using DH in order to prevent such attacks. In general we can say that the goal of DH is to perform a key exchange and not to provide any type of authentication.

1.2 Elliptic-curve Diffie-Hellman

Elliptic-curve Diffie–Hellman (ECDH) is a variant of DH which uses elliptic-curve cryptography. We will not get into details to describe ECDH, interested readers can check [13, 6, 8, 12].

1.3 Ephemeral Diffie-Hellman and forward secrecy

Some protocols (e.g. TLS) refer to so called Ephemeral Diffie-Hellman (usually denoted as DHE). Here, the term *ephemeral* is used to denote that the DH values used by both parties are not reused. That is, every time the DH protocol is executed, a new shared secret is established. In DHE the keys obtained from the shared secret are considered temporary and are never reused, nor stored for future used. Client and server will generate a new DH parameter for each session.

This opposes to the fixed or static DH where one or both of the secret values (random values) used by either party is reused. E.g. a server could always use the same value to perform the DH key exchange.

The use of DHE guaranties **forward secrecy** (or *perfect forward secrecy. PFS*), which is usually defined as follows.

> A cryptographic protocol has **perfect forward secrecy (PFS)** if the compromise of long-term keys does not allow an attacker to obtain past session keys. [9]

In terms of DH, this means that the compromise of any secret part o session key will not compromise previous session keys. In other words the compromise of a key now does not compromise past communications (performed by the same parties). If unique session keys and DH secret values are generated for each session, it is clear the PFS is guaranteed. If an attacker can get the current session key she can decrypt the current session, but she will not be able to decrypt previous sessions.

Note that, although TLS 1.2 did support both ephemeral and non-ephemeral key exchange algorithms, version 1.3 only supports ephemeral DH. Non ephemeral approaches were removed from TLS 1.3 in order to ensure forward secrecy.

2 Secure Shell Protocol

The Secure Shell (SSH) provides a secure tunnel over insecure networks. SSH version 2 was adopted as standard by the **IETF** (see Table 1), and its most popular implementation is OpenSSH (open source). Some of its main uses are:

- Remote login
- Remote command execution
- Remote X forwarding
- Secure file transfer (SCP, SFTP, ...)
- ...

RFC 4250	Protocol Assigned Numbers [7].
RFC 4251	Protocol Architecture [16].
RFC 4252	Authentication Protocol [14].
RFC 4253	Transport Layer Protocol [17].
RFC 4254	Connection Protocol [15].
RFC 4255	Key Fingerprints [11].
RFC 4256	Generic Message Exchange Authentication [3].
RFC 4344	Transport Layer Encryption Modes [1].
RFC 4716	Public Key File Format [4].

Table 1: SSH in RFCs

2.1 SSH Protocol Architecture

SSH consists of three principal protocols or components (see Figure 1):

- **Transport Layer Protocol** (RFC 4253): algorithms and parameters negotiation, session key exchange, session ID establishment, and server authentication.
- **User Authentication Protocol** (RFC 4252): client authentication.
- **Connection Protocol** (RFC 4254): flow control, remote command execution, port forwarding, X forwarding, ...

We will focus on the transport and user authentication protocols, which are the ones providing the most common security features of SSH.

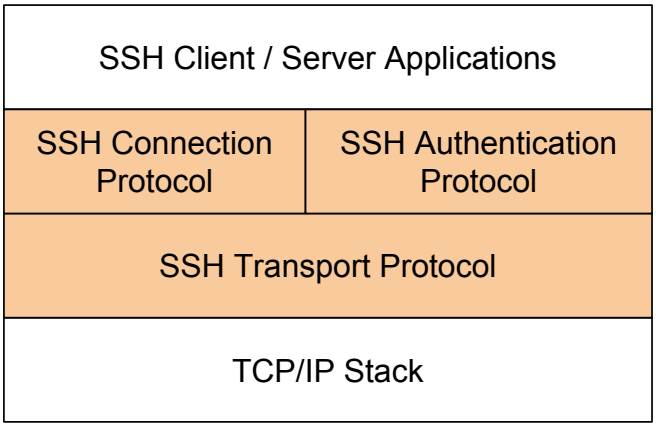


Figure 1: SSH Protocol Architecture

2.2 SSH Transport Layer

The SSH Transport layer protocol mainly provides:

- Connection setup:
  - should work over a transport protocol protecting against transmission errors (e.g. TCP).
  - client initiates the connection (server listens on port TCP 22).
- Binary packet format.
- Key exchange (and crypto negotiation).
- Confidentiality and integrity.
- Server authentication.

2.2.1 SSH Host key

Each server has a **host key**: an asymmetric public key. A host may have multiple host keys for different algorithms.

The host key is typical RSA, DSA, or ECDSA key. It is usually stored in `/etc/ssh/ssh_host_*_key`, where `*` is the public key algorithm: `rsa`, `dsa`, `ecdsa`, ...

A client authenticates the server by its host key. That is, the authentication process ensures that the server is the owner of the private key corresponding to the given host key. The client thus, needs to have the correspondence between the host identifier and its host key.

2.2.2 Server authentication

How does the client authenticate the server?: SSH support two main alternatives (**trust models**):

1. **Client database**: the client maintains a database (usually a text file `.ssh/known_hosts`) with each host name and its public host key (actually, a hash of the public). Here, the name of the host is its identifier as typed by the user, e.g. it might be a full DNS name, alias, or IP address.
   - no centrally administered infrastructure;
   - no need for third-party coordination;
   - database is a key point (maintenance and security).
   - common configuration allows the user to accept, and store the key, if the key is not in the database.
   - protocol has an option for NOT checking host key when connecting for the first time.
   - Sometimes this approach is known as **key continuity** model (knowing that what you get now is what you had before: "Starbucks 'coffee' is the same no matter which country you are in").
2. **CA-based**: A CA certifies host keys. Client knows the CA root key to verify certificates of host name-key of accepted CAs.
   - eases maintenance (for the clients, since there is no need to maintain the database).
   - need for trusted third party (CA) and related PKI infrastructure.

2.2.3 SSH transport packet

← 4 bytes →	← 1 byte →	← var. →	← 4 - 255 bytes →	← mac len. →
packet length	padding len.	payload	rand. padding	MAC

Table 2: SSH binary packet

The SSH transport packet is a binary package with the format shown in Table 2. It includes

- *packet length*: (bytes) not including *MAC* and *packet length* itself;
- *padding length*: (bytes) length of *random padding*;
- *payload*: content (can be compressed if compression is negotiated);
- *random padding*: padding so total length is multiple of the cipher block size;
- *MAC*: Message Authentication Code (see Section 2.2.4).

2.2.4 SSH integrity: MAC

Integrity is provided by a Message Authentication Code (MAC), which uses a HMAC (keyed-hash message authentication code) as:

MAC = HMAC(key, sequence num. ‖ unencrypted packet)

where:

- key: integrity key (see Section 2.2.6).
- sequence num.: packet sequence number, 0 in the first packet and increases after every packet (never reset again during connection).
- unencrypted packet: whole packet without MAC.
- "‖" denotes concatenation.

The concrete HMAC used is negotiated in the handshake. Common supported HMAC algorithms are: `hmac-sha1`, `hmac-sha1-96`, `hmac-md5`, ...

2.2.5 Simplified SSH handshake

A simplified version of the SSH handshake initiated by the client includes the negotiation of cryptographic parameters and Diffie-Hellman key exchange:

$$C \to S : ID_C, IC = (CP, R_C) \quad (1)$$
$$S \to C : ID_S, IS = (CS, R_S) \quad (2)$$
$$C \to S : g^c \mod p \quad (3)$$
$$S \to C : g^s \mod p,\ PK_S,\ Sig_{SK_S}(H) \quad (4)$$

Where:

- ID_A: identifier (username) of user A.
- CP: crypto proposed; CS = crypto selected
- R_A: random nonce from user A
- PK_S: server host (public) key.
- $H = h(ID_C,\ ID_S,\ IC,\ IS,\ PK_S,\ g^c \mod p,\ g^s \mod p,\ g^{cs} \mod p)$
- $Sig_{SK_S}(H)$: H signed by S (with its private key SK_S)

In the simplified protocol:

(1) C identifies herself and sends information regarding the cryptographic parameters preferred by the client, and a random nonce R_C. Cryptography parameters selection (CP, and CS in the previous protocol description) include a list of the following algorithm identifiers:
   - kex_algorithms
   - server_host_key_algorithms
   - encryption_algorithms_client_to_server
   - encryption_algorithms_server_to_client
   - mac_algorithms_client_to_server
   - mac_algorithms_server_to_client

  - compression_algorithms_client_to_server
  - compression_algorithms_server_to_client

(2) S selects cryptographic parameters and returns his selection, with his nonce R_S.
(3) C sends her Diffie-Hellman value.
(4) S sends his Diffie-Hellman value, PK_S, and the signature of H.

Here we can see that the use of the signature on H provides the authentication in the protocol. With this signature the client can authenticate the server using either the client database or the CA-based models (see Section 2.2.2).

2.2.6 SSH session keys

The key exchange from the handshake using DH yields:

- K: a shared secret from the DH exchange.
- H: (exchange hash) acts as the session identifier (unique for the whole connection).

From these values both client and server obtain the session keys. There are 6 keys obtained as shown in Table 3.

Initial IV $C \rightarrow S$:	$h(K \parallel H \parallel "A" \parallel session\_id)$
Initial IV $S \rightarrow C$:	$h(K \parallel H \parallel "B" \parallel session\_id)$
Encryption key $C \rightarrow S$:	$h(K \parallel H \parallel "C" \parallel session\_id)$
Encryption key $S \rightarrow C$:	$h(K \parallel H \parallel "D" \parallel session\_id)$
Integrity key $C \rightarrow S$:	$h(K \parallel H \parallel "E" \parallel session\_id)$
Integrity key $S \rightarrow C$:	$h(K \parallel H \parallel "F" \parallel session\_id)$

Table 3: SSH session keys derivation

The *session_id* is the first H value. H will change if there is another key exchange (key re-exchange is usually performed after each GB of transmitted data or each hour of connection time).

2.3 SSH Authentication Protocol

The SSH Authentication Protocol allows to authenticate the client with 3 different authentication methods and "none", which does not provide authentication of the client:

- "none"
- Public key
- Password
- Host based

The authentication protocol messages always follow the same structure:

$$C \rightarrow S : ID_C, \text{authn. method, parameters} \quad (5)$$
$$S \rightarrow C : \text{OK / FAIL} \quad (6)$$

That is the client sends a message with the corresponding authentication method parameters, and the server answers with a success or failure message. A failure message can include a list of other authentication methods to be used. I can also be used to indicate partial success when requiring more than 1 different authentication method.

In the following sections we show a simplified version of each authentication method.

2.3.1 Public key authentication

$$C \rightarrow S : ID_C, PK_C, Sig_{SK_C}(ID_C, PK_C) \quad (7)$$
$$S \rightarrow C : \text{SUCCESS / FAILURE} \quad (8)$$

- This is the only required authentication methods in SSH.
- PK_C is the public key of C. It can be a certificate, $cert_C$.
- Authentication is based on the possession of the corresponding private key.
- Server must verify the that the key is a valid authenticator and that the signature is valid.
- PK_C is usually stored in server in `$HOME/.ssh/authorized_keys`

2.3.2 Password authentication

$$C \rightarrow S : ID_C, \text{<password>} \quad (9)$$
$$S \rightarrow C : \text{SUCCESS / FAILURE} \quad (10)$$

- The password is transmitted in cleartext (recall that encryption is provided by the transport protocol).
- The server can request a change of password (e.g. if expired) with a change requests response:

$$S \rightarrow C : \text{SSH\_MSG\_USERAUTH\_PASSWD\_CHANGEREQ} \quad (11)$$
$$C \rightarrow S : ID_C, \text{<old password>, <new password>} \quad (12)$$
$$S \rightarrow C : \text{SUCCESS / FAILURE} \quad (13)$$

2.3.3 Host-based Authentication

$$C \to S : ID_C, \text{hostname}, PK_C, cert_C, \\ Sig_{SK_C}(ID_C, \text{hostname}, PK_C, cert_C) \quad (14)$$

$$S \to C : \text{SUCCESS / FAILURE} \quad (15)$$

- PK_C is the public client **host key** (with SK_C being the corresponding private key); $cert_C$ certificate for client host.
- Server must verify: client host key belong to the hostname in the message, that user (ID_C) is allowed to log in (if required), and signature is valid.
- Server can also verify that the hostname corresponds to the network address sending the message (this is recommended but not required by RFC 4252).
- Server config:
  - `/etc/shosts.equiv`: list of host that can use client host-based authentication;
  - `/etc/ssh/ssh_known_hosts`: public keys of hosts listed in `shosts.equiv`

3 Transport Layer Security

SSL (Secure Socket Layer) was a protocol designed in 1995 by *Netscape Communications* to provide security on the WWW. After SSL version 3.0, TLS 1.0 was developed as an upgrade to SSL. Now SSL is completely deprecated and the current version of TLS at the moment is TLS 1.3 (although TLS 1.2 is still widely used). TLS 1.3 is defined in RFC 8446 [10].

TLS is implemented between the application layer (HTTP, FTP, NNTP, ...) and the transport layer in the TCP/IP stack. The transport protocol must be reliable (e.g. TCP).

TLS main goal is to establish a secure channel between two peers, client and server, providing: authentication, confidentiality, and integrity. It provides two principal components:

- **Handshake protocol**: performs cryptographic negotiation, authentication, session keys establishment, ...
- **Record protocol**: transmits data with the parameters and properties determined by the handshake.

We will focus here on the handshake protocol, and more precisely, on how it provides authentication and key exchange. We will not get into other interesting topics such as the whole PKI involved in supporting TLS for the Web.

3.1 TLS Handshake Protocol

The TLS handshake provides messages to perform different types of authentication and key exchange procedures, which can be adapted to different situations.

In its basic form, the handshake provides session key agreement through DH and authentication of the server. A very simplified version is shown in Figure 2.

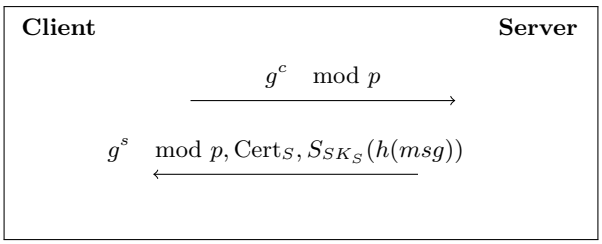


Figure 2: (Very) Simplified TLS handshake

The client randomly chooses the secret value c and the server chooses s. Then the client initiates a DH exchange. Note that the response message from the server includes the server certificate $Cert_S$ and the signature made with the server's private key of *h(msg)*. We will consider *h(msg)* to be the hash of all previous exchanged messages in the handshake. This signature is sent, so the client can actually authenticate the server. Not surprisingly it is very similar to the simplified handshake of SSH (see Section 2.2.5).

The client needs also to verify the server certificate. To do that we usually relay in a PKI. The actual model is not provided by the specification. In the case of the Web, it follows a trusted list model, where the Web browser has a list trusted CAs. The browser (client) will accept any valid certificate rooted in one of such CAs.

This simplified version of the handshake can be extended to support different authentication key exchange modes.

TLS 1.3 supports 2 main authentication options:

1. Client authenticates the server: the most common case.
2. Client authenticates the server, and the server authenticates the client: this case is usually denoted as mutual authentication.

The key exchange can be performed using ephemeral Diffie-Hellman (DHE) or by a pre-shared-key (PSK), or by both. In the PSK exchange both client and server share a common secret previous to the handshake. The key exchange modes in TLS 1.3 are:

- (EC)DHE: ephemeral Diffie-Hellman over either finite fields (DHE) or elliptic curves (ECDHE).
- PSK-only
- PSK with (EC)DHE

We will not get into details of how the PSK key exchange works and will focus on the DH approach.

3.2 Full TLS 1.3 handshake

The full handshake includes a set of messages that can be exchanged between the client and the server. Some of them are optional or condition dependent. Its use will depend on the authentication options and key exchange mode used.

Figure 3 shows all possible messages. There, "*[*]*" denotes an optional message (it might not be sent if there is no need for it). The whole handshake can be performed in 3 steps. Note that each step contains all messages to be sent from client to server or vice versa, since they can be sent on one single network packet.

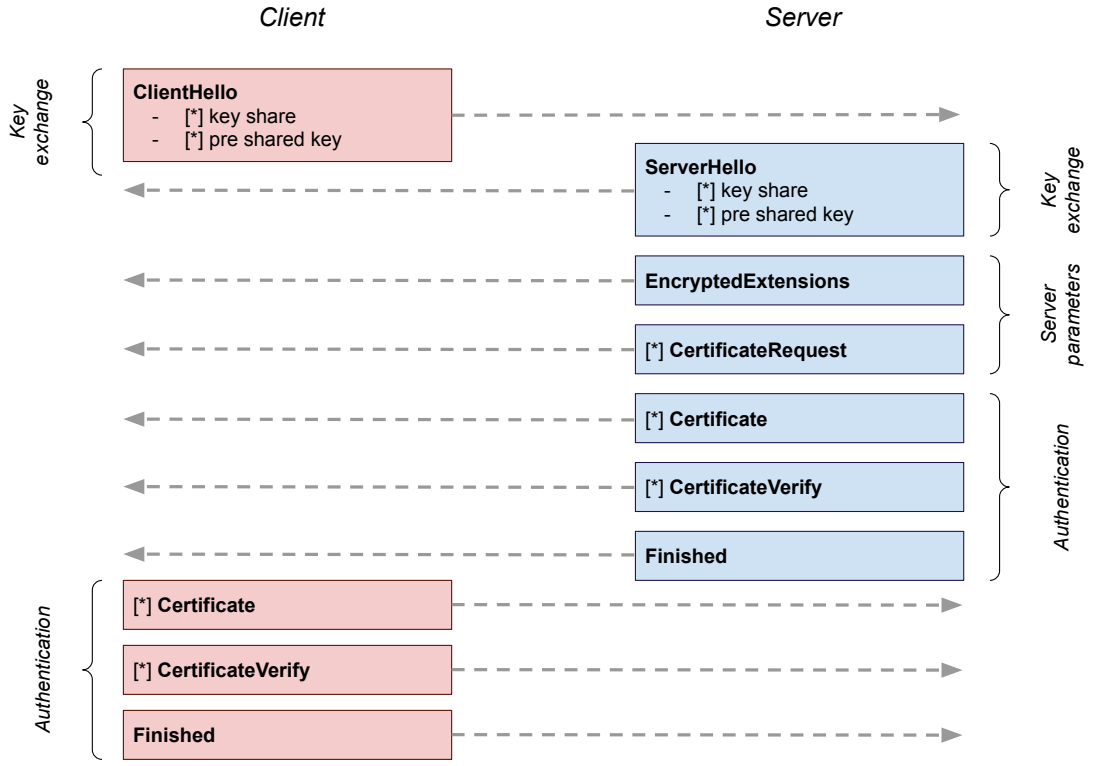


Figure 3: Full TLS handshake

The handshake can be divided in 3 phases:

1. **Key Exchange**: establishes the shared secret to derive the various shared keys and the cryptographic parameters. This comprises the `ClientHello` and `ServerHello` messages, and is the only phase that is unencrypted. All following messages are encrypted.

2. **Server Parameters**: determines additional parameters by the server (whether client authentication is required, or additional parameters for the application-layer protocol).
3. **Authentication**: performs the actual authentication of the server (and the client if required) and provides a final confirmation of the handshake integrity and shared secret.

A novelty of version 1.3 is that all messages of the handshake, except the `ClientHello` and `ServerHello` are encrypted. This provides additional security and privacy to the message exchanges. Note that even the certificates are encrypted preserving the privacy of the subjects.

Both parties can transmit application data after receiving and verifying the `Finished` message. As an exception, the server can start data transmission just after sending its `Finished` message without having to wait for the client `Finished`.

3.3 TLS 1.3 handshake messages

In the following sections we provide a brief description of the messages from the handshake.

3.3.1 ClientHello

This message initiates the handshake. Includes, among other things:

- Random value generated by the client R_C,
- List of supported cipher suites (symmetric cipher, hash functions, ...) in order of preference.
- Extensions: extended functionality required by the client.

Some of the extensions are mandatory. Two important ones are:

- Key share: this is the DH value sent by each party.
- Pre shared secret (PSK): the pre shared secret between client and server if it exists. If it is not available, PSK is set to a string of bytes set to 0.

3.3.2 ServerHello

Sent by the server in response to a `ClientHello`. It includes, among other things:

- Random value generated by the server R_S.
- The cipher suite selected by the server from those included in the `ClientHello` message.
- Extensions: list of extensions required to establish the cryptographic context.

3.3.3 ExcryptedExtensions

This message might contain extensions from the server not directly related to establishing the cryptographic parameters. This is the first message that is encrypted. We will not get into detail about such extensions, please refer to [10].

3.3.4 CertificateRequest

Message sent by the server in the case it wants to authenticate the client (mutual authentication).

3.3.5 Certificate

This message includes the certificate to be used for authentication. It also includes the chain of certificates to the root. We usually denote the server certificate as $cert_S$ and the client certificate as $cert_C$

Note that the client certificate is only sent when the server has sent `CertificateRequest` requiring client authentication.

3.3.6 CertificatedVerify

This message contains a signature over the hash of the previously exchanged messages in the handshake. When the message is sent by the server it includes $Sig_{SK_S}(h(msg))$, where SK_S is the private key of the server corresponding to the public key included in the certificate sent in the previous message. When the message is sent by the client it contains the corresponding $Sig_{SK_C}(h(msg))$, where SK_C is the private of the client. As the previous case, the client only sends this message if required by the server (after its `Certificate`)

These messages allow the authenticated party to proof that it does hold the private key corresponding to the public key provided in the corresponding certificate.

3.3.7 Finished

This is the final message sent by each party in the handshake. It is used to authenticate the whole handshake and the computed keys.

The content of the message is the HMAC of the hash of the previously exchanged messages from the handshake, using the *finished key*, Kf_S or Kf_c (see Section 3.4). E.g.: $HMAC(Kf_S, h(msg))$.

Note that we use $h(msg)$ to broadly denote the hash of the concatenation of all previously exchanged messages in the handshake. In the terminology of TLS this is known as the *transcript hash*.

3.4 Key derivation

Similarly to SSH (see Section 2.2.6), TLS also derives several keys from the shared secret established in the handshake. In order to do that it relies on a function called HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [5].

The whole key derivation procedure generates several keys including those to resume a handshake using PSK secrets. Among the generated secrets the most relevant are:

- Client handshake traffic secret: derives the encryption key and IV to be used by the client to encrypt all its handshake messages (excluding the `ClientHello`). It also derives the finished key used in the HMAC of the `Finished` message (see Section 3.3.7).
- Server handshake traffic secret: analogous to the previous but for the server's handshake messages.
- Client application traffic secret: derives the encryption key and IV used by the client to encrypt all its application data (after the handshake)
- Server application traffic secret: analogous to the previous but for the server.

handshake keys	$\rightarrow$ encryption key for the server (Kh_S)
	$\rightarrow$ IV for the server (IV_S)
	$\rightarrow$ finished key for the server (Kf_S)
	$\rightarrow$ encryption key for the client (Kh_C)
	$\rightarrow$ IV for the client (IV_C)
	$\rightarrow$ finished key for the client (Kh_C)
application keys	$\rightarrow$ encryption key for the server
	$\rightarrow$ IV for the server
	$\rightarrow$ encryption key for the client
	$\rightarrow$ IV for the client

Table 4: Main keys used in the TLS handshake

Note that TLS 1.3 does not explicitly use integrity keys (other than the finished key). In TLS 1.3 all supported symmetric encryption operation modes are Authenticated Encryption with Associated Data (AEAD) algorithms [2]. Details of its use are determined by the TLS Record Protocol.

As a summary the main keys generated in the handshake are summarized in Table 4. There are *handshake keys* used in the handshake messages, and *application keys* used when transmitting application data. Note that both server and client generate all the keys.

3.5 Example of TSL 1.3 handshake

We will show a simplified example of TLS 1.3 handshake using the notation from Table 5.

We will consider the case where the key exchange is solely performed with DHE (no PSK is used), which is usually the case of the first handshake. The client authenticates the servers, but the

R_S	Random value generated by user S
$cert_S$	Public key certificate of user S
PK_S, SK_S	public key pair of user S, where PK denotes the public key and SK the private key
CP	Crypto proposed (set of cryptographic parameters proposed)
CS	Crypto selected (set of cryptographic parameters selected)
p, g	Prime (p) and generator (g) for Diffie-Hellman key exchange
msg	Message including all previous messages exchanged in the protocols until the current one (the transcript hash).
$E_{PK_A}(m)$	Encryption of message m with public key of user A PK_A.
$E_K(m)$	Encryption of message m with symmetric key K.
Kh_A	Symmetric handshake key for user A.
Kf_A	Handshake finished key for user A.
$Sig_{SK}(m)$	Digital signature of message m with private key SK.
$h(m)$	(cryptographic) hash function h applied to message m.
S	the server
C	the client

Table 5: Notation used for the examples

$C \rightarrow S$	ClientHello	$R_C, CP, g^c \mod p$
$S \rightarrow C$	ServerHello	$R_S, CS, g^s \mod p$
$S \rightarrow C$	EncryptedExtensions	E_{Kh_S}(EncryptedExtensions)
$S \rightarrow C$	Certificate	$E_{Kh_S}(cert_S)$
$S \rightarrow C$	CertificateVerify	$E_{Kh_S}(Sig_{SK_S}(msg))$
$S \rightarrow C$	Finished	$E_{Kh_S}(HMAC(Kf_S, msg))$
$C \rightarrow S$	Finished	$E_{Kh_C}(HMAC(Kf_C, msg))$

Table 6: Simplified TLS 1.3 handshake with DHE key exchange

server does not authenticate the client (there is no mutual authentication). A simplified version of such handshake can be seen in Table 6.

4 Exercises

4.1 TLS 1.3 handshake with mutual authentication

Following the example from Section 3.5, fill the following table with the exchanged messages between client and server for a handshake using only DHE for key exchange, and with mutual authentication. That is, the client authenticates the server, and the server authenticates the client.

	Message	*Content*
$C \rightarrow S$	ClientHello	

4.2 Sniffing a TLS handshakes

Use a network sniffer to actually see the exchanged packets in a TLS handshake when accessing a secure web.

References

[1] M. Bellare, T. Kohno, and C. Namprempre. The secure shell (ssh) transport layer encryption modes. RFC 4344, RFC Editor, January 2006. URL: `http://www.rfc-editor.org/rfc/rfc4344.txt`.

[2] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Cryptology ePrint Archive, Report 2000/025, 2007. version 20070715:054351. URL: `https://eprint.iacr.org/2000/025.pdf`.

[3] F. Cusack and M. Forssen. Generic message exchange authentication for the secure shell protocol (ssh). RFC 4256, RFC Editor, January 2006.

[4] J. Galbraith and R. Thayer. The secure shell (ssh) public key file format. RFC 4716, RFC Editor, November 2006. URL: `http://www.rfc-editor.org/rfc/rfc4716.txt`.

[5] H. Krawczyk and P. Eronen. Hmac-based extract-and-expand key derivation function (hkdf). RFC 5869, RFC Editor, May 2010. URL: `http://www.rfc-editor.org/rfc/rfc5869.txt`.

[6] A. Langley, M. Hamburg, and S. Turner. Elliptic curves for security. RFC 7748, RFC Editor, January 2016. URL: `https://tools.ietf.org/rfc/rfc7748.txt`.

[7] S. Lehtinen and C. Lonvick. The secure shell (ssh) protocol assigned numbers. RFC 4250, RFC Editor, January 2006. URL: `http://www.rfc-editor.org/rfc/rfc4250.txt`.

[8] Y. Nir, S. Josefsson, and M. Pegourie-Gonnard. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls) versions 1.2 and earlier. RFC 8422, RFC Editor, August 2018. URL: `https://www.rfc-editor.org/rfc/rfc8422.txt`.

[9] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Berlin Heidelberg, 2010. URL: `http://link.springer.com/10.1007/978-3-642-04101-3`, `doi:10.1007/978-3-642-04101-3`.

[10] E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor, August 2018. URL: `https://tools.ietf.org/rfc/rfc8446.txt`.

[11] J. Schlyter and W. Griffin. Using dns to securely publish secure shell (ssh) key fingerprints. RFC 4255, RFC Editor, January 2006. URL: `http://www.rfc-editor.org/rfc/rfc4255.txt`.

[12] Nigel P. Smart. *Cyrptography made simple*. Springer, 2016. `doi:10.1007/978-3-319-21936-3`.

[13] Wikipedia contributors. Elliptic-curve diffie–hellman — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Elliptic-curve_Diffie%E2%80%93Hellman&oldid=940581391`, 2020. [Online; accessed 14-March-2020].

[14] T. Ylonen and C. Lonvick. The secure shell (ssh) authentication protocol. RFC 4252, RFC Editor, January 2006. URL: `http://www.rfc-editor.org/rfc/rfc4252.txt`.

[15] T. Ylonen and C. Lonvick. The secure shell (ssh) connection protocol. RFC 4254, RFC Editor, January 2006. URL: `http://www.rfc-editor.org/rfc/rfc4254.txt`.

[16] T. Ylonen and C. Lonvick. The secure shell (ssh) protocol architecture. RFC 4251, RFC Editor, January 2006. URL: `http://www.rfc-editor.org/rfc/rfc4251.txt`.

[17] T. Ylonen and C. Lonvick. The secure shell (ssh) transport layer protocol. RFC 4253, RFC Editor, January 2006. URL: `http://www.rfc-editor.org/rfc/rfc4253.txt`.