ADVANCED INFORMATION SYSTEM SECURITY

Hoping to get a better grade this time around.

Fabio Lorenzato

October 11, 2024

Contents

Chapter 1

Transport Layer Security

TLS, or Transport Layer Security, was originally proposed by Netscape in 1995 as a way to secure communications between a web browser and a web server. It is the successor to SSL, or Secure Sockets Layer, which was first introduced by Netscape in 1995. The two terms are often used interchangeably, but TLS is the more modern and secure protocol.
The main goal of SSL was to create secure network channel, almost at session level(4.5), between two parties, to provide some security services that neither TCP nor IP provides:

- **peer authentication** based on asymmetric challenge-response authentication(the challenge for the service is implicit, while for the client is explicit). Server authentication is alway compulsory, while client authentication is optional and requested by the server.
- **message confidentiality** base on symmetric encryption
- **message integrity** and authentication based on MAC computed on the trasmitted data
- **replay, filtering and reordering attack protection** using implicit record numbers(the correct order of transmission is provided by TCP, for this reason the number is implicit). This number is used also in the MAC computation.

You can see the TLS packet structure in figure 1.1. The TLS handshake protocol is used to establish a new session or reestablish an existing session. The TLS change cipher spec protocol is used to trigger the change of the algorithms to be used for message protection, or most notably to pass from the previous unprotected session to a protected one. The TLS alert protocol is used to signal errors or signal the end of the connection. The TLS record protocol contains the generic protocols informations and its content depend of the state of the connection and the protocol it is tunneling.

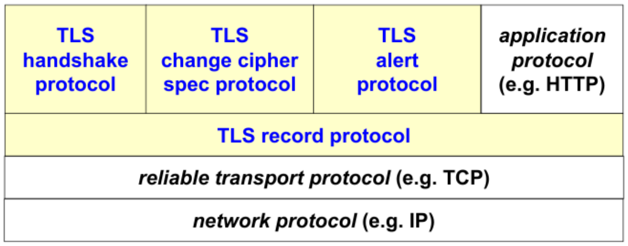

TLS handshake protocol
TLS change cipher spec protocol
TLS alert protocol
application protocol (e.g. HTTP)
TLS record protocol
reliable transport protocol (e.g. TCP)
network protocol (e.g. IP)

Figure 1.1: TLS packet structure.

1.1 TLS session and connection

It is important to make a clear distinction between TLS session and connections.
TLS sessions a **logical association** between client and server, created via an handshake protocol and its shared between different TLS connections(1:N).
TLS connections are a **transient TLS channel** between client and server, which means that each connection is associated with only one specific TLS session(1:1).

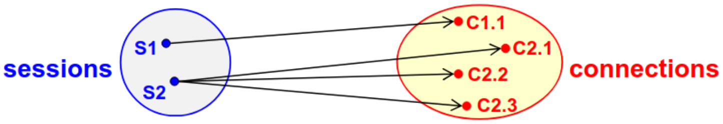


Figure 1.2: TLS session and connection.

1.2 TLS handshake protocol

The TLS handshake protocol is used to establish a new session or reestablish an existing session. It's a critical part of the TLS protocol, because the channel pass from an unprotected state to a protected one. During this phase the two parts agree agree on a set of algorithms for confidentiality and integrity, exchange random numbers between the client and the server to be used for the subsequent generation of the keys, establish a symmetric key by means of public key operations (originally RSA and DHKE, but nowadays the elliptic curve versions of algorithms are used) and negotiate the session-id and exchange the necessary public keys certificates for the asymmetric challenge-response authentication.

1.3 Achieving Data protection

Data protection is achieved by using symmetric encryption algorithms to encrypt the data and Message Authentication Codes(MAC) to ensure the integrity of the data and the authentication of the sender.
Figure 1.3 shows how the data protection is achived in TLS using authenticate-then-encrypt approachm, but also encrypt-then-authenticate is possible.
The MAC is computed over the data(compressed or not), the TLS sequence number and the key used for the MAC computation. The padding is also part of the MAC computation to avoid those attacks that change the padding.

The MAC is then encrypted with the dedicated symmetric key and a suitable initialization vector(IV)

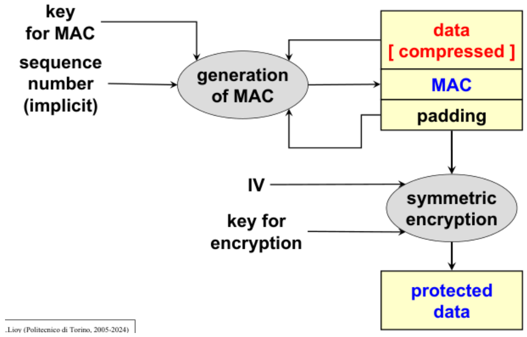


Figure 1.3: TLS data protection.

The keys are **directional**, so there are two keys(one for client to server and one for server to client) to protect against reuse of the sequence number in the opposite direction.

1.4 Relationship among keys and sessions

When a new session is created using the handshake protocol, a new **pre-master secret** is established using public key cryptography. Then from the session a new connection is created, which requires a random number to be generated, and exchanged between the client and the server. Those two values are combined via a KDF, usually HHKDF(HMAC-based key derivation function) to generate the **master secret**. This computation is done only once, and the secret is common to several connections.
The pre-master secret is then discarded, and the keys necessary for the MAC computation, encryption and, if necessary, IVs will be derived from the master secret.
You can notice that the master secret is common to any connections inside a session, but the per-connection keys are different every time. This is another important feature to avoid replay attacks, possible because numbering is per-connection. This solution also allows to reduce the cost of establishing new keys for each connection.

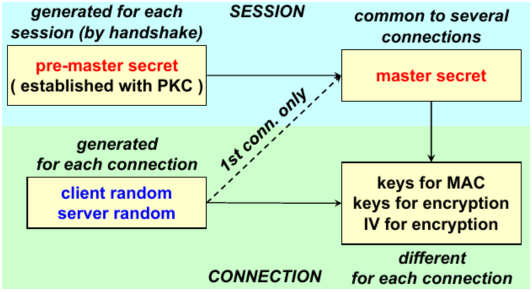


Figure 1.4: Relationship among keys and sessions.

1.5 Perfect Forward Secrecy

Since the keys are generated from symmetric crypto, if the private key used to perform encryption and decryption of the pre-master secret is compromised, all the previous communication can be decrypted because it is possible to derive the master-secret. This is only possible if the server has a certificate valid for both signature and encryption In this context, perfect forward secrecy is desirable.

> **Perfect Forward Secrecy** is a property of key-agreement protocols ensuring that the compromise of the secret key used for will compromise only current (and eventually future) traffic but not the past one

The most common way to achieve this is to use **ephemeral keys**, which are one-time asymmetric keys(used for key exchange). This means that the key pair used for key exchange is not a long term key pair, but a temporary one generated on-the-fly when necessary.
The ephemeral key needs to be authenticated, so only for this purpose the long-term key is used for signing the ephemeral key. This is done using DHKE instead of RSA because the latter one is really slow, while the former one is faster with the compromise of only using the established key for a certain number of session.
Let's now go over some considerations: if the temporary key is compromised, perfect forward secrecy of the communication is still valid because he can only decrypt the traffic exchanged using the temporary key. On the contrary, if the long-term key is compromised, no secret is really disclosed, because no traffic has been exchanged using it for encryption, but is still a problem for server authentication.

1.6 The protocol

The TLS handshake is always initiated by the client.

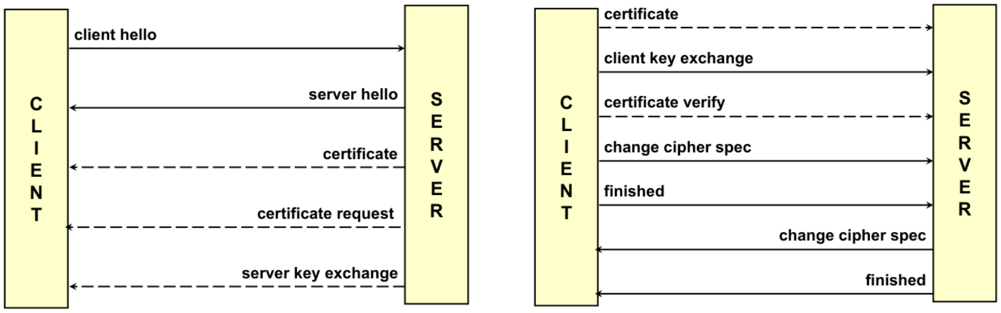
1.6.1 Client Hello and Server Hello

In version 1.2 the client sends a **Client Hello**, which contains:

- the SSL version preferred by the client, and the highest supported(2=SSL-2, 3.0=SSL-3, 3.1=TLS-1.0, . . .)
- a 28 bytes pseudo-random number, which is the client random
- a session-id, which is *0* if the client is starting a new session, and *1* if the client is trying to resume a previous session
- a list of cipher suites supported by the client, in order to let the server choose the most secure one (the set of algorithms used for encryption, for key exchange, and for integrity)
- a list of compression methods supported by the client (supported only up to TLS 1.2)

And then a **server hello** is sent back, which contains:

- the SSL version chosen by the server, the highest one supported by both the client and the server
- a 28 bytes pseudo-random number, which is the server random
- a session identifier(session-id), which is a new one if the server is starting a new session, and the same as the client's if the server is resuming a previous session
- the cipher suite chosen by the server, the strongest common one between the client and the server
- the compression method chosen by the server



(a) The TLS handshake protocol(TLS 1.2). (b) The TLS handshake protocol(TLS 1.3).

1.6.2 Cipher suite

A cipher suite is a string which contains the set of cryptographic algorithms used in the TLS protocol. A typical cipher suite consists of a key exchange algorithm, the symmetric encryption algorithm, and the hash function used for generating MACs. Some example of those are:

- SSL_NULL_WITH_NULL_NULL (no protection, used for the record protocol to be used in the handshake)
- SSL_RSA_WITH_NULL_SHA
- SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
- SSL_RSA_WITH_3DES_EDE_CBC_SHA

1.6.3 Certificates

After the initial exchange, the server is ready to authenticate itself.
The server sends its long-term public key certificate to the client for server authentication. Actually, the whole certificate chain, up to the root CA, must be sent. Furthermore , the subject of the certificate must match the server name.
Server authentication is implicit, because its private key is used to decrypt the pre-master secret, while client authentication is always explicit.

> The implicit server authentication is based on the fact that the MAC is computed trough the key derived trough knowledge of the private key, so only the server can compute it.

Optionally, the server can request a certificate from the client for client authentication. In this case the server specifies the list of trusted CA's, and the client sends its certificate chain. The browsers show to the users (for a connection) only the certificates issued by trusted CAs. If client certificate verification is required, an explicit request to send the hash computed over all the handshake messages before this one and encrypted with the client private key is sent to the client.

1.6.4 Key exchange

The key exchange is the most important part of the handshake protocol. If the server is using RSA for key exchange, the client generates a pre-master secret, encrypts it with the server's public key(which can be ephemeral or from its x.509 certificate) and sends it to the server. If RSA is no used, DHKE can be used to generate the pre-master secret, and in this case the server computes the value independently and the two parts can derive the master secret.
Another option is to use FORTEZZA, which is a key exchange algorithm based on DH.

1.6.5 Certificate verify

In case the server requested client authentication, the client will be required to send the certificate to prove that he is the owner of it's private key. The message to be signed is the hash of all the messages exchanged up to this point in the handshake protocol. This is to avoid replay attacks too.

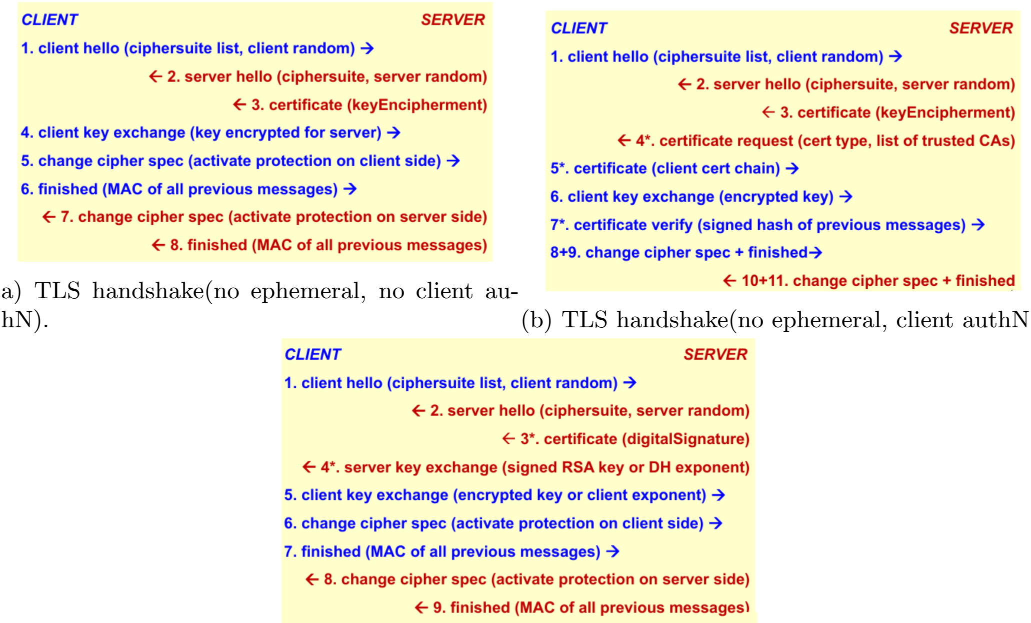
1.6.6 Change cipher spec

The change cipher spec message used to trigger the change of the algorithms to be used for message protection. It allows to pass from the previous unprotected messages to the protection of the next messages with algorithms and keys just negotiated, thus is technically a protocol on its own and not part of the handshake. Some analysis even say that it could be removed from it.

1.6.7 Finished message

The finished message is the last message of the handshake protocol, and the first message protected by the negotiated keys and algorithms. It is necessary to ensure that the handshake has not been tampered with, and it contains contains a MAC computed over all the previous handshake messages (but change cipher spec) using as a key the master secret. Notice that the finished message is different for the client and the server, because the MAC is computed over different messages.

This allows to prevent rollback man-in-the-middle attacks (version downgrade or ciphersuite downgrade)



CLIENT SERVER

1. client hello (ciphersuite list, client random) →

← 2. server hello (ciphersuite, server random)

← 3. certificate (keyEncipherment)

4. client key exchange (key encrypted for server) →

5. change cipher spec (activate protection on client side) →

6. finished (MAC of all previous messages) →

← 7. change cipher spec (activate protection on server side)

← 8. finished (MAC of all previous messages)

(a) TLS handshake(no ephemeral, no client authN).

CLIENT SERVER

1. client hello (ciphersuite list, client random) →

← 2. server hello (ciphersuite, server random)

← 3. certificate (keyEncipherment)

← 4*. certificate request (cert type, list of trusted CAs)

5*. certificate (client cert chain) →

6. client key exchange (encrypted key) →

7*. certificate verify (signed hash of previous messages) →

8+9. change cipher spec + finished→

← 10+11. change cipher spec + finished

(b) TLS handshake(no ephemeral, client authN).

CLIENT SERVER

1. client hello (ciphersuite list, client random) →

← 2. server hello (ciphersuite, server random)

← 3*. certificate (digitalSignature)

← 4*. server key exchange (signed RSA key or DH exponent)

5. client key exchange (encrypted key or client exponent) →

6. change cipher spec (activate protection on client side) →

7. finished (MAC of all previous messages) →

← 8. change cipher spec (activate protection on server side)

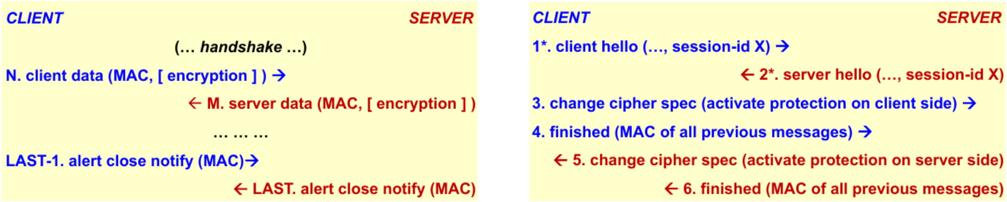
← 9. finished (MAC of all previous messages)

(c) TLS handshake(ephemeral, no client authN).

Figure 1.6: TLS handshake protocol.

1.7 Setup Time

The setup time is the time required to establish a secure connection between the client and the server. TLS depends on TCP, so the TCP handshake must be taken into account. Then the TLS handshake is performed, meaning that typically 3 RTTs (1 for TCP and 2 for TLS) are required to establish a secure connection. Usually after 180ms the two parties are ready to send protected data(assuming 30ms delay one-way).



CLIENT SERVER

(… *handshake* …)

N. client data (MAC, [encryption]) →

← M. server data (MAC, [encryption])

… … …

LAST-1. alert close notify (MAC)→

← LAST. alert close notify (MAC)

(a) TLS link teardown.

CLIENT SERVER

1*. client hello (…, session-id X) →

← 2*. server hello (…, session-id X)

3. change cipher spec (activate protection on client side) →

4. finished (MAC of all previous messages) →

← 5. change cipher spec (activate protection on server side)

← 6. finished (MAC of all previous messages)

(b) TLS resume session.

Figure 1.7: TLS link teardown and resume session.

1.8 TLS versions

1.8.1 TLS 1.0

TLS 1.0, or SSL 3.1, was released in 1999. It is the first version of the protocol, and it is based on SSL 3.0. Previous version were using proprietary solutions, so the adoption of open standards was strongly encouraged.

1.8.2 TLS 1.1

TLS 1.1 was released in 2006, and it introduced some security fixes especially to protect against CBC attacks. In fact, the implicit IV is replaced with an explicit IV to protect against CBC attacks. Also protection against padding oracle attacks were introduces to reduce the information leaks. For this reason Passing errors now use the bad_record_mac alert message (rather than the decryption_failed one). Furthermore, premature closes no longer cause a session to be non- resumable.

1.8.3 TLS 1.2

TLS 1.2 was released in 2008, and it introduced some new features and improvements. The chipersuite also specifies the pseudo random function instead of leaving the choice to the implementation. The sha-1 algorithm was replaced with SHA-256, and its also added support for authenticated encryption, such as AES in GCN or CCM mode.

All the chipersuites tat use IDEA and DES are deprecated.

1.9 TLS attacks

1.9.1 Heartbleed

Heartbleed is a security bug in the OpenSSL cryptography library, which is a widely used implementation of the TLS protocol. It was able to exploit the fact that the heartbeat extension keeps the connection alive without the need to negotiate the SSL session again. The attacker could send a heartbeat request, but the length of the response is much longer(up to 64KB) than the actual data sent by the client. This attack could then allow to leak memory contents.

1.9.2 Bleichenbacher attack

Blackenbacker is a 1998 attack and it's the so-called the million message attack because it exploited a vulnerability in the way the RSA encryption was done. RSA requires the padding to be done in a certain way, because if it is unoptimally done, it could cause some issues.

The attacker could perform an RSA private key operation with a server's private key by sending a million or so well-crafted messages and looking for differences in the error codes returned. By basically knowing the public key and trying to decrypt a message with some guessed private keys, the different responses obtained were giving hints about which bits were correct and which bits were wrong.

Later one the RSA implementations moved to RSA-OAEP, which is a padding scheme that is provably secure against chosen-ciphertext attacks.

In 2017 another variant of this attack was discovered, called ROBOT(Return Of Bleichenbacher's Oracle Threat), to which many major websites, like Facebook, were vulnerable.

1.9.3 Other attacks against SSL/TLS

Some other attacks against SSL/TLS are CRIME, BREACH, BEAST and POODLE.

Crime is an attack against the **compression algorithm** used in **SSL/TLS**, which by injection chosen plaintext in the user requests, for example by using a form or choosing fraudulently an username that is displayed, and then measure the size of the encrypted traffic, an attacker could recover specific plaintext parts exploiting information leaked from the compression, and this is part of the reason why the compression is deprecated in TLS 1.3.

BREACH is an attack against the **HTTP compression** to deduce a secret within the HTTP response provided by the server. It is different from Crime because the former is an attack against the compression algorithm used in SSL/TLS, while the latter is an attack against the HTTP compression.

BEAST is an attack that exploits a vulnerability in the way the **CBC** mode of operation is used in SSL/TLS. The attack is possible if **IV concatenation**is used, meaning that the initial vector for the next encryption is taken from the end of the previous encryption. A MITM may decrypt HTTP headers with a blockwise-adaptive chosen-plaintext attack, and by doing so, he's able to decrypt HTTPS requests and steal information such as session cookies.

POODLE, or Padding Oracle On Downgraded Legacy Encryption, is an attack that exploits the fact that SSL 3.0 uses a padding scheme that is vulnerable to a **padding oracle attack**, by acting as a MITM. This is done by exploiting SSL-3 fallbacks to decrypt data. This is also the only attack among those that still works today.

FREAK, or Factoring RSA Export Keys, is an attack that exploits the downgrades on TLS to export-level RSA keys to a factorizable bit length(512 bits). It is also possible to carry this out by downgrading the symmetric key too and then perform a brute force attack(40-bit). As you can see from figure 1.8, in the first phase the random and the supported elliptic curve are not altered by the MITM, but only the supported cipher suites are altered(to export level ones). Usually 40 bits chiper suits should not be configured at all, but some misconfigurations may happen. The third phase is where the magic happen: we have theoretically the MAC in the FINISHED message to protect against tampering (recall that the MAC will be computed over all the handshake messages) but since the MAC is protected with the master secret, the master secret is only 40 bits. The attacker can then brute force the master secret on-the-fly, recompute the MAC so that the server will accept the message. Since the attacker have access to the premaster secret and the master secret, all traffic beyond this point is encrypted with a weak shared key and the middleman can read and even modify the traffic.

For all those reasons SSL-3 has been disabled on most browsers, but its still needed for some browsers, for example IE6 by Microsoft, which is outlasting its expected life span, because its the default browser on Windows XP, which is still used today unfortunately, meaning that the window of exposure for those attacks is still open.

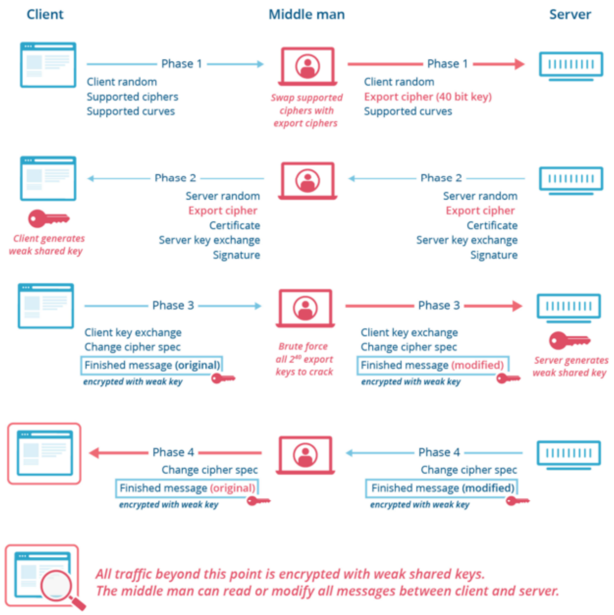


Figure 1.8: FREAK attack.

1.10 ALPN extension

The ALPN extension, or Application-Layer Protocol Negotiation, is an extension that allow to negotiate the application protocol to speed up the connection creation, avoiding additional round-trips for application negotiation. It is used to negotiate the protocol to be used on top of the TLS connection, such as HTTP/2, SPDY, or QUIC, before the connection is established. This is useful because it saves time, obviously, because after the connection is established, the client and server could still fail to communicate because the application protocol is not supported by the server.

The extension is inserted in the client hello message, by setting the ALPN flag to true and providing a list of supported protocols in the client HELLO message. The server will respond with the ALPN flag set to true(if it supports the extension) and the selected protocol.

This is also useful for those servers that use different certificates for the different application protocols.

1.11 TLS False Start

TLS False Start is another extension that allows the client can send application data together with the ChangeCipherSpec and Finished messages, in a single segment, without waiting for the corresponding server messages. The biggest advantage of using this is the reduction of the latency to 1 RTT. In theory this should work without changes, but to use this in Chrome and Firefox they require the ALPN and the Forward Secrecy enabled, while Safari requires forward secrecy.

1.12 The TLS downgrade problem

In theory, when negotiating the TLS version to be used, the client sends (in ClientHello) the highest supported version, while the server notifies (in ServerHello) the version to be used (highest in common with client).

For example if the client support up to SSL 3.3(TLS 1.2) and the server support the same version, the connection will be established using TLS 1.2. But if the server supports only SSL 3.2(TLS 1.1), the connection will be established using SSL TLS 1.1.

Some servers, instead of sending the highest version supported, just close the connection, forcing the client to retry with a lower version of the protocol. An attacker could exploit this behavior to force the client to use an older version of the protocol, by repeatedly closing the connection, and then exploit the vulnerabilities of the older version of the protocol.

> This means that its not a problem of the protocol itself, but of the implementation of the server.

1.12.1 TLS Fallback Signalling Cipher Suite Value (SCSV)

This behavior is not always an attack, for example there could be and error in the channel, which is closed by the server. This means that there's a need to distinguish between a real attack and a simple error.

The **TLS Fallback SCSV** is a **special value** that is used to prevent the protocol downgrade attacks, not only chiper suite. It do so by sending a new (dummy) ciphersuite value(TLS_FALLBACK_SCSV) which is sent by the client when opening a downgraded connection as the last value in the chipersuite list.

If the server receives this value and still supports an higher version of the protocol, it will know that the client is trying to downgrade the connection, and it will refuse to establish the connection by sending a **inappropriate_fallback** alert message an closing the channel. This notify the client that he should retry with the highest version of the protocol supported by himself.

Many servers do no support SCSV yet, but most servers have fixed their behavior when the client requests a version higher than the supported one so browsers can now disable insecure downgrade

1.13 TLS session tickets

We know that session resumption is possible with TLS, but the server needs to keep a cache of session IDs, which may become very large for high traffic servers. For this reason, the **TLS session tickets** were introduced, which are an extension allowing the server to send the session data to the client encrypted with a server secret key. This data is stored by the client, which will send it again when it wants to resume a session. This allows to move the cache to the client side. Obviously, this data has to be encrypted with a server secret key.

Even if this behaviour is desirable for the server, it still need to be supported by the browser (it's an extension after all) and needs a mechanism to share keys among the servers in a load-balancing heavy environment.

1.14 The Virtual Server Problem

Nowadays, virtual servers are very common in web hosting, because they allows to have different logical names associated with the same IP address(ie: home.myweb.it=10.1.2.3, food.myweb.it=10.1.2.3). This is easy to manage in HTTP/1.1 but quite troublesome with HTTPS, because TLS is activated before the HTTP request is sent, which makes it difficult to know which certificate should be provided in advance. The solutions are quire simple:

- use a wildcard certificate, which is a certificate that is valid for all the subdomains of a domain (ie: *.myweb.it)
- use the SNI (Server Name Indication) extension, which is an extension that allows the client to specify the hostname of the server it is trying to connect to, allowing the server to provide the correct certificate. This is sent in the ClientHello message.
- provide a certificate with a list of servers in subjectAltName, which allow to share the same private key for different servers.

1.15 TLS 1.3

TLS 1.3 was released in 2018, and it introduced some new features while solving some of the most common problems of the previous versions:

- **reduce** the **handshake latency** in general
- **encrypting** more of the **handshake** (for security and privacy, after all up to TLS 1.2 the handshake was in clear text)
- **improving resiliency** to cross-protocol attacks
- removing legacy features

We will now go over the main changes in TLS 1.3.

1.15.1 Key exchange

In this version, the support for static RSA and DH key exchange was removed for many reasons:

- it does not implement forward secrecy
- its difficult to implement correctly, which is a problem because it exposes the system to many attacks like Bleichenbacher

Now Diffie-Hellman ephemeral (DHE) and Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) are the only key exchange methods supported, with some required parameters(some implementations weren't really up to the standards, ie: used DH with small numbers(just 512 bits) or generated values without the required mathematical properties).

1.15.2 Message protection

TLS 1.3 greatly improves message protection by eliminating several vulnerabilities present in earlier versions. Previously, issues arose from using CBC mode with authenticate-then-encrypt, which led to attacks like Lucky13 and POODLE. The use of RC4 also allowed plaintext recovery due to measurable biases, and compression enabled the CRIME attack.
TLS 1.3 addresses these weaknesses by removing CBC mode and enforcing AEAD modes for stronger security. Insecure algorithms such as RC4, 3DES, Camellia, MD5, and SHA-1 have been dropped, and compression is no longer used, ensuring a more secure cryptographic environment.

1.15.3 Digital signature

In TLS 1.3, digital signatures have been strengthened to address earlier vulnerabilities. Previously, RSA signatures were used on ephemeral keys with the outdated PKCS#1v1.5 schema, leading to potential flaws. The handshake was authenticated using a MAC instead of a proper digital signature, which exposed the protocol to attacks like FREAK.
TLS 1.3 improves this by using the modern, secure RSA-PSS signature scheme. Additionally, the entire handshake is signed, not just the ephemeral keys, providing more comprehensive security. The protocol also adopts modern signature schemes, enhancing the overall strength of the cryptographic process.

1.15.4 Ciphersuites

TLS 1.3 simplifies the protocol by reducing the complexity seen in earlier versions, which had a long list of cryptographic options that grew exponentially with each new algorithm. This complexity made configuration and security management difficult.
To address this, TLS 1.3 specifies only essential, orthogonal elements: a cipher (and mode) combined with an HKDF hash function. It no longer ties the protocol to specific certificate types (like RSA, ECDSA, or EdDSA) or key exchange methods (such as DHE, ECDHE, or PSK).
Moreover, TLS 1.3 narrows the selection to just five ciphersuites:

- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256
- TLS_AES_128_CCM_SHA256
- TLS_AES_128_CCM_8_SHA256 (deprecated but yet supported for computational environments with low capacity)

1.15.5 EdDSA

EdDSA, or Edwards-curve Digital Signature Algorithm, is a digital signature scheme using the EdDSA signature scheme, which is a variant of the standard elliptic curve DSA schema. The EdDSA scheme, unlike standard DSA, doesnt require a PRNG, which could in some cases leak the private key if the underlying generation algorithm is broken or predictable. EdDSA picks a nonce based on a **hash of the private key** and the **message**, which means after the private key is generated there's no need anymore for random number generators. Another advantage is that the EdDSA is faster in signature generation and verification than the standard DSA, because it implements simplified point addition and doubling.

EdDSA is using an n-bit private and public keys and will generate a signature which has a size which is the double of the keys size. As per the RFC stipulations, EdDSA must support the Ed25519 and Ed448 curves, which are based on the Edwards-curve Digital Signature, which uses respectively SHA-2 and SHA-3 as hash functions. Among those two, Ed25519 is the most used, which is also used for an implementation of EcDH.

When it comes to the RFCs there's an issue: there are two RFCs that are slightly different, one is the RFC 8032, which is for general internet implementations, with the implementational details left to the developers, and there's also FIPS 186-5, which specifies not only the mathematics behind an algorithm but also implementation details because it is a standard for the US government.

1.15.6 Other improvements

TLS 1.3 introduces several key improvements to enhance security and efficiency. One major change is that all handshake messages sent after the ServerHello are now encrypted, ensuring better confidentiality during the handshake process, even if the keys have not been enstablished(we will go over the details later). The new **EncryptedExtensions** message allows additional extensions, which were previously sent in cleartext during ServerHello, to also benefit from encryption, improving privacy.

Additionally, key derivation functions have been redesigned to support easier cryptographic analysis due to their clear key separation properties. HKDF is now used as the underlying primitive for key derivation, providing a robust and flexible method.

The handshake state machine has also been overhauled, making it more consistent by removing unnecessary messages, such as **ChangeCipherSpec**, which is now only retained for compatibility with older systems. This restructuring streamlines the handshake process, reducing complexity and improving protocol efficiency.

1.15.7 HKDF in TLS 1.3

HKDF is one of the novelties of TLS 1.3, and is a HMAC-based extract-and-expand Key Derivation Function. As a normal key derivation function, it takes 4 inputs: the salt, the input keying material(which is basically a secret key), the info string and the output length. It basically takes the first 2 inputs to derive a key which will be used to derive a pseudorandom key .Then the second stage expands this key into several additional pseudorandom keys, which are the output of the KDF. So multiple outputs can be generated from the single IKM value by using different values info strings.

By calling the function repeatedly call the HMAC function with with PR keys and the info string as the input message, and then concatenate the results, by appending the output of the HMAC function to the previous output and incrementing an 8-bit counter.

This means that from one call we can generate at most $256(2^8)$ different keys.

The finished message is not protected with the master secret as it was in TLS 1.2, but with the specific key, which is created with expand starting from the base key and putting as info the text ”finished” and then the desired length. The pre-shared key to protect the ticket contains the word ”resumption”. So you start with some basic key material and then by using different labels, you are able to generate the different keys.

```
HKDF-Expand( HKDF-Extract ( salt, IKM ), info, length )
```

Listing 1: HKDF-Expand pseudocode.

1.15.8 TLS-1.3 handshake

Even the handshake in TLS 1.3 was changed. The first thing that we notice is that the change chiper specs has been removed from the standard.
At first, as you can see from figure 1.9, the connection starts with an hello message from the client, which will also send it's list of chipersuites and the key share material , in which the client will send it's DH parameters(it is still needed after all). All this data is sent in the same message to save some RTTs and to allow the middleboxes with lower TLS version support to understand the message, interpreting them as an extension of the client hello and thus skipping them. In fact most of the TLS 1.3 features are implemented in message extensions.
The following messages are just the specular ones but sent from the server this time. After this point it is possible to have confidentiality of the communication, because a shared key has been established(the curly braces in the picture are just for that).

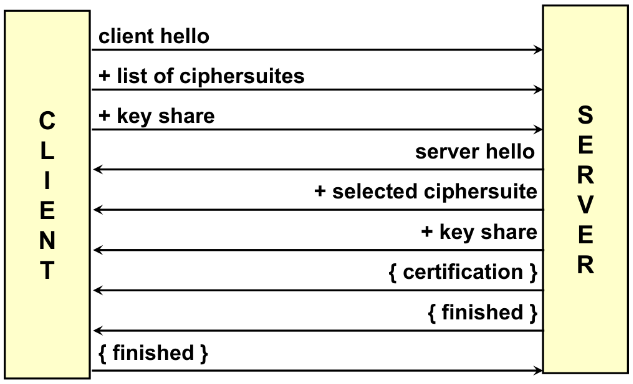


Figure 1.9: TLS 1.3 handshake.

Client request

The client hello message contains the **client random**, the highest supported protocol version(which is 1.2), client random, the supported cipher suites and compression methods(no compression allows but still necessary for backward compatibility) and of course the session id.
The supported extensions are:

- key_share = client (EC)DHE share

- signature_algorithms = list of supported algorithms
- psk_key_exchange_modes = list of supported modes for the pre-shared key exchange
- pre_shared_key = list of PSKs offered, which are not the real keys, are just an index label for the pre-shared keys available

Recall that if the extension is not supported by the server or middlebox it will be ignored.

Server response

The server will respond with it's parameters, some of which will already be encrypted because the key has been established. The key exchange material will be sent in clear of course(the key_share and the pre_shared_key).
The servers parameters are present too and encrypted, which are

- { EncryptedExtensions } = responses to non-crypto client extensions
- { CertificateRequest } = request for client certificate, optional and encrypted for privacy

Even the server authentication parameters are sent in the same message:

- { Certificate } = X.509 certificate (or raw key, RFC-7250, useful for low power IoT devices)
- { CertificateVerify } = signature over the entire handshake
- { Finished } = MAC over the entire handshake

Finally, the server can immediately send application data, if needed, which will be encrypted with the key derived for data protection, not the one derived for the handshake.

Client finish

The client will respond with the last messages, which are:

- { Certificate } = X.509 certificate (or raw key, RFC-7250)
- { CertificateVerify } = signature over the entire handshake
- { Finished } = MAC over the entire handshake

Pre-shared keys

In TLS, the pre-shared key (PSK) replaces the session ID and session ticket. PSKs are agreed upon during a full handshake and can be reused for multiple connections.

> Pre-shared keys are only used if a previous session has been established.

They can be combined with (EC)DHE to achieve forward secrecy, where the PSK is used for authentication and (EC)DHE for key agreement. While PSKs can be generated out-of-band (OOB) from a passphrase, this is risky due to the potential lack of randomness, making brute-force attacks feasible. Therefore, using OOB PSKs is generally discouraged.

0-RTT connections

In TLS 1.3, when using a pre-shared key (PSK), a client can send ”early data” with its initial message, which is protected by a specific key. However, this approach lacks forward secrecy because it relies solely on the PSK and could be vulnerable to replay attacks. While some complex mitigations exist, they are particularly challenging for multi-instance servers.

Incorrect share

In TLS 1.3, if a client sends a list of (EC)DHE groups that the server does not support, the server responds with a HelloRetryRequest, prompting the client to restart the handshake with different groups. If the new groups are also unacceptable, the handshake will be aborted, and the server will send an appropriate alert.

1.16 TLS and PKI

Even with TLS 1.3, public key certificates are required, meaning TLS has a connection with Public Key Infrastructure (PKI), as certificates are typically issued by a PKI. PKI is always needed for server authentication and optionally for client authentication unless using ”true” PSK (Pre-Shared Key) authentication. In this case, ”true” refers to manually pre-installed shared keys between the client and server, which is rare, often limited to certain embedded systems.

When a peer (server or client) sends its certificate to the other peer:

- The entire certificate chain is sent, which includes the peer’s certificate and certificates of intermediate CAs (Certification Authorities). However, the root CA certificate is not sent, as it should already be trusted and present on the other side. Some attackers may attempt to send a fake root CA certificate; if the receiving peer accepts it without proper verification, the attack is successful. For instance, in TLS monitoring systems, it’s crucial to ensure that the certificate chain never includes the root CA.

- The receiving peer must validate each certificate in the chain. A potential vulnerability is when some browsers and servers only validate the peer’s end certificate, neglecting the intermediate CAs, which could be invalid. Proper validation must recursively verify each certificate in the chain, including checking the correctness of signatures, issue, and expiration dates.

- It’s also essential to verify whether a seemingly valid certificate has been revoked. This can be done using either:

  - **CRL (Certificate Revocation List):** A list of revoked certificates, though the list can be large, making downloading and storing CRLs cumbersome—especially since a separate CRL is needed for each step in the chain.
  - **OCSP (Online Certificate Status Protocol):** More efficient performance-wise, as it allows querying an OCSP server to check if a certificate is valid. However, this method raises privacy concerns, as the OCSP server gains visibility into the servers being visited.

Both CRL and OCSP add additional network requests and delays to the setup, as the certificates must be validated before the session begins. For example, OCSP can introduce a median delay of +300ms, with an average of +1 second. Therefore, privacy and performance trade-offs must be considered.

1.17 TLS and certificate status

What should be done if the CRL or OCSP servers are unreachable? This could happen due to server or network failures, or even because a firewall is blocking access—common in companies with strict firewall rules due to security policies. Another reason could be that OCSP responses are sent in clear text, even though they are signed. The potential responses to this situation are:

- **Hard fail:** The page is not displayed, and the user is shown a security warning. For instance, "Sorry, we cannot display this page because we couldn't verify its revocation status." While this is the most secure option, it is also the most inconvenient for users. It could also result from a (D)DoS attack on the OCSP or CRL server.

- **Soft fail:** The page is shown regardless of whether the certificate's revocation status can be verified, assuming the certificate is still valid.

Both approaches lead to extra load time, often longer than just waiting for an OCSP response. This happens because the system needs to wait for the timeout after initiating a TCP connection. To address these issues, two different solutions are typically used.

1.17.1 Pushed CRL

In most cases, revoked certificates result from a compromised intermediate CA. When this happens, all certificates issued by that CA must be revoked, which can be a significant number. As a result, browser vendors have chosen to push updates containing only major revoked certificates. If an intermediate CA becomes invalid, it is promptly marked as invalid, and the CRL containing this information is distributed through a browser update. Here are some examples of how different browsers handle this:

- **Internet Explorer:** Updates its revoked certificate list with each browser update, which isn't ideal. If an update is delayed, the user may not be informed about the CA revocation in a timely manner.

- **Firefox:** Uses a system called *oneCRL*, which is part of its blocklist management process. This list, maintained by Firefox's developers, is updated nightly and includes CRLs of revoked intermediate CAs as well as known malicious websites.

- **Chrome:** Utilizes *CRLsets* to manage these CRLs, which are pushed with each browser update. On startup, Chrome checks for new CRLsets.

1.18 OSCP Stapling

Let's now look at how browsers and servers manage OCSP-related issues, which can affect both verification and privacy. The term "Stapling" refers to keeping things together.

- Automatic downloads of CRL and OCSP are often disabled because they take too long. As a result, browsers may be vulnerable to attacks using revoked certificates.
- Pushed CRLs only include a subset of revoked certificates; otherwise, the list would become too large.
- Browser behavior varies significantly in this area. Depending on the browser, different features may be available, and there is no standardized method for handling certificate revocation.

To improve this, instead of relying on the client to fetch revocation information, the server takes responsibility through a process called *OCSP stapling.* When the server sends its certificate, it also provides recent revocation information via an OCSP response (essentially saying, ”Here's my certificate, and here's proof it's valid”). This allows the client to skip making a separate OCSP request, as the necessary information is included in the initial TLS handshake.

OCSP Stapling is a TLS extension, meaning it is not part of the standard handshake and must be supported by both the server and the client. It is announced during the TLS handshake when the server sends the *Server Hello* message, indicating to the client that it will be using this extension.
The first version of OCSP Stapling is defined in RFC-6066 (extension **status_request**), while version two is in RFC-6961 (extension **status_request_v2**) with the value **CertificateStatusRequest**. The TLS server pre-fetches the OCSP response from an OCSP server and includes it in the handshake as part of the server's certificate message. This message contains not only the certificate chain but also the OCSP responses for each certificate in the chain. As a result, the message size increases since there must be an OCSP response for each certificate, which validates that the certificates have not been revoked. This process “staples” the OCSP responses to the certificates.

The main advantage is improved privacy for the client, as it no longer needs to request information directly from the OCSP server—this is handled by the TLS server, meaning the OCSP server remains unaware of which clients are requesting access. Additionally, the client avoids having to establish a separate connection to the OCSP server, improving both speed and privacy.

However, a downside is that the freshness of the OCSP responses can be an issue. Since the OCSP response is pre-fetched by the server, it may not always be up-to-date (the server might store the response for a few minutes). This delay creates a small window of opportunity for attacks. For example, an attacker could retrieve the valid certificate and OCSP response, then attack the server to steal the private key and set up a fake server. When users connect to this fake server, the attacker can still present a valid certificate and OCSP response until the browser accepts it (would a browser accept an OCSP response from 30 minutes ago?).

OCSP stapling is typically handled automatically by the server, though the client can explicitly request it. When the client connects, it may not know whether the server supports OCSP stapling, so the client can send a *Certificate Status Request* (CSR) as part of the *ClientHello* message to request the OCSP responses in the TLS handshake.

This process is optional: even if the client requests OCSP responses, the server may not support stapling. In this case, the server might return the OCSP responses for its certificate chain in a separate message called *CertificateStatus*.

The issues include:

- Servers may ignore the status request; they are not obligated to provide a response.
- Clients may proceed with the handshake even if no OCSP response is provided.

Although the technology exists, there is uncertainty about whether it is widely implemented. It is important to monitor the process to verify whether OCSP responses are being provided, requested, or ignored, as this offers insight into the actual security status, not just the theoretical one. To address this uncertainty, a newer solution called *OCSP Must Staple* has been introduced, which forces OCSP stapling.

1.18.1 OCSP Must Staple

When a server requests a certificate from a certification authority, it may declare that it will always provide OCSP responses to clients, even if not explicitly requested. This information can be included in the certificate. Such servers present a certificate to the client that contains a certificate extension called **TLSFeatures**, as defined in RFC-7633. This extension informs the client that every time it connects to that server, it must receive a valid OCSP response as part of the TLS handshake. If the client does not receive an OCSP response, it should reject the server's certificate. Essentially, the server promises to always send both its certificate and the OCSP response. If the server does not provide the OCSP response, it can be considered a fraudulent server.

The benefit of this approach is that the client no longer needs to query the OCSP responder, and it enhances security by preventing attacks that block OCSP responses for a specific client or DoS attacks against the OCSP responder.

Key actors and their responsibilities

- **Certification Authority (CA)**: Must include the **TLSFeatures** extension in the server certificate if requested by the server's owner.
- **OCSP Responder**: Must be available 24/7 to provide valid OCSP responses. This requires a robust infrastructure with multiple network access points, backup servers, and resilience against DoS attacks. Otherwise, servers would not be able to establish connections, as they must provide OCSP responses.
- **TLS Client**: Must send the Certificate Status Request (CSR) extension in the *ClientHello* message, recognize the OCSP Must-Staple extension (if present in the server's certificate), and reject the server's certificate if no OCSP response is provided when promised.
- **TLS Server**: Must pre-fetch and cache OCSP responses for a certain period, and include an OCSP response in the TLS handshake. It must also handle errors when communicating with OCSP responders, in case the responder is temporarily unavailable.

- **Server Administrators**: Should configure their servers to use OCSP Stapling and request a server certificate that includes the OCSP Must-Staple extension.

One potential issue is the duration of the OCSP stapled response (for example, Cloudflare has a validity of 7 days). A window of 7 days may be problematic if a server is compromised, as there is potential exposure during that time.

1.19 Questions and answers

Question 1

what of the following remarks regarding TLS are possibly true?

✓ has been established a session S1 that involved connection C1 and C2 at time t1

✓ has been established a session S2 that involved connection C1, C2 and C3, at different and non- overlapping time intervals

× connection C5 has been used initially in session S3 and then re-used in session S4 for performance-sake

× in connection C1.1 of session S1 has been used AES192, while in Connection C1.2 of the same session S1 has been used AES128 since regarded less sensible information

Question 2

focusing on TLS Connections and Sessions

✓ sessions are typically relatively long-life in respect to connections

× connections allow to re-use cryptographic parameters defined during handshake, thus reducing significantly the handshake phase

× by using resumption mechanisms, the client can resume a connections decreasing the overall connection time

✓ each connection has its own specific set of parameters like seguence numbers and keys for integrity and confidentiality

Question 3

focusing on Perfect Forward Secrecy

✓ starting from version TLS version 1.3, it must be always enabled

× using RSA mechanisms in TLS, it would be theoretically impossible to achieve

✓ using RSA mechanisms in TLS, it would be impractical to achieve

× using ECDH mechanisms in TLS, it would be impractical due to the length of the key parameters

Question 4

Which of the following properties is NOT directly related to Perfect Forward Secrecy in a TLS context?

- × the use of ephemeral keys
- ✓ protection against replay attacks
- × security of past sessions if the server's private key is compromised
- × independent session keys for each connection

Question 5

In the context of TLS 1.3, what is the role of PFS in session resumption mechanisms like O-RTT?

- × O-RTT resumption is vulnerable to replay attacks, thus does not support PFS
- × O-RTT session resumption uses pre-shared keys that provide Perfect Forward Secrecy.
- ✓ O-RTT session resumption compromises Perfect Forward Secrecy for faster handshakes.
- × O-RTT session resumption reuses the original session's ephemeral keys

Question 6

Which TLS feature was introduced to specifically mitigate downgrade attacks?

- × HSTS (HTTP Strict Transport Security)
- × OCSP Stapling
- × Certificate Pinning
- ✓ TLS FALLBACK SCSV

Question 7

Given the following packet capture:

- × indicate a
- × might be the initial phase of a secured real-time data exchange (like video streaming)
- × It refers to an aborted TLS session, due to the impossibility to verify the x509v3 certificate
- ✓ lt refers to an aborted TLS session, due to TLS version mismatch between the versions supported by the server and by the client

Chapter 2

SSH

SSH, or Secure Shell, is a network protocol that allows for secure communication between two computers. It has been introduced in 1995 by Tatu Ylönen, a Finnish computer scientist, but has been since commercialized so it was not open source anymore. A open source fork called OpenSSH has been created and is now the most used implementation of SSH. Nowadays is the most widely diffused version of SSH and is the one we will refer to in this document.

> Here we will always refer to SSH-2 (unless otherwise noted)

2.1 Architecture

SSH is a three layer architecture, which sits on top of TLS, which as alway provides initial connection, server authentication, confidentiality and integrity and key re-exchange (RFC-4253 recommends after 1GB of data transmitted or after 1 hour of transmission) which is crucial for long running connections.
there is also a specific protocol for user authentication, and a connection protocol too, which supports supports multiple connections (channels) over a single secure channel (implemented with the transport layer protocol). This means that SSH can as as a site-to-site VPN.

2.1.1 SSH Transport Layer Protocol

The connection is setup with TCP, by default on port 22, on which the client initiates the connection.
Then the SSH version string is exchanges, which contains the security features supported. Both sides must send a version string of the following form: SSH-protoversion-softwareversion SP comments CR LF. Its also used to triggers compatibility extensions.
All packets that follow the version string exchange is sent using the Binary Packet Protocol. Then the keys are exchanged, at which point the data can be exchanged and then the connection is closed.

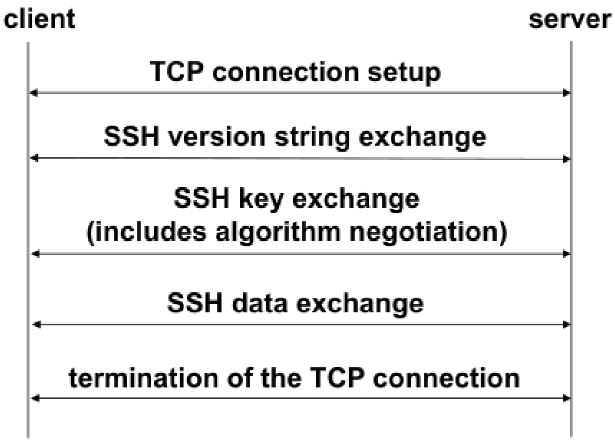

client
server
TCP connection setup
SSH version string exchange
SSH key exchange
(includes algorithm negotiation)
SSH data exchange
termination of the TCP connection

Figure 2.1: SSH Transport Layer Protocol

Binary Packet Protocol

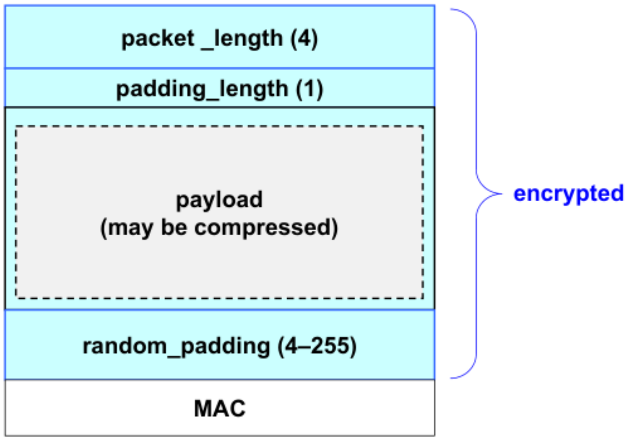


Figure 2.2: Binary Packet Protocol

The Binary Packet Protocol is pretty simple, it consists of:

- the length of the packet (4 bytes), which does not include the MAC and the packet length field itself
- the payload, which may be compressed. Its size is the length of the packet minus the length of the length field minus 1 (for the padding length field), up to a maximum uncompressed size of 32768 bytes(32KB)
- the random padding
- the MAC

By default the communication is in binary.

key exchange

Algorithm specification

Diffie-Hellman key exchange

This is also an implicit authentication of the server.
In this phase the client generates a random number x and computes $e = g^x \mod p$ and sends it to the server. The server generates a random number y and computes $f = g^y \mod p$ and sends it to the client. The server computes $K = e^y \mod p$ and exchange $H = \mathrm{HASH}(K)$. The server generates the signature on H using its private key and sends

key derivation

Many keyed digest require a initialization vector

session_id is the hash value computed for the first key exchange, and remains such even when key re-exchange is performed

Encryption

The supported algorithms are:

- (required) 3des-cbc (w/ three keys, i.e. 168 bit key)
- (recommended) aes128-cbc
- (optional) blowfish-cbc, twofish256-cbc, twofish192-cbc, twofish128-cbc, aes256-cbc, aes192-cbc, serpent256-cbc, serpent192-cbc, serpent128-cbc, arcfour, idea-cbc, cast128-cbc, none

MAC