Advanced Information Systems Security

Alessandro Milani

October 12, 2024

Contents

Chapter 1

TLS

1.1 Original SSL (Secure Socket Layer)

Secure transport channel originally proposed by Netscape Communications in 1995, it have has the following characteristics:

- **Peer authentication** (server, server+client) → asymmetric challenge-response (implicit, explicit)
- **Message confidentiality** → symmetric encryption
- **Message authentication and integrity** → MAC computation
- **Protection against replay, filtering, and reordering attacks** → implicit record number (used in MAC computation!) plus layering on TCP -¿ **For exam** tell that the record number is also included in the computation of the MAC

1.2 TLS achitecture

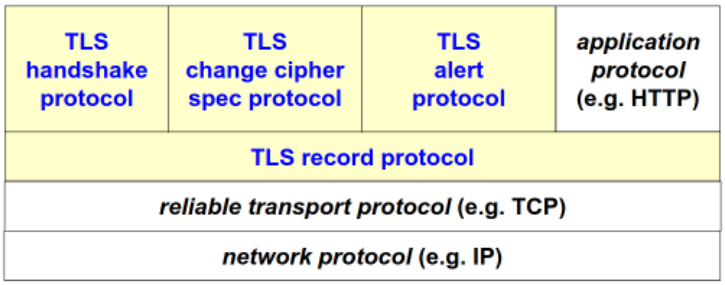


Figure 1.1: Arch TLS

- one **network protocol**, typically nowadays IP, but that is not compulsory (Anything at layer 3 can work)
- a reliable **transport protocol**, For example, TCP.
- Then the **TLS record protocol**: a generic protocol just for transporting things between the two peers. First it transport the **TLS handshake protocol**, then the **ChangeCypherSpec** (When you need to change from one algorithm to another, and notably from no protection to protection), when something bad happens one last message, which is part of the **alert protocol** (and then close the connection). If everything is going well, the record protocol is used to transport some **application protocol**.

1.3 TLS sessions and connections

In a **TLS session**, created by the handshake protocol, is a a logical association betweeen the client and server that agree on a set of cryptographic parameters (is shared between different connections 1:N)

A **TLS connection** is the transient TLS channel between the client and server, and is associated with a session.

1.4 TLS handshake protocol

Used to agree in a set of algorithms for confindentality, integrity. During the handshake are exchange random numbers between the client and the server to be used for the subsequent generation of the keys, use to establish a symmetric key by means of public key operations (RSA, DH, ...).
In the end are also negotiate the session-id and exchanged the necessary certificates.

1.5 Data protection

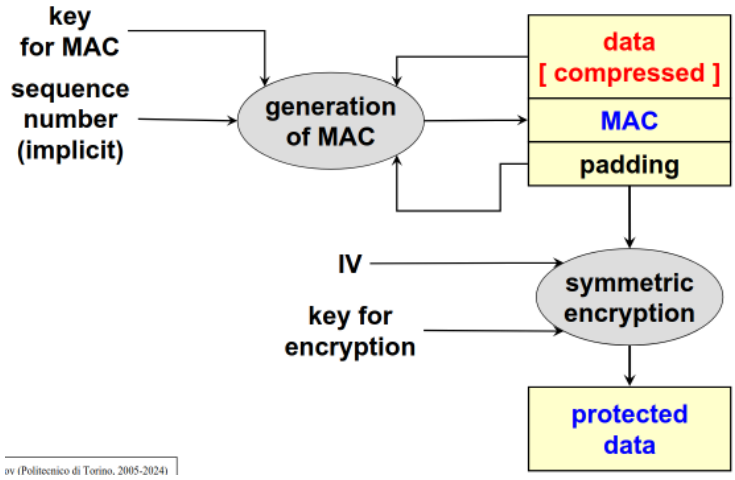
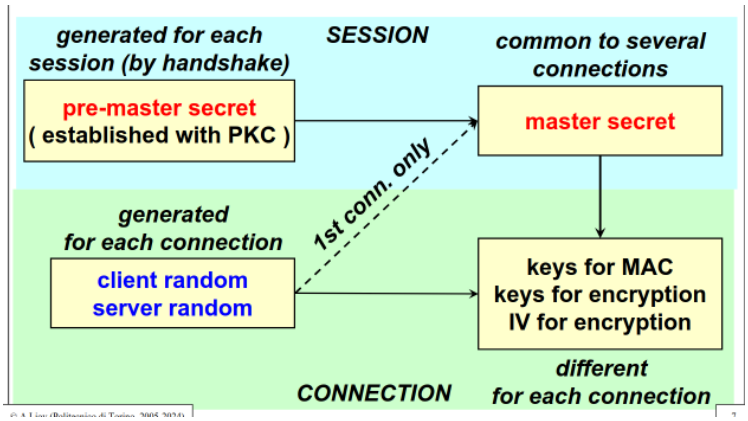


Figure 1.2: Authenticate then encrypt

- In authenticate-then-encrypt data that are coming from the application layer. Until TLS 1.2 it was possible to compress those data before transmitting and the negotiation also of compression algorithm was part of the handshake. (Can lead to attack so in new version of TLS is not possible to compress data)
- The keys are not unique, but are directional (one for encrypt server to client, and one for encrypt client to server). This is done because, is used the number of the packet in the sequence for accept it or not, and if the key is the same one part can copy a packet with id X and sent it back when his sequence arrive at the same number.

1.6 Relationship among key and sessions



The only value common at each connection is the master key used for generate keys for enc and IV. (all other value keys, iv are unique for each connection).
Every time you open a new channel, there are new client and server random that are used together with the master secret to create the specific keys for that connection.

1.6.1 Problem

Since the key material is generated starting from a symmetric crypto, we have a problem. If the private key used for performing encryption and decryption of the pre-master secret is discovered, then if someone has made a copy of all

the past traffic, will now be **able to decrypt all the traffic** (present, past and future).
We would like very much to have a specific property which is named perfect forward secrecy.

1.7 Perfect forward secrecy (PFS)

Perfect Forward Secrecy ensures that even if the session key used for key exchange is compromised, only current and eventually future communications are at risk, while past communications remain secure.

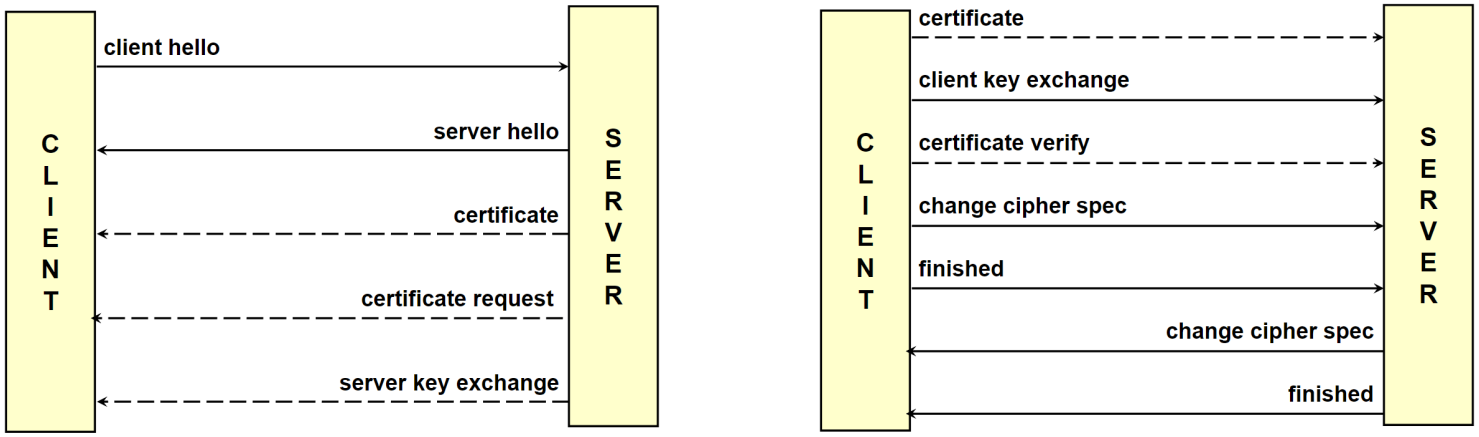
1.7.1 Ephemeral mechanisms

An ephemeral key exchange mechanism is used to generate a new session key on the fly, that need to be signed with the long term private key (but cannot have an associated X.509 certificate because the CA process is slow and often not on-line).
Usually used for Diffie-Hellman (DH) but is slow in RSA.

SO, we have the server's private key used only for signing and we obtain a perfect forward secrecy.

- If the (temporary or short-lived) private key is compromised then the attacker can decrypt only the related traffic
- Compromise of the long-term private key is an issue for authentication but not for confidentiality

!! Valutare immagini slide 10 e 11 !!
...



(a) Image 1

(b) Image 2

Figure 1.3: Steps of ephemeral key exchange

Client Hello

In the "Client Hello" message, the client begins by specifying its preferred SSL/TLS version, indicating the highest version it supports (e.g., 2 for SSL-2, 3.0 for SSL-3, 3.1 for TLS-1.0). It also includes 28 pseudo-random bytes known as the "Client Random," and a session identifier (session-id). (If the session-id is 0 → requested a new session. Non-zero value → resume a previous session). The message also contains a list of supported cipher suites, which define the algorithms for encryption, key exchange, and integrity, and a list of supported compression methods.

Server Hello

In the "Server Hello" message, the server responds by selecting the SSL/TLS version to use. It then sends 28 pseudo-random bytes known as the "Server Random" and provides a session identifier. If the session-id in the client hello was 0, the server generates a new session-id or rejects the session-id proposed by the client. If the server accepts the session resumption request, it echoes the session-id provided by the client. The server also selects the strongest cipher suite that is common with the client, and it specifies the compression method to be used.

Cipher suite

- key exchange algorithm
- symmetric encryption algorithm
- hash algorithm (for MAC)
- examples:
  - SSL_NULL_WITH_NULL_NULL
  - SSL_RSA_WITH_NULL_SHA
  - SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
  - SSL_RSA_WITH_3DES_EDE_CBC_SHA

  complete list maintained by IANA

Certificate (server)

In the **"Certificate"** message from the server, a certificate is provided for **server authentication**. The certificate's **subject** or **subjectAltName** must match the server's identity, such as its **DNS name** or **IP address**. The server must send the **entire certificate chain** up to a trusted root, though the root CA itself should not be included.
The certificate can be used for **signing**, or both **signing and encryption**, as specified in the **keyUsage** field.
If the certificate is only for signing, a separate **server key exchange** phase is required to exchange the ephemeral key.

Certificate Request

- used for client authentication
- specifies also the list of CAs considered trusted by the server
  - the browsers show to the users (for a connection) only the certificates issued by trusted CAs

Server key exchange

The **Server Key Exchange** message is used to carry the server's public key for key exchange. It is required in the following cases:

- The RSA server certificate is usable only for **signature**.
- **Anonymous** or **ephemeral Diffie-Hellman (DH)** is used to establish the pre-master secret.
- There are **export restrictions** requiring the use of ephemeral RSA/DH keys.
- **Fortezza ephemeral** keys are used.

This message is **explicitly signed by the server**, and it is important to note that it is the **only message** in the handshake process that is explicitly signed by the server.

Certificate (client)

This message carries the certificate for client authentication. The certificate must have been issued from one of the CAs in the trusted CA list specified in the Certificate Request message.

Client key exchange

The client generates material for symmetric keys derivation and sends it to the server in varius ways:

- pre-master secret encrypted with the server RSA public key (ephemeral or from its X.509 certificate)
- client's public part of DH
- client's Fortezza parameter

Certificate verify

The **Certificate Verify** message involves an explicit test signature performed by the client. In this phase, a hash is computed over all the previous handshake messages and is then encrypted using the client's private key. This step is required only when **client authentication** is used, allowing the server to identify and reject fake clients.

Change cipher spec

The **Change Cipher Spec** message signals the switch to the newly negotiated algorithms for message protection. It transitions from unprotected messages to encrypting subsequent messages using the agreed-upon algorithms and keys.
While it is theoretically considered a separate protocol from the handshake, some analyses suggest that it could potentially be **eliminated** from the process.

Finished

The **Finished** message is the first message protected with the negotiated algorithms. It plays a crucial role in authenticating the entire handshake process.
The message contains a **MAC** computed over all the previous handshake messages (excluding the **Change Cipher Spec**), using the **master secret** as the key. This ensures the integrity of the handshake and prevents rollback **man-in-the-middle attacks** such as version or cipher suite downgrade. The **Finished** message is different for the client and server, with each calculating their own MAC.

1.8 TLS Comparison

- TLS, no ephemeral key, no client auth
- TLS, no ephemeral key, client auth
- TLS, ephemeral key, no client auth
- TLS, data exchange and link teardown
- TLS, resumed session

!! immagini da slide 23 a 27, valutare se inserirle !!

1.9 TLS setup time

The **TLS setup time** involves both the TCP and TLS handshakes. First, a **TCP handshake** is performed, followed by the **TLS handshake**. Multiple TLS messages can fit within a single TCP segment. Typically, this setup requires **1-RTT** for TCP and **2-RTT** for TLS:

- (C > S) **SYN**

- (S > C) **SYN-ACK**
- (C > S) **ACK** + **ClientHello**
- (S > C) **ServerHello** + **Certificate**
- (C > S) **ClientKeyExchange** + **ChangeCipherSpec** + **Finished**
- (S > C) **ChangeCipherSpec** + **Finished**

After around **180 ms** (assuming a 30 ms one-way delay), the client and server are ready to exchange protected data.

1.10 TLS versions

1.10.1 TLS 1.0 (SSL 3.1)

Transport Layer Security (TLS) is a standard defined by the IETF, with **TLS 1.0** specified in *RFC 2246* (January 1999). TLS 1.0 is also known as **SSL 3.1**, sharing a 99% similarity with SSL 3.
This version places a strong emphasis on using standard (i.e., non-proprietary) digest and asymmetric cryptographic algorithms, which are mandatory. The required algorithms include **Diffie-Hellman (DH)**, **Digital Signature Algorithm (DSA)**, and **3DES** for encryption, as well as **HMAC-SHA1** for message authentication.
That lead to the ciphersuite *TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA*.

1.10.2 TLS 1.1

TLS 1.1 is defined in *RFC 4346* (April 2006) and introduces several enhancements to improve security. To protect against **Cipher Block Chaining (CBC)** attacks, the implicit initialization vector (IV) is replaced with an explicit IV.
Additionally, padding errors now trigger the **bad_record_mac** alert message instead of the previous **decryption_failed** alert (This is for give less information to an attacker that is trying a padding oracle attack).
The update also establishes IANA registries for various protocol parameters. Notably, premature session closes no longer result in sessions being non-resumable, allowing for improved session management.
Furthermore, TLS 1.1 includes additional notes addressing various new attack vectors.

1.10.3 TLS 1.2

TLS 1.2 is specified in *RFC 5246* (August 2008) and introduces significant improvements in cryptographic functions and security features. One of the key enhancements is that the ciphersuite now specifies the **pseudo-random function (PRF)** used in the protocol. TLS 1.2 extensively employs **SHA-256** for various operations, including the **Finished** message and HMAC.
The protocol also adds support for **authenticated encryption**, utilizing **AES** in GCM or CCM mode. Additionally, TLS 1.2 incorporates protocol extensions from *RFC 4366* and the AES ciphersuite from *RFC 3268*.
The default ciphersuite is *TLS_RSA_WITH_AES_128_CBC_SHA*, while older ciphersuites such as **IDEA** and **DES** have been deprecated.

1.11 TLS evolution

1.11.1 Ciphersuites and Encryption

The evolution of TLS has introduced various ciphersuites and encryption algorithms. Notable encryption algorithms include:

- **AES** specified in *RFC 3268*
- **ECC** (Elliptic Curve Cryptography) introduced in *RFC 4492*
- **Camellia** defined in *RFC 4132*

- **SEED** as outlined in *RFC 4162*
- **ARIA** specified in *RFC 6209*

In terms of authentication, the following methods have been integrated:

- **Kerberos** as described in *RFC 2712*
- **Pre-shared Key** (secret, DH, RSA) introduced in *RFC 4279*
- **SRP** (Secure Remote Password) detailed in *RFC 5054*
- **OpenPGP** specified in *RFC 6091*

1.11.2 Compression and Other Features

The TLS evolution also includes advancements in compression methods:

- Compression methods and **Deflate** defined in *RFC 3749*
- Protocol compression using **LZS** as per *RFC 3943*

Additionally, several other protocols and extensions have been established:

- **Extensions** (specific and generic) as specified in *RFC 4366*
- **User mapping extensions** detailed in *RFC 4681*
- **Renegotiation indication extensions** outlined in *RFC 5746*
- **Authorization extensions** described in *RFC 5878*
- Prohibition of **SSL 2.0** as per *RFC 6176*
- **Session resumption** without server state as outlined in *RFC 4507*
- **Handshake with supplemental data** specified in *RFC 4680*

!! vedere attacchi da slide 34 a 41 !!

1.12 Attacks

1.12.1 Heartbleed

The **Heartbleed** vulnerability is tied to the **RFC 6520**, which defines the TLS/DTLS heartbeat extension. This extension was designed to keep a connection alive without the need for constant renegotiation of the SSL session, particularly useful for **DTLS** (Datagram TLS) and in **Path Maximum Transmission Unit (PMTU)** discovery.

However, the vulnerability, assigned **CVE-2014-0160**, stems from a bug in OpenSSL. Due to a **buffer over-read** issue, the TLS server may send back more data (up to 64kB) than originally requested in the heartbeat message. This can expose sensitive data from the server's RAM, such as user credentials (username and password) or even the server's private key—if the key is not stored in a secure hardware module like an **HSM** (Hardware Security Module).

1.12.2 Bleichenbacher attack (and ROBOT)

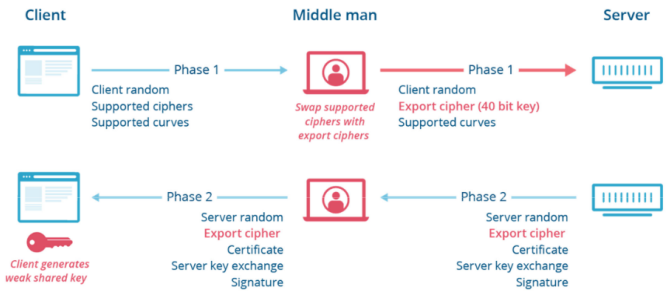
In **1998**, **Daniel Bleichenbacher** introduced the so-called "million-message attack," a cryptographic vulnerability targeting the way **RSA encryption** was implemented. By sending approximately a million specially crafted messages to a server, an attacker could deduce the server's **private key** through subtle differences in the returned **error codes**.

Over the years, this attack has been refined to require far fewer messages—in some cases, just thousands—making it feasible to execute from a laptop. In **2017**, the **ROBOT (Return Of Bleichenbacher's Oracle Threat)** attack, a variant of the original, emerged and affected major websites, including **facebook.com**.

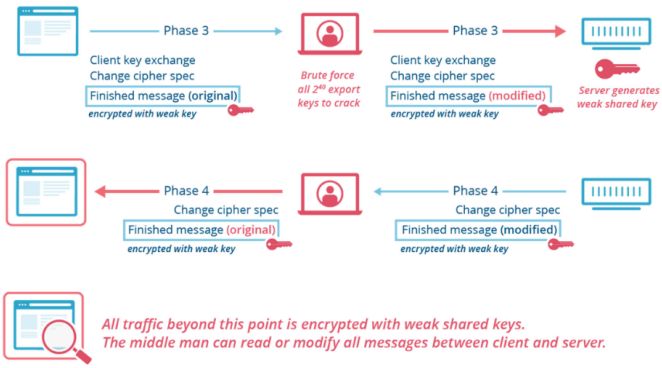
1.12.3 Other minor attacks

- **CRIME** (2012)
  - Attacker can:
    1. Inject chosen plaintext in user requests
    2. Measure the size of the encrypted traffic
  - Exploits information leaked from compression to recover specific plaintext parts.
- **BREACH** (2013)
  - Deduces a secret within HTTP responses from a server that:
    1. Uses HTTP compression
    2. Inserts user input into HTTP responses
    3. Contains a secret (e.g., CSRF token) in the HTTP responses
- **BEAST** (Browser Exploit Against SSL/TLS) 2011
  - SSL channel using CBC with IV concatenation
  - A MITM attacker may decrypt HTTP headers with a blockwise-adaptive chosen-plaintext attack
  - Attacker may decrypt HTTPS requests and steal session cookies or other information.
- **POODLE** (Padding Oracle On Downgraded Legacy Encryption)
  - A MITM attack that exploits SSL-3 fallback to decrypt data.
  - Dec 2014 variant: Exploits CBC errors in TLS 1.0–1.2, so it works even if SSL-3 is disabled.
- **FREAK** (Factoring RSA Export Keys) 2015
  - Downgrade to export-level RSA keys (512-bit) and factorize to decrypt the channel.
  - Alternatively, downgrade to export-level symmetric keys (40-bit) and brute-force the key.

1.12.4 Freak phases



(a) Image 1



(b) Image 2

Figure 1.4: Freak Attack step by step

1.13 ALPN extension (Application-Layer Protocol Negotiation)

is an extension that can be added to tls message and give the possibility to negotitate the application layer protocol to use to speed up the connection creation, avoiding additional round-trips for application negotiation:

- (ClientHello) ALPN=true + list of supported app. protocols
- (ServerHello) ALPN=true + selected app. protocol

Important for the negotiaton of the HTTP/2 (for FX and chrome is onlt over TLS) and QUIC protocol, but also useful also for those servers that use different certificates for the different application protocols.

possible value: http/1.0, http/1.1, h2 (HTTP/2 over TLS), h2c (HTTP/2 over TCP)

1.14 TLS False Start

Another extension, whit it he client can send application data together with the ChangeCipherSpec and Finished messages, in a single segment, without waiting for the corresponding server messages.
This lead to recude the latency to 1-RTT, but even if it can work without changes, chome and FX require ALPN + forware secrecy. Safari only require forwared secrecy.
For enable it, the server must advertise the supported protocols and be configured to prefer cipher suites with forward secrecy.

1.15 The TLS Downgrade Problem

During a TLS handshake, the client sends (via **ClientHello**) the highest TLS version it supports. The server then replies (in **ServerHello**) with the version it will use, which is the highest version supported by both the client and server. In a typical scenario of version negotiation, both parties may agree on **TLS 1.2**:

- Client to Server: 3,3 (indicating TLS 1.2)
- Server to Client: 3,3 (agreement on TLS 1.2)

If the server does not support **TLS 1.2**, a fallback to **TLS 1.1** occurs:

- Client to Server: 3,3 (requests TLS 1.2)
- Server to Client: 3,2 (fallback to TLS 1.1)

In some cases, servers do not respond correctly, potentially closing the connection instead of negotiating the correct version. This forces the client to retry the handshake with a lower protocol version. This behavior can be exploited in a **downgrade attack**, where an attacker sends a fake server response to force multiple downgrades. Eventually, this may result in the connection being established using a vulnerable version (e.g., **SSL 3.0**), which could then be exploited with an attack like **Poodle**. Although downgrades do not always signify an attack, they present security risks when exploited maliciously.

1.16 TLS Fallback Signalling Cipher Suite Value (SCSV)

Used for prevents protocl downgrade attacks. It introduce a new dummy chipersuire **TLS_FALLBACK_SCSV** that is used by the client to signal to the server that the connection is a downgraded connection.
The server can respond with a new fatal Alert **in_appropriate_fallback** and close the connection if it recive TLS_FALLBACK_SCSV and a version lower than the highest one supported. At this point the client can retry the connection with the highest version supported.

1.16.1 Notes

It's not support by a lot of server, but most servers have fixed their bad behaviour when the client requests a version higher than the supported one so browsers can now disable insecure downgrade

1.17 TLS session tickets

Another improvment in TLS, that allow the server to send the session datat to the client (to reduce the cache on the server).
The session data in the client is:

- encrypted with a server secret key
- returned to the server when resuming the session
- it just move the session ache to the client

There are some issues:

- needs support at the browser (it's an extension)
- in a load balancing environment, it requires key sharing among the various end-points (and periodic key update!)

1.18 TLS and virtual servers

A virtual server is a server with a single IP address and multiple domain names associated to it.
It'e easy to manage in HTTP/1.1 because in the headerthere is the host field that specify the domain name.

1.18.1 The problem

In https it's difficult because TLS is activated before HTTP, so there is no way to know that certificare use.

1.18.2 The solution

- **Collective (wildcard) certificate**: A single certificate covers multiple subdomains, e.g., `CN=*.myweb.it`, with the private key shared among all servers. However, different browsers may handle wildcard certificates differently.
- **Certificate with a list of servers in subjectAltName**: This method includes a list of servers within the `subjectAltName` field of the certificate, with the private key shared across servers. A drawback is that the certificate needs to be reissued whenever a server is added or removed.
- **SNI (Server Name Indication)**: The `Server Name Indication` (SNI) extension, included in the **ClientHello** message (as per *RFC 4366*), allows specifying which server the client is trying to reach. However, SNI has limited support among certain browsers and servers.

1.19 TLS 1.3

The goals of TLS 1.3 are to:

- reducing handshake latency
- encrypting more of the handshake (for security and privacy)
- improving resiliency to cross-protocol attacks
- removing legacy features

1.19.1 key exchange

TLS 1.3 introduced significant changes to the key exchange process to improve security:

- **Removal of static RSA and DH key exchange**: These methods were eliminated because they do not provide forward secrecy. They also introduced vulnerabilities such as the **Heartbleed** attack and were difficult to implement correctly. Additionally, they were susceptible to the **Bleichenbacher attack** (and its variant, **ROBOT**).
- **Use of DHE (Diffie-Hellman Ephemeral)**: TLS 1.3 exclusively uses **DHE** for key exchange but restricts the use of arbitrary parameters to prevent attacks. For example:
  - **LogJam (2015)** tricked servers into using weak DH parameters (512-bit).
  - **Sanso (2016)** found that OpenSSL generated DH values without the required mathematical properties.
- **Predefined groups**: To mitigate these risks, TLS 1.3 uses only **DHE** with a few predefined, secure groups, ensuring better protection against cryptographic attacks.

1.19.2 message protection

TLS 1.3 addressed several vulnerabilities from earlier versions by improving message protection and cryptographic methods:

Previous Pitfalls

- **CBC mode and authenticate-then-encrypt**: These techniques were vulnerable to attacks like **Lucky13**, **Lucky Microseconds**, and **POODLE**.
- **RC4 usage**: In 2013, researchers demonstrated that **RC4** could expose plaintext due to measurable biases in its output.
- **Compression**: The use of compression in TLS was a factor in the **CRIME** attack, which exploited compressed data to leak information.

TLS 1.3 Improvements

- TLS 1.3 no longer uses **CBC** or authenticate-then-encrypt schemes. Instead, it only permits **AEAD** (Authenticated Encryption with Associated Data) modes, which offer strong integrity and confidentiality.
- Deprecated older cryptographic methods such as **RC4**, **3DES**, **Camellia**, **MD5**, and **SHA-1**.
- Compression is no longer used in TLS 1.3, preventing compression-related attacks.
- Only modern, secure cryptographic algorithms are supported to ensure the safety of data transmission.

1.19.3 digital signature

previous pitfalls:

- RSA signature of ephemeral keys
- done wrongly with the PKCS#1v1.5 schema
- handshake authenticated with a MAC, not a signature
- makes possible attacks such as FREAK

TLS-1.3 uses:

- RSA signature with the modern secure RSA-PSS schema
- the whole handshake is signed, not just the ephemeral keys
- modern signature schemes

1.19.4 cipher suites

TLS 1.3 simplifies the complexity of ciphersuites from previous versions by reducing the number of options and focusing on secure, modern cryptographic methods:

- **Avoids combinatorial complexity**: In previous TLS versions, the ciphersuite list grew exponentially as each new algorithm was added. TLS 1.3 addresses this by specifying only the essential orthogonal elements.
- **Orthogonal elements**: TLS 1.3 ciphersuites specify only two key components:
  - **Cipher (and mode)**: E.g., AES, ChaCha20.
  - **HKDF hash**: The hash used for the HMAC key derivation.
- TLS 1.3 does not specify certificate types (RSA, ECDSA, EdDSA) or key exchange methods (DHE/ECDHE, PSK, PSK+DHE/ECDHE), reducing complexity.
- **Only 5 ciphersuites** are supported in TLS 1.3:
  - `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)

1.19.5 EdDSA

Variant of the DSA.
DSA requires a PRNG that can leak the private key if the underlying generation algorithm is broken or made predictable, otherwire **edSA does not need a PRNG**, but **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 more need for random number generators.
This lead to a **faster signature** and verification wrt ECDSA (simplified point addition and doubling) and in general it use N-bit private and public keys, 2N-bit signatures.

EdDSA implementation

- **Ed25519**: This implementation uses **SHA-512** (SHA-2) and **Curve25519**. It provides:
  - 256-bit key
  - 512-bit signature
  - 128-bit security
- **Ed448**: This version employs **SHAKE256** (SHA-3) and **Curve448**, offering:
  - 456-bit key
  - 912-bit signature
  - 224-bit security
- **Curve25519** is the most widely used elliptic curve, defined over the field $2^{255} - 19$, and is also used in **X25519** for ECDH (Elliptic-Curve Diffie-Hellman).

A minor problem is that **EdDSA has two standards** slighly different

- **RFC-8032**: Defines EdDSA for general Internet applications, leaving many implementation details up to developers.
- **FIPS 186-5**: Specifies stringent guidelines for key management, generation, and secure implementation practices.

1.19.6 Other improvements

- **Encrypted Handshake Messages**: All handshake messages following the `ServerHello` are now encrypted, enhancing security and confidentiality.
- **EncryptedExtensions Message**: A new message, `EncryptedExtensions`, ensures that extensions previously sent in plaintext during the `ServerHello` now benefit from encryption.
- **Redesigned Key Derivation Functions**: TLS 1.3 uses a redesigned key derivation process to improve cryptographic analysis thanks to key separation, leveraging **HKDF** as the underlying primitive.
- **Restructured Handshake State Machine**: The handshake process has been streamlined to be more consistent, removing unnecessary messages like `ChangeCipherSpec`, except in cases where it's needed for middlebox compatibility.

1.20 HKDF

The **HKDF** is a two-stage process used for key derivation, consisting of `HKDF-Extract` and `HKDF-Expand`:

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

1. **Extract Stage**: The input keying material (**IKM**) is used to generate a fixed-length pseudorandom key (**PRK**)
2. **Expand Stage**: The PRK is then expanded into multiple pseudorandom keys. This is done by calling HMAC repeatedly, using the PRK as the key and appending an incrementing 8-bit counter to the message, chaining the results:

$$\text{HMAC}(\text{PRK}, \text{info} \parallel \text{counter})$$

The "info" field can vary, allowing the generation of multiple outputs from a single IKM value. Each output key is a result of the chaining process, where the previous hash block is prepended to the next HMAC input.

1.20.1 HKDF in TLS 1.3

In TLS 1.3, HKDF is employed for various key derivation tasks. The function `HKDF-Expand-Label` is used to derive keys by incorporating specific parameters and labels to ensure separation of cryptographic keys.

HKDF-Expand-Label

```
HKDF-Expand-Label( Secret, Label, Context, Length ) = HKDF-Expand( Secret, HkdfLabel, Length )
```

HkdfLabel is structured as:

```
struct {
    uint16 length = Length;
    opaque label<7..255> = "tls13 " + Label;
    opaque context<0..255> = Context;
} HkdfLabel;
```

Key Derivation Functions

- **Derive-Secret**:

  ```
  Derive-Secret( Secret, Label, Messages ) = HKDF-Expand-Label( Secret, Label, Transcript-Hash(Mess
  ```

- **Finished Key**:

  ```
  finished_key = HKDF-Expand-Label( BaseKey, "finished", "", Hash.length )
  ```

- **Ticket PSK**:

  ```
  ticket_PSK = HKDF-Expand-Label( resumption_master_secret, "resumption", ticket_nonce, Hash.length )
  ```

These derivations ensure secure key management and cryptographic operations by binding the keys to specific TLS 1.3 contexts (e.g., finished messages, resumption tickets).

1.21 TLS 1.3 Handshake

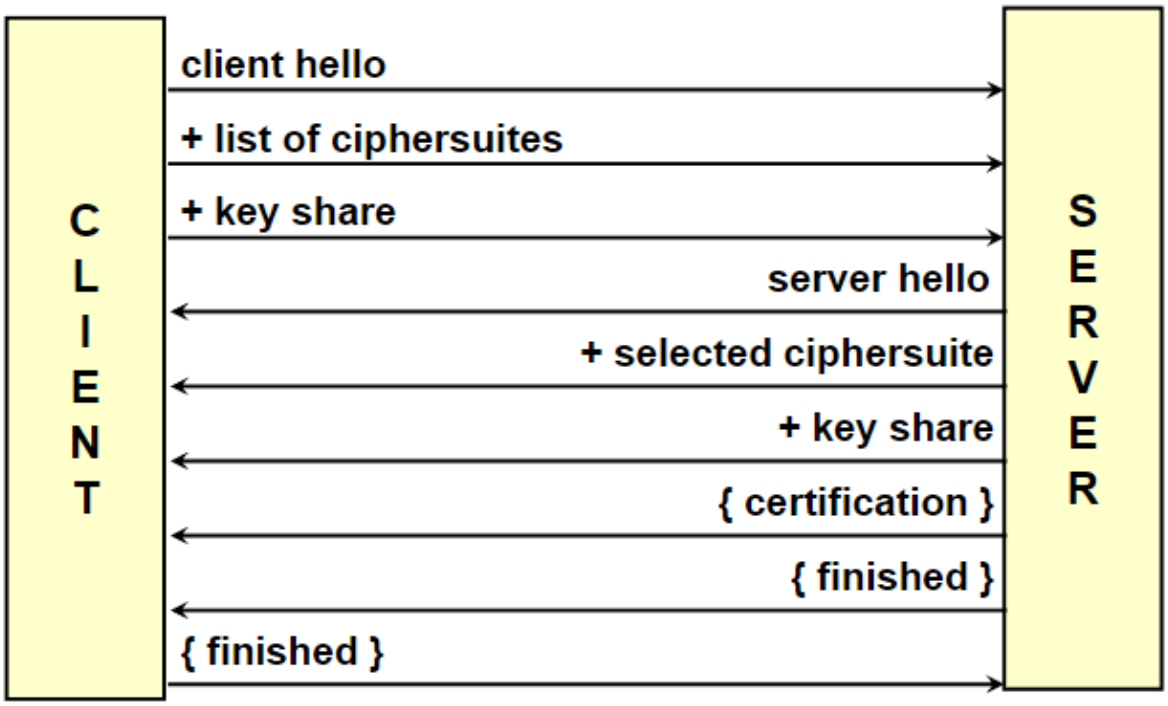


Figure 1.5: TLS 1.3 Handshake

1.21.1 Notes

For backward compatibility, TLS-1.2 messages are also sent, and most TLS-1.3 features are in message extensions. Basically, it's a 1-RTT handshake, that can be reduced to 0-RTT upon resumption of a previous session (or by using true PSK, which is rare)
notation:

- { data } = protected by keys derived from a

  sender

  _handshake_traffic_secret

- $$data$$

  = protected by keys derived from a

  $$sender$$

  _application_traffic_secret_N

1.21.2 client request

Client Hello:

- client random
- highest supported protocol version (note: TLS-1.2 !)
- supported ciphersuites and compression methods
- session-ID

contains extensions for key exchange:

- key_share = client (EC)DHE share
- signature_algorithms = list of supported algorithms
- psk_key_exchange_modes = list of supported modes
- pre_shared_key = list of PSKs offered

1.21.3 Server Response

The server response during the TLS-1.3 handshake includes several critical components that facilitate secure communication.

- **ServerHello**: This message contains:
  - `server random` - a random value generated by the server.
  - `selected version` - the TLS version chosen for the session.
  - `ciphersuite` - the cipher suite selected for encryption and authentication.
- **Key Exchange**:
  - `key_share` - the server's (EC)DHE share, which is used for key exchange.
  - `pre_shared_key` - the selected PSK (Pre-Shared Key).
- **Server Parameters**:
  - { **EncryptedExtensions** } - responses to non-crypto client extensions.
  - { **CertificateRequest** } - a request for the client's certificate.
- **Server Authentication**:
  - { **Certificate** } - the server's X.509 certificate (or raw key, RFC-7250).
  - { **CertificateVerify** } - a signature covering the entire handshake to validate the server's identity.
  - { **Finished** } - a message that contains a MAC (Message Authentication Code) over the entire handshake to ensure integrity.
- **Application Data**:

1.21.4 client finish

1.21.5 Pre Shared Key

PSK replaces session-ID and session ticket (one or more PSKs agreed in a full handshake and re-used for other connections).
With ECDHE can be used for forward secrecy.
It's also used for OOB, but this is risky (and discouraged) if have insufficient randomness (possible brute force attack).

1.21.6 0-RTT connection

TLS 1.3 - 0-RTT Connections

In TLS 1.3, when using a Pre-Shared Key (PSK), the client can send **early data** along with its first message (client request). This early data is protected using a specific key called the `client_early_traffic_secret`.

However, 0-RTT does not provide **forward secrecy**, as the security depends solely on the PSK. This also opens up the possibility of **replay attacks**. While partial mitigations are feasible, they are complex, especially in the case of multi-instance servers.

1.21.7 Incorrect Share

In TLS 1.3, the client can send a list of (EC)DHE groups that are not supported by the server. In this case, the server will respond with a **HelloRetryRequest**, and the client must restart the handshake with a different group. If the newly proposed groups are also unacceptable to the server, the handshake will be aborted, and the server will send an appropriate alert.

1.22 TLS and PKI

PKI needed for server (and optionally) client authentication, unless PSK authentication is adopted.
When a peer sends its certificare:

- The whole chain is needed (but the root CA - beware!)
- Validate the whole chain (not just the EE certificate)
- Revocation status needed at each step of the chain

To check the revocation staus:

- CRL can be used but big size and lengthy look-up
- OCSP (faster) can be used but generates privacy problems (leaks client navigation history)
- both require one additional network connection and add delay (e.g. for OCSP +300ms median, +1s average)

1.22.1 TLS and certificate status

When dealing with TLS and certificate validation, a critical concern arises when the URLs for the Certificate Revocation List (CRL) or the Online Certificate Status Protocol (OCSP) are unreachable. This situation can occur due to **various reasons**, including:

- **Server Error**: The server hosting the CRL or OCSP service is experiencing issues.
- **Network Error**: Connectivity problems preventing access to the revocation service.
- **Access Blocked by Firewall**: Security policies or insecure channels (common with OCSP) may block access.

Possible Approaches to handle these situations:

- **Hard Fail**: The page is not displayed, and a security warning is issued to the user.
- **Soft Fail**: The page is displayed under the assumption that the certificate is valid.

Both approaches typically require additional load time while waiting for the connection to timeout, which can affect user experience. Therefore, selecting the appropriate method for certificate status handling is essential for maintaining both security and usability in TLS connections.

Pushed CRL

Revoked certificates are often generated by compromised intermediate CA.
Browser vendors thus decided to push (some) revoked certificates:

- Internet Explorer (with browser update – bad, can be blocked)
- Firefox - oneCRL (part of the blocklisting process)
- Chrome (also Edge, Opera) - CRLsets

```
$ curl https://firefox.settings.services.mozilla.com/v1/buckets/security-state/collections/onecrl/
$ jq '[.data[] | {enabled,issuerName,serialNumber} | select(.enabled) | del(.enabled)]' crl.json >
```

Some articles about the Microsoft certificate problems with Windows XP / Internet Explorer 6:

- Unauthorized Microsoft Digital Certificates
- Microsoft Security Advisory 2718704

1.23 OCSP stapling

1.23.1 Concept

Certificate Revocation Lists (CRL) and Online Certificate Status Protocol (OCSP) automatic downloads are often disabled. When a client sends an OCSP request, it creates a **privacy issue**. Furthermore, pushed CRLs contain only a subset of revoked certificates, and browser behavior in handling these is highly variable.

The solution is **OCSP stapling**, where the server autonomously obtains the OCSP response and sends it along with its certificate to the client.

1.23.2 Implementation

OCSP Stapling is a TLS extension, and it must be specified in the TLS handshake. The first version is defined in **RFC 6066** with the extension `status_request`, and version 2 is defined in **RFC 6961** as `status_request_v2`, with the value `CertificateStatusRequest`.

How it Works

The TLS server pre-fetches OCSP responses and provides them to the client during the handshake, as part of the server's certificate message. These OCSP responses are "stapled" to the certificates.

Benefit: This eliminates the client's privacy concern and removes the need for the client to connect to an OCSP responder.

Downside: The freshness of the OCSP responses is a potential issue.

Client Request for OCSP Stapling

The TLS client **MAY** send a **Certificate Status Request (CSR)** to the server as part of the `ClientHello` message to request the transfer of OCSP responses in the TLS handshake. If the server receives a `status_request_v2` in the `ClientHello`, it **MAY** return OCSP responses for its certificate chain, which are sent in a new message called `CertificateStatus`.

Problems and Solutions

- Servers **MAY** ignore the status request.
- Clients **MAY** decide to continue the handshake even if OCSP responses are not provided.

The solution to these problems is the use of **OCSP Must Staple**, where the server is required to provide OCSP responses with the certificates.

1.24 OCSP Must Staple

In **OCSP Must Staple**, X.509 certificates for servers **MAY** include a certificate extension called `TLSFeatures`, which is identified by the OID `1.3.6.1.5.5.7.1.24` and defined in **RFC 7633**.

This extension informs the client that it **MUST** receive a valid OCSP response as part of the TLS handshake. If the client does not receive a valid OCSP response, it **SHOULD** reject the server's certificate.

Benefits:

- **Efficiency:** The client does not need to query the OCSP responder directly.
- **Attack Resistance:** It prevents attackers from blocking OCSP responses for a specific client or launching a denial-of-service (DoS) attack against the OCSP responder.

1.24.1 actors and duties

The **CA** must include the `TLSFeatures` extension into the server's certificates, if requested by the server's owner.

The **OCSP Responder** must:

- Be available 24/7 (365x24).
- Return valid OCSP responses promptly.

The **TLS client** must:

- Send the `Certificate Status Request (CSR)` extension in the `ClientHello` message.
- Understand the OCSP Must Staple extension if it is present in the server's certificate.
- Reject the server's certificate if it does not receive an OCSP stapled response.

The **TLS server** must:

- Support OCSP Stapling by prefetching and caching OCSP responses.
- Provide an OCSP response during the TLS handshake.
- Handle errors in communication with the OCSP responders.

The **TLS server administrators** should:

- Configure their servers to use OCSP Stapling.
- Request a server certificate with the OCSP Must Staple extension.

Open Issue: Duration of OCSP Responses. One potential pitfall is the duration of the OCSP stapled response. For example, Cloudflare OCSP responses last for 7 days.

1.25 TLS status

- F5 telemetry report (October 2021) for the top 1M servers:
- 63% have TLS-1.3 (from 80% USA to 15% China and Israel)
- 25% certs use ECDSA and 99% choose non-RSA handshake
- 52% permit RSA, 2.5% expired certs, 2% permit SSL-3
- Encryption is abused: 83% of phishing sites use valid TLS and 80% of sites are hosted by 3.8% hosting providers
- SSLstrip attacks still successful, so urgent need for HSTS or to completely disable plain HTTP
- Cert revocation checking mostly broken, which pushes for very short-lived certs
- By TLS fingerprinting, 531 servers potentially match the identity of Trickbot malware servers, and 1,164 match Dridex servers

Chapter 2

SSH

2.1 Introduction

2.1.1 history

First version of SSH by Tatu Ylönen in 1995 as a response to a hacking incident (sniffer on backbone, thousands of user+pwd copied).
In the same year was founded the company SSH Communications Security.
in 1999 a OpenSSH fork appears in ObenBSD 2.6.
The actual version is SSH-2 by IETF (Internet Engineering Task Force) in 2006, that is:

- incompatible with SSH-1
- improvements in security and features
- frequently is the only version supported nowadays

2.2 Architecture

SSH (Secure Shell) is based on a **three-layer architecture**, which includes the following protocols:

- **Transport Layer Protocol:**
  - Establishes the initial connection.
  - Provides server authentication.
  - Ensures confidentiality and integrity with **Perfect Forward Secrecy (PFS)**.
  - Supports key re-exchange (as per **RFC 4253**, it is recommended after 1 GB of data transmission or 1 hour of communication).
- **User Authentication Protocol:**
  - Authenticates the client to the server.
- **Connection Protocol:**
  - Supports multiple connections (channels) over a single secure channel, which is implemented by the Transport Layer Protocol.

2.2.1 Transport Layer Protocol

During the TCP connection setup, the server listens on port 22 (default) and the client is the one that initiates the connection. After this, both sides must send a version string of the following form: SSH-protoversion-softwareversion SP commer
used to inidcate the capabilities of an implementation (riggers compatibility extensions).
All packets that follow the version string exchange is sent using the Binary Packet Protocol

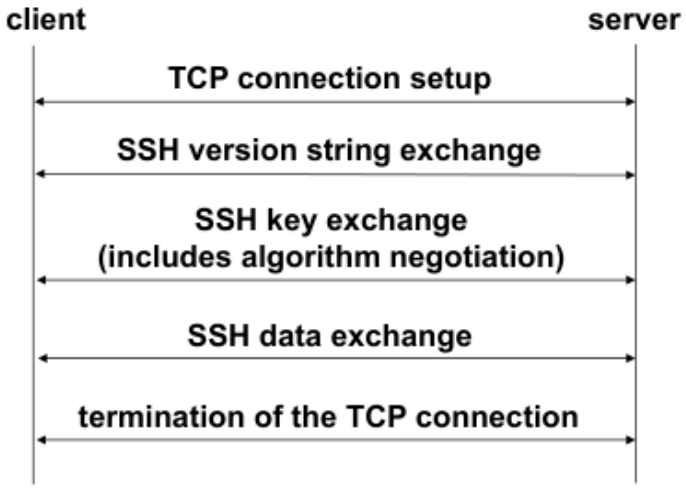


Figure 2.1: SSH Transport Layer Protocol

Binary Packet Protocol

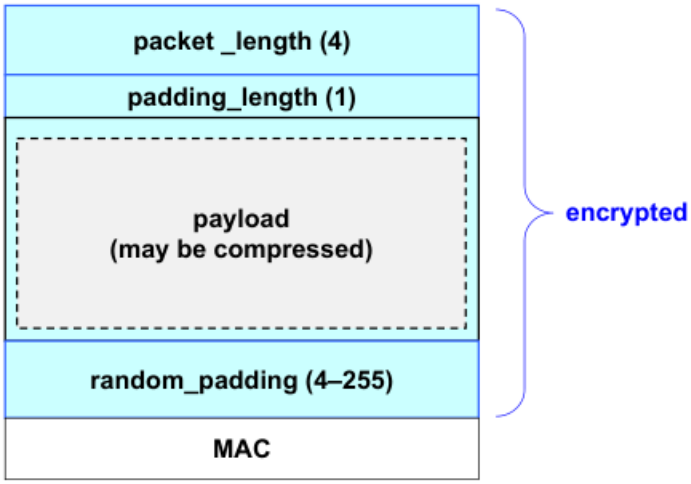


Figure 2.2: SSH Binary Packet Protocol

The SSH Binary Packet Protocol (BPP) is structured as follows:
Packet Length

- The packet length does **not** include the MAC or the packet length field itself.

Payload

- The payload size is calculated as:

$$\text{payload\_size} = \text{packet\_length} - \text{padding\_length} - 1$$

- The maximum uncompressed payload size is **32,768 bytes**.
- The payload might be compressed.

Random Padding

- Padding can range from **4 to 255 bytes**.
- The total packet length (excluding the MAC) must be a multiple of `max(8, cipher_block_size)` even when a stream cipher is used.

MAC (Message Authentication Code)

- The MAC is computed as part of the authenticate-and-encrypt schema.
- It is computed over the cleartext packet and an implicit sequence number.
- Since the MAC requires the packet to be decrypted before checking its integrity, this process could be exploited for a **denial of service (DoS)** attack.

TLP key exchange

algorithm negotiation:

- SSH_MSG_KEXINIT
- cookie (16 random bytes)
- kex_algorithms
- server_host_key_algorithms
- encryption_algorithms_client_to_server, ... _server_to_client
- mac_algorithms_client_to_server, ... _server_to_client
- compression_algorithms_client_to_server, ... _server_to_client
- languages_client_to_server, ... _server_to_client
- first_kex_packet_follows (flag)
  - attempt to guess agreed kex_algorithm

Algorithm Specification

- kex_algorithms (note: contains HASH)
  - e.g. diffie-hellman-group1-sha1, ecdh-sha2-OID_of_curve
- server_host_key_algorithms
  - e.g. ssh-rsa
- encryption_algorithms_X_to_Y
  - e.g. aes-128-cbc, aes-256-ctr, aead_aes_128_gcm
- mac_algorithms_X_to_Y
  - e.g. hmac-sha1, hmac-sha2-256, aead_aes_128_gcm
- compression_algorithms_X_to_Y
  - e.g. none, zlib
- complete list maintained by IANA: https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml

SSH: Diffie-Hellman (DH) Key Agreement

The Diffie-Hellman key agreement process in SSH follows these steps:

- **[C] (Client)** generates a random number x and computes:

$$e = g^x \mod p$$

- **[C ¿ S]**: Client sends e to the server.
- **[S] (Server)** generates a random number y and computes:

$$f = g^y \mod p$$

- **[S]** computes the shared secret:

$$K = e^y \mod p = g^{xy} \mod p$$

  The server then computes the exchange hash:

$$H = \text{HASH}(c\_version\_string \,|\, s\_version\_string \,|\, c\_kex\_init\_msg \,|\, s\_kex\_init\_msg \,|\, s\_host\_PK \,|\, e \,|\, f \,|\, K)$$

- **[S]** generates a signature sigH on H using its private key s_host_SK. This may involve an additional hash computation on H.
- **[S ¿ C]**: Server sends its public key s_host_PK, f, and sigH to the client.
- **[C]** verifies that s_host_PK is indeed the server's public key.
- **[C]** computes the shared secret:

$$K = f^x \mod p = g^{xy} \mod p$$

  The client then computes the exchange hash:

$$H = \text{HASH}(\ldots)$$

- **[C]** verifies the signature sigH on H.
- **[C, S]**: The exchange hash H becomes the session ID.

2.2.2 SSH: Key Derivation

In SSH, keys are derived using the shared secret K, the exchange hash H, and the `session_id`. The derivation process is as follows:

Initial IV (Initialization Vector)

- **Client to Server:**

$$\text{IV}_{\text{C to S}} = \text{HASH}(K \,||\, H \,||\, "A" \,||\, \text{session\_id})$$

- **Server to Client:**

$$\text{IV}_{\text{S to C}} = \text{HASH}(K \,||\, H \,||\, "B" \,||\, \text{session\_id})$$

Encryption Key

- **Client to Server:**

$$\text{Key}_{\text{C to S}} = \text{HASH}(K \,||\, H \,||\, "C" \,||\, \text{session\_id})$$

- **Server to Client:**

$$\text{Key}_{\text{S to C}} = \text{HASH}(K \,||\, H \,||\, "D" \,||\, \text{session\_id})$$

Integrity Key

- **Client to Server:**

$$\mathrm{MAC}_{\mathrm{C\ to\ S}} = \mathrm{HASH}(K \,||\, H \,||\, "E" \,||\, \mathrm{session\_id})$$

- **Server to Client:**

$$\mathrm{MAC}_{\mathrm{S\ to\ C}} = \mathrm{HASH}(K \,||\, H \,||\, "F" \,||\, \mathrm{session\_id})$$

Note: The `session_id` is the H value computed during the first key exchange and remains unchanged even when key re-exchange is performed.

2.3 Encryption

The used algorithms is negotiated during the key exchange phase (also key and iv are established during in this phase), and can be diffrnt in each direction.
A first example of **Supported Algorithms:**

- **Required** (for backward compatibility):
  - 3des-cbc (with three keys, i.e., 168-bit key)

  [itemsep=0pt]
- **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

Note: all packets sent in one direction is a single data stream, so IV is passed from the end of one packet to the beginning of the next one

2.3.1 Encryption: Handling of IV

In the firsty packet, the IV is randomly generated during KEX. After that, depends is it us CBC or CTR:

- **CBC:** IV for next packet is the last encrypted block of the previous packet
- **CTR:** IV for next packet is the IV of the pervious packet incremeted by one

2.4 MAC

The used algorithms is negotiated during the key exchange phase and can be diffrnt in each direction.
Supported Algorithms (Basic Set):

- **hmac-sha1 (required)** [key length = 160-bit] (backward compatibility)
- **hmac-sha1-96 (recommended)** [key length = 160-bit]
- **hmac-md5 (optional)** [key length = 128-bit]
- **hmac-md5-96 (optional)** [key length = 128-bit]

Even if these are the value inidicate in the RFC, for security we always need to use algorithms in the **SHA-2 family**.
The MAC is computed in the following way (**MAC = mac(key, seq_number — cleartext_packet)**):

1. sequence number is implicit, not sent with the packet
2. sequence number is represented on 4 bytes
3. seq_number initially 0 and incremented after each packet
4. seq_number never reset (even if keys/algos are renegotiated)

2.5 SSH: Peer Authentication

2.5.1 Server

The server authetication is done by an symmetric challenge-response (explicit server signature of the key exchange hash H).
So the client need to locally store the server public keys (not good). Typically are saved in `~/.ssh/known_hosts` and, is a key is absenty then it's offered at first connection with a TOFU (Trust On First Use) policy, **Very Danger**.
For good practice we need to protect `known_hosts` for **authentication and integrity** and **periodic audit/review** all the `known_hosts` files to quickly detect added/deleted hosts or changed keys

2.5.2 Client

Can be done in two ways:

- **Pusername and password:** the client sends the password to the server only after the protected channel is created. Usefull for avoid sniffing, but open to other attacks (e.g. on-line password enumeration)
- **asymmetric challenge-response:** server locally stores the public keys of the users allowed to connect as a local user (typically in `~/.ssh/authorized_keys`).

Some good practices are the protection of authorized_keys for authentication and integrity and the periodic audit/review of all the authorized_keys files to quickly detect added/deleted users or changed keys.