ADVANCED INFORMATION SYSTEM SECURITY

Hoping to get a better grade this time around.

Fabio Lorenzato

November 4, 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.

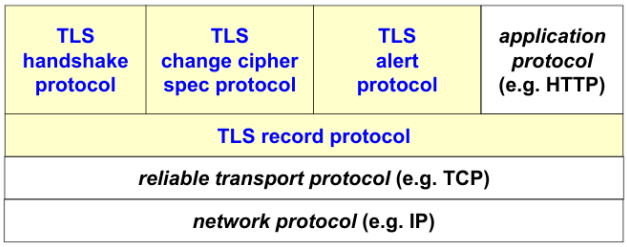


Figure 1.1: TLS packet structure.

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).

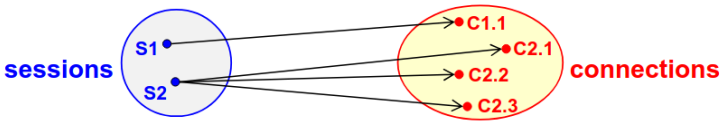


Figure 1.2: TLS session and connection.

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)

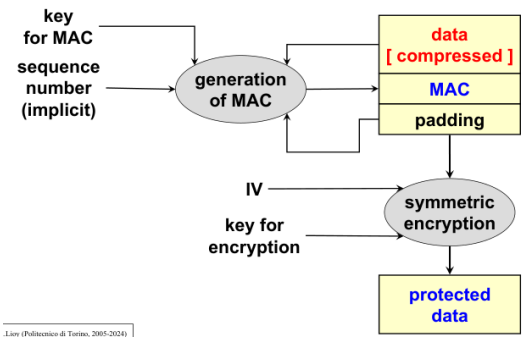


Figure 1.3: TLS data protection.

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.

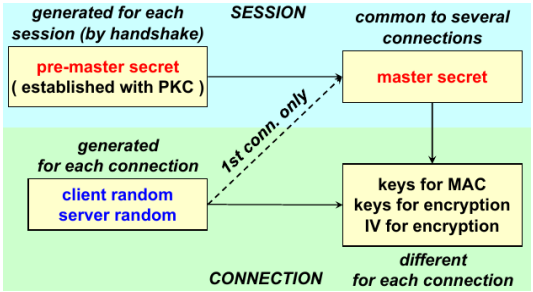


Figure 1.4: Relationship among keys and sessions.

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.

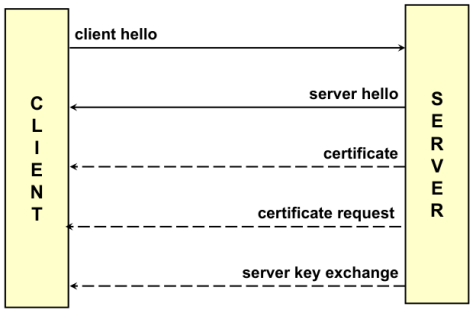
1.6.1 Client Hello and Server Hello

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

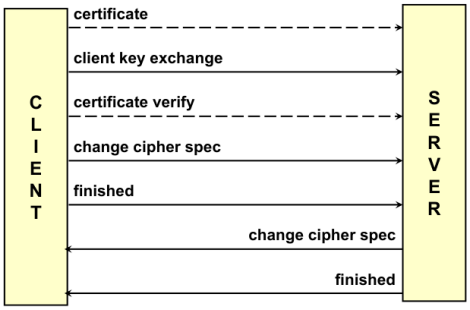
- the SSL version preferred by the client, and the highest supported(2=SSL-2, 3.0=SSL-3, 3.1=TLS-1.0, ...)
- a 28 bytes pseudo-random number, which is the client random
- a session-id, which is *0* if the client is starting a new session, and *1* if the client is trying to resume a previous session
- 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.

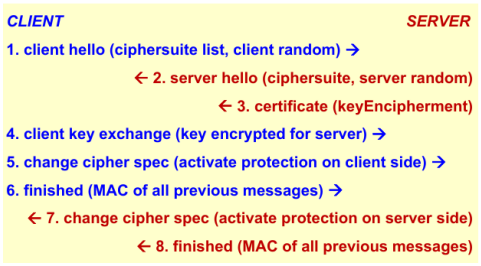
1.6.6 Change cipher spec

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.

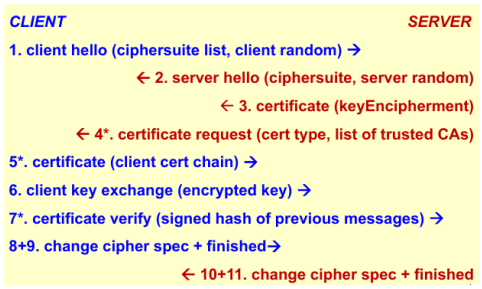
1.6.7 Finished message

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.

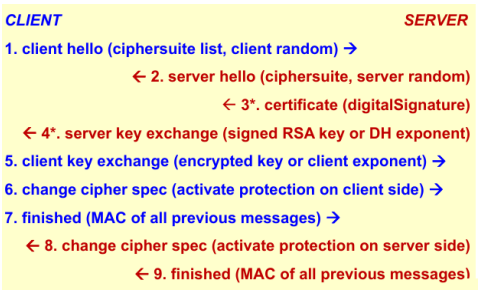
This allows to prevent rollback man-in-the-middle attacks (version downgrade or ciphersuite downgrade)



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



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



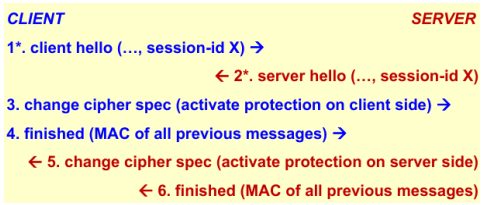
(c) TLS handshake(ephemeral, no client authN).

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).

(a) TLS link teardown.



(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.

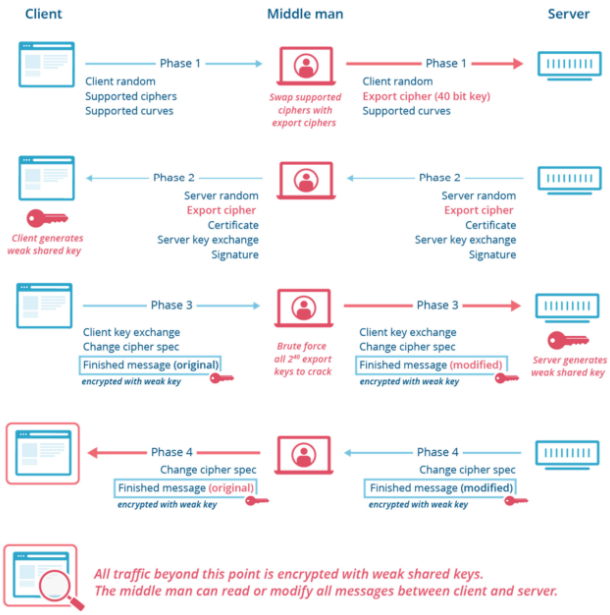


Figure 1.8: FREAK attack.

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).

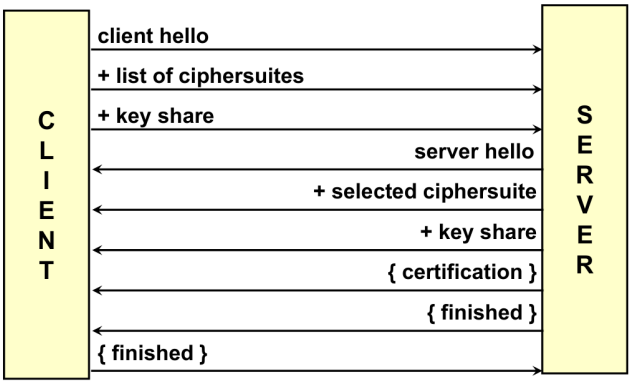


Figure 1.9: TLS 1.3 handshake.

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, which makes it a *"competitor"* of the much more used TLS protocol. It has been introduced in 1995 by Tatu Ylönen, a Finnish computer science student, as a response to a security incident per which the FTP credentials of his university were sniffed, but it has been commercialized since so it was not open source anymore. A open source fork called OpenSSH has been introduced in OpenBSD in 1999, and has been standardized in 2006 by the IETF in RFC 4251, referred to it as SSH-2.

> As it is the most used version, here we will always refer to SSH-2 (unless otherwise noted)

2.1 Architecture

From an architectural point of view, SSH is a three layer architecture, which means it is simpler than TLS. SSH uses TCP as the Transport Layer Protocol, but the SSH transport layer provides a nice set of features, which are **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, notice that this does not refer specifically to SSH bu to any encrypted channel) which is crucial for long running connections.

> Beware that the SSH transport layer is not the same as the TCP transport layer, but it is a layer that is built on top of the TCP transport layer, even though they have similar names.

There's also a specific protocol for **user authentication**, compulsory with SSH, 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 act as a site-to-site VPN.

2.1.1 SSH Transport Layer Protocol

The general schema of an SSH connection is shown in figure 2.1.
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 by both sides, which contains the security features supported. Both sides must send a **version string** of the following form(the pipe symbol is used to separate the fields):

```
SSH-protoversion-softwareversion|SP|comments|CR|LF
```

SP is a space character, CR is a carriage return character and LF is a line feed character. If the comments are omitted, the SP is also omitted.
Its also used to triggers compatibility extensions and to advertise the protocol implementation capabilities.
A **key exchange** follows, which includes algorithm negotiation for the key exchange itself. After this point data can be exchanged, and the connection is closed when the data exchange is finished.
The termination of the TCP connection is not part of the protocol itself, but it's performed by TCP itself.

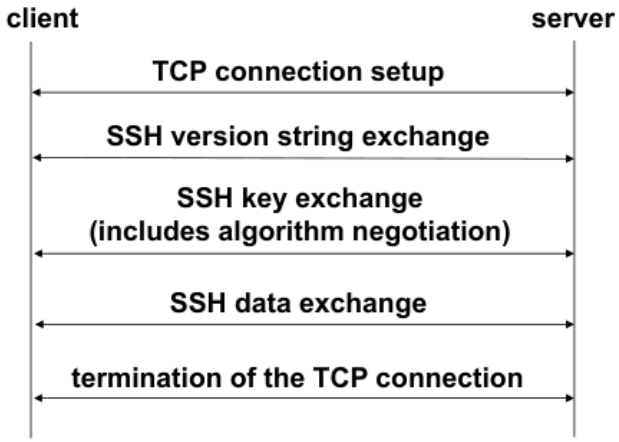


Figure 2.1: SSH Transport Layer Protocol

Binary Packet Protocol

All packets that follow the version string exchange is sent using the Binary Packet Protocol, which is roughly the basic unit on transmission. Every message in this kind of packet is sent as a sequence of bytes. 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 **padding length** (1 byte), which is the length of the random padding field
- 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)

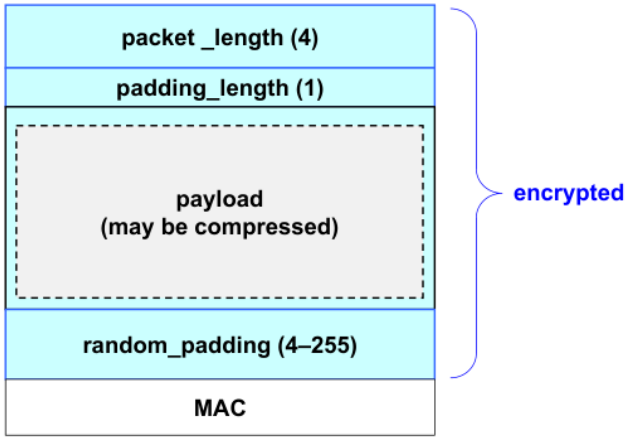


Figure 2.2: Binary Packet Protocol

- the **random padding**, minimum 4 bytes, up to the entire size of one block, and random to add confusion to the packet
- the **MAC**, computed over the clear text packet and the implicit sequence number

The first 4 fields are encrypted.

Key exchange

Another essential part of the SSH protocol is the key exchange, which is used to established the keys and to negotiate the algorithms to been used.
One peculiarity of the SSH protocol is that the algorithms used are selected based on directionality, which means that it is possible to use different algorithms for the client-to-server and server-to-client communication.
SSH-2 allows to select the key exchange function as well as the hash algorithm to be used in the key derivation function.

> This means that the two parties have *"full"* control over the algorithms to be used(if supported).

Another interesting aspect of the protocol is that, as it was proposed as an alternative to the Telnet protocol, it included negotiation of the language to be used, because of the text-only nature of the protocol.
The last field of the key exchange(first key exchange packet follows) contains an attempt to guess the agreed key exchange algorithm.
It contains the following fields:

- SSH_MSG_KEXINIT
- cookie (16 random bytes)
- kex_algorithms
  - eg:diffie-hellman-group1-sha1, ecdh-sha2-OID_of_curve

- server_host_key_algorithms: used for the server authentication
  - eg:ssh-rsa, ssh-dss, ecdsa-sha2-NISTP256
- encryption_algorithms_client_to_server, ... server_to_client
  - aes-128-cbc, aes-256-ctr, aead_aes_128_gcm
- mac_algorithms_client_to_server, ... server_to_client
  - hmac-sha1, hmac-sha2-256, aead_aes_128_gcm
- compression_algorithms_client_to_server, ... server_to_client
  - none, zlib
- languages_client_to_server, ... server_to_client
- first_kex_packet_follows (flag)

Keep in mind that because of the negotiation of the protocols to be used, SSH is potentially vulnerable to downgrade attacks.

> A complete list of the supported algorithms can be found **here**

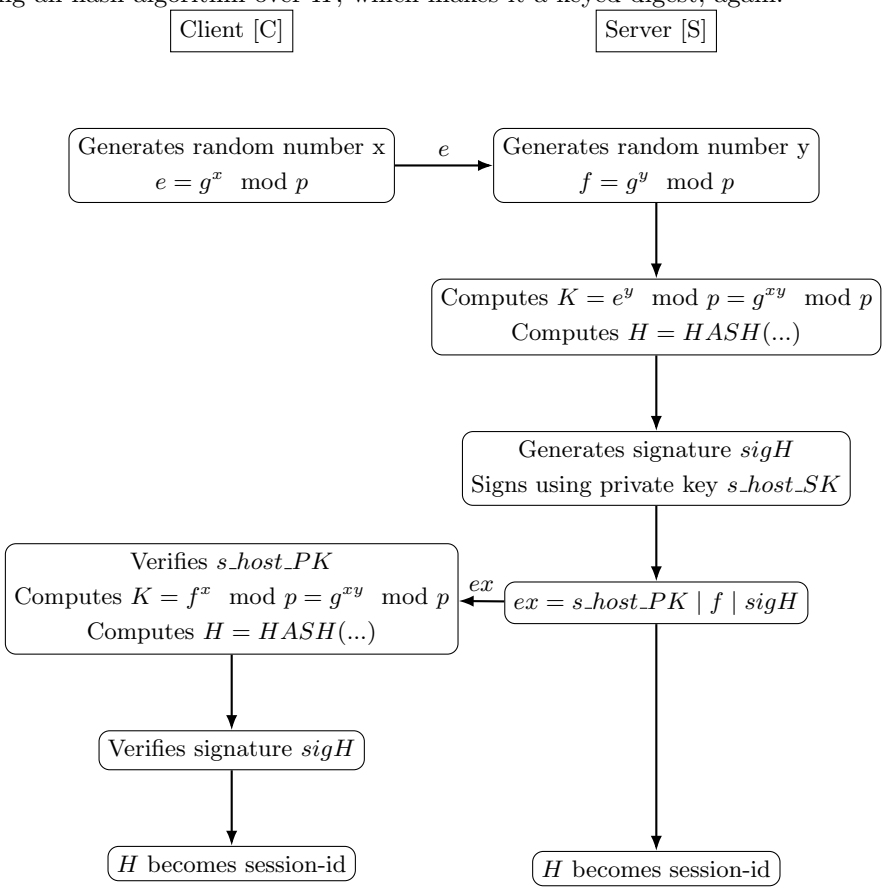
Diffie-Hellman key agreement

This is also an **implicit authentication** of the server, due to the fact that the server can compute the correct premaster secret using the private key.
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. At this point the key K has been established. The server computes the **exchange hash** H over many fields(which makes it basically a keyed digest):

- the client version string
- the server version string
- the client's KEXINIT message
- the server's KEXINIT message
- the shared secret K
- the exchange value f
- the exchange value e
- the host public key

and signs it with its private key to generate the signature $sigH$.
Those two values, together with the host public key in a raw format, are sent to the client, which verifies the signature and computes the same hash value. Notice that because the public key is sent in a raw format, the client must have a copy of the server's public key stored locally, and this is one of the major attack vectors of SSH.

Notice also that, because many algorithms need an initialization vector, which is computed using an hash algorithm over H, which makes it a keyed digest, again.



Key derivation

Many keyed digest require a initialization vector.
Starting with those exchanged values, if there is any kind of algorithm that requires an IV (nearly all), then the initial IV is:

- From client to server: HASH($K \parallel H \parallel$ "A" $\parallel$ session_id), where K is the key that has been negotiated and "A" is simply a letter.
- From server to client: HASH($K \parallel H \parallel$ "B" $\parallel$ session_id).

The encryption key is generated as follows:

- From client to server: HASH($K \parallel H \parallel$ "C" $\parallel$ session_id).
- From server to client: HASH($K \parallel H \parallel$ "D" $\parallel$ session_id).

The integrity key:

- From client to server: HASH($K \parallel H \parallel$ "E" $\parallel$ session_id).
- From server to client: HASH($K \parallel H \parallel$ "F" $\parallel$ session_id).

The HASH algorithm is the one used to create all these keys, and this could be a weak point because it would be much better to generate keys using a KDF (Key Derivation Function).

2.1.2 Encryption

Encryption is compulsory in SSH, which algorithms are negotiated during the key exchange. The supported algorithms are:

- (required for backward compatibility) 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

The key and initialization vector needed are established during the key exchange and all packets sent in one direction is a single data stream, meaning that the first packet sent has got an initialization vector, which is the one that was created with the key derivation function. But in the following ones, the IV is passed in the packet from the end of one to the beginning of the next one. In fact, there are two cases:

- if **CBC mode** is used, the IV for next packet is the last encrypted block of the previous packet
- if **CTR mode** is used, the IV for the next packet is the incremented value of the last IV

2.1.3 MAC

The MAC algorithm and the key are negotiated during the key exchange and they can be different in each direction.
The basic set of algorithms are:

- hmac-sha1 (required for backward compatibility) [key length = 160-bit]
- hmac-sha1-96 (recomm) [key length = 160-bit]
- hmac-md5 (opt) [key length = 128-bit]
- hmac-md5-96 (opt) [key length = 128-bit]

> The mac is computed using the key for the right direction over the sequence number concatenated with the packet in clear text.

The sequence number is implicit, as in TLS, and its possible because of TCP. It is represented as a 32-bit unsigned integer, which is incremented by one for each packet sent.
This value is never reset, even if keys and algorithms are re-negotiated, which means that when the maximum value is reached, the channel has to be reset.

2.1.4 Peer authentication

The **server** is authenticated using an **asymmetric challenge-response** mechanism, which requires and explicit server signature of the key exchange hash H. (The challenge is implicit, the signature is implicit).
The client locally stores the public keys of the server, which are usually stored in the `known_hosts` file in the `.ssh` directory.
When connecting to a server not listed in that file, the client is asked to store the public key of the server in the file, following a **TOFU** (Trust On First Use) policy.
Because of this, a good practice is to protect the `known_hosts` file for authentication and integrity, and to periodic audit/review all the known_hosts files to quickly detect added/deleted hosts or changed keys.
Client authentication can be performed by two methods. The first one is based on **credentials**(username and password), which are exchanged only after the protected channel is established. This method protects against sniffing attacks, but not other ones like password guessing.
The second method is based on **asymmetric challenge-response**. In that case, the server must locally store the public keys of the users allowed to connect, typically in the `authorized_keys` file in the `.ssh` directory.
As per the server authentication process, as a good practice, the `authorized_keys` file should be protected for authentication and integrity, and periodically audited/reviewed to quickly detect added/deleted keys or changed keys.

2.2 Port forwarding

We already mentioned that SSH can act as a site-to-site VPN, but rather than creating a general purpose VPN, SSH can be used to perform **port forwarding** or **tunneling**.

> Port forwarding is a technique that allows to forward unprotected TCP traffic through a secure SSH channel.

This allows to secure unprotected service like POP3, SMTP or HTTP while also allowing the application to perform their normal authentication over an encrypted channel.
There are two types of port forwarding, **local** and **remote**, but both use the Connection Protocol to encapsulate a TCP channel inside a SSH one.

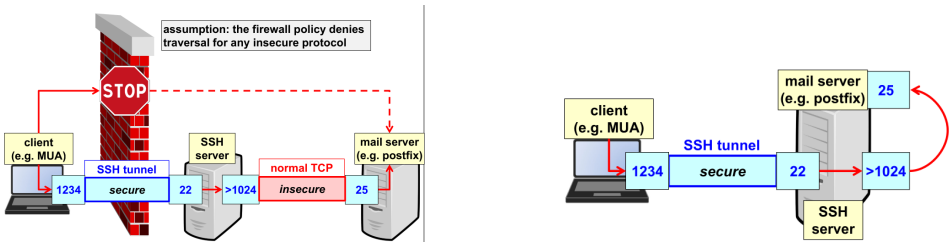
2.2.1 Local port forwarding

The concert of local port forwarding is to forward a local port to a remote one, which means that the client listens on a local port and forwards the traffic to a remote port.
For example, let's suppose to be behind a firewall that blocks access to an external mail server because only secure traffic is permitted. An SSH tunnel can be created to forward the local port 110 to the remote port 25 using the following command:

```
ssh -L 1234:mail_server:25 user@ssh_server
```

the traffic to port 1234 on the (internal) client will be forwarded to port 25 on the mail_server

by using a tunnel to the (external) ssh_server.
The firewall will see the traffic as SSH traffic, but the user has still to configure the mail user agent to use the local port 1234 to connect as the outgoing mail server.

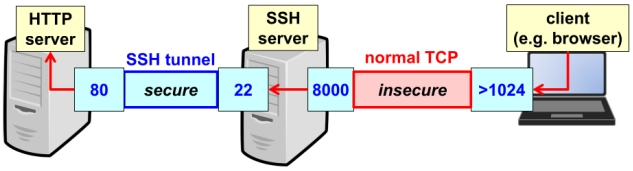


2.2.2 Remote port forwarding

Remote port forwarding is the opposite of local port forwarding, which means that the client listens on a remote port and forwards the traffic to a local port.
For example, let's suppose to have a web server on a local machine that is behind a firewall, and the user wants to make it available to the public internet. The user can create a tunnel to the remote machine with the following command:

```
ssh -R 8000:localhost:80 user@ssh_server
```

The traffic to port 8000 on the remote machine will be forwarded to port 80 on the local machine.
The user can now access the web server on the local machine by connecting to the remote machine on port 8000.



2.2.3 Causes of insecurity

In general, the biggest causes of insecurity for SSH are direct trust in the public keys. In fact, x.509 certificates are not used, but some commercial SSH implementations support them and openSSH has SSH certificates, which are a different thing.
Furthermore, blindly trusting the public keys can lead to a MITM attack. The server could also have a weak platform security(worms, malicious software, rootkits, etc.) as well as the client (malware, keyloggers, etc.).
One last point is that any connection to a local forwarded port will be tunneled, even if coming from another node. This is undesirable, as such it is recommended to specify the binding address when creating a tunnel:

```
ssh -L 127.0.0.1:1234:mail_server:25 user@ssh_server
```

2.3 Attacks on SSH

2.3.1 BothanSpy

BothanSpy is believed to be a CIA tool, according to information released by Wikileaks on July 6, 2017, as part of their Vault 7 disclosure. This malware targets Xshell, a Windows-based SSH client widely used in the USA and South Korea. BothanSpy operates by injecting a malicious DLL into the Xshell process, enabling it to steal sensitive information from SSH connections. Specifically, it collects:

- Usernames and passwords from password-authenticated connections.
- Usernames, private key filenames, and passphrases from public-key authenticated connections.

BothanSpy is designed to work with the ShellTerm attack framework, featuring:

- A covert communication channel with a Command and Control (C&C) server.
- DLL injection capabilities for direct C&C communication, with no data written to disk, making it difficult to detect using traditional anti-malware solutions.
- An offline mode, where the collected data is written to disk, encrypted with AES for later retrieval.

2.3.2 Gyrfalcon

Gyrfalcon is another suspected CIA tool, as revealed by Wikileaks on July 6, 2017, in their Vault 7 release. This malware specifically targets OpenSSH on enterprise Linux systems, such as RedHat and CentOS. Gyrfalcon operates by pre-loading a malicious DLL to intercept plaintext traffic, capturing information both before encryption and after decryption. The tool is capable of capturing:

- Usernames and passwords.
- Actual data transmitted over SSH connections.

Key features of Gyrfalcon include:

- An encrypted configuration file.
- An encrypted file for storing captured data.

While Gyrfalcon requires root access for installation, it notably does not integrate with a Command and Control (C&C) server. Instead, it freely reads and writes files on disk, potentially exploiting the limited use of anti-malware solutions on Linux platforms. Despite these capabilities, Gyrfalcon is not particularly stealthy or sophisticated compared to other tools.

2.3.3 Brute force attack

Brute force attacks can exploit the false sense of security provided by secure communication channels, especially when insecure authentication methods like reusable passwords are employed. In a typical brute-force attack, an attacker systematically attempts to guess a password by trying all possibilities from a dictionary of commonly used passwords.

An illustrative example of such an attack occurred in September 2016:

- Two servers, one with an IPv4 address and the other with an IPv6 address, were activated in the cloud to test SSH brute-force attacks.
- The IPv4 server was immediately targeted and compromised within just 12 minutes, primarily because the root password was set to "password."
- Once compromised, the server was quickly used in a Distributed Denial of Service (DDoS) attack.
- In contrast, after one week, the IPv6 server remained unscathed, as it had not even been attacked.

This example demonstrates the risks of relying on weak passwords and highlights the disparity in attack frequency between IPv4 and IPv6 networks.

Protection from brute-force attacks

To mitigate the risk of SSH brute-force attacks on Linux systems, several strategies can be implemented:

- **Account Lockout**: Automatically lock accounts after a certain number of failed authentication attempts using tools like `pam_tally2` or `pam_faillock`.
- **TCP Wrappers**: Use TCP wrappers to restrict SSH access by permitting or denying connections from specific hosts or networks.
- **SSH Rate Control**: Implement rate limiting for SSH connections using IPtables, for example, by allowing only 5 connection attempts per minute.
- **Disable Root Access**: Deny direct SSH access to the root account to limit exposure.
- **Change Default SSH Port**: Move SSH from the default port 22 to another port to reduce the chances of automated scans and attacks.
- **Fail2ban**: Use `Fail2ban` to monitor log files and blacklist IP addresses after a defined number of authentication failures.
- **Two-Factor Authentication (2FA)**: Implement 2FA, such as Google Authenticator, for an added layer of security.
- **Disable Password-Based Authentication**: Disable password-based authentication altogether in favor of more secure methods like public-key authentication.

These measures can significantly reduce the effectiveness of brute-force attacks and improve the overall security of SSH services.

2.4 Main Applications of SSH

SSH (Secure Shell) is widely used for a variety of secure applications due to the encryption and authentication mechanisms it offers. The main applications include:

- **Remote Interactive Access**: SSH allows users to securely connect to remote systems with a text-based interface, enabling command-line interaction.
- **Remote Command Execution**: SSH facilitates the secure execution of commands on remote systems, making it ideal for system administration and automation tasks.
- **Tunneling**: SSH can create secure tunnels for various applications, such as forwarding network traffic, encrypting data, or creating virtual private networks (VPNs).

Keep in mind that SSH is now natively available on Linux, macOS, and Windows, providing both client and server functionality on these platforms.
All of these functions benefit from the security properties inherent to SSH, including confidentiality, integrity, and authentication.

Chapter 3

The X.509 standard and PKI

We will now discuss the X.509 standard and Public Key Infrastructure (PKI), which is widelly adopted for public key certification.

3.1 Public-key certificate (PKC)

Lets begin with a definition:

> A **public-key certificate** is a **data structure** to securely bind a public key to some attributes.

A PKC is said to "securely bind" a public key to some attributes because it is signed by a trusted entity, the **Certification Authority**, but other techniques for certification exists (e.g. blockchain, direct trust and personal signature).
The attributes in the certificate are those employed in the transaction being protected by the PKC, which are difficult to decide a-priori without knowing the context: for example one attribute could be an email address associated with the entity, which may not be valid for all the transactions.
PKC are most important to achieve **non-repudiation**, many certificates have legal value, and **digital signature** in a secure way.

> Keep in mind that a PKC is the complement of a corresponding **personal private key**, which is kept secret by the entity.

Which means that if the private key is compromised, the PKC is compromised as well.

3.1.1 Key Generation in Public Key Cryptography (PKC)

Key generation in public key cryptography (PKC) involves complex algorithms and often relies on random number generators (RNGs) to ensure strong, unpredictable keys. Once a key is generated, the private key must be securely protected in various contexts:

- **Storage**: The private key needs to be securely stored to prevent unauthorized access.

- **Usage**: The private key must also be protected when being used, as it could be exposed during operations in the CPU as it is used in clear text.
- **Software Application**: In software environments (e.g., web browsers), the trustworthiness of the computing platform must be considered, as it may be vulnerable to malware or weak implementations. This is true both for the context of key generation and use.
- **Dedicated Hardware**: Storing and using keys in dedicated hardware, such as a smart card, offers enhanced protection but comes with limitations in terms of algorithm updates or vulnerability patching(which may be difficult or even impossible). For example many old smart cards use 1024-bit RSA keys, which are considered insecure today.
- **Key Injection**: Keys may be generated in software and injected into hardware devices, a process that can be useful for key recovery, but should be restricted to encryption keys to maintain security and ensure non repudiation. This is most important for the private key.

PKC key management requires careful consideration of both security and operational challenges, especially when dealing with long-term security mechanisms.

3.1.2 Certification architecture

In Public Key Infrastructure (PKI), several key entities are responsible for managing the lifecycle of Public Key Certificates (PKCs):

- **Certification Authority (CA)**: The CA is responsible for generating and revoking PKCs. It also publishes PKCs and maintains information about their status, such as whether they are valid or revoked, or even suspended.
- **Registration Authority (RA)**: The RA plays a critical role in verifying the claimed identity and attributes of a certificate requestor. It authorizes the issuance or revocation of PKCs but does not generate them itself.
- **Validation Authority (VA)**: The VA provides services that allow third parties to verify the validity status of a PKC, because some responsibilities or the RA may be delegated to it. This may involve downloading Certificate Revocation Lists (CRLs) or querying an Online Certificate Status Protocol (OCSP) responder to check the certificate's status in real-time.
- **Revocation Authority**: Although not an official term, this role may be assigned to either the RA or CA. The revocation authority is delegated to revoke certificates, often on a more urgent basis than their issuance (they are available most of the time), ensuring that compromised or invalid certificates are promptly rendered unusable.

These entities work together to ensure the security and trustworthiness of PKC-based systems by handling key tasks such as certificate issuance, verification, and revocation.

3.1.3 Certificate generation

The general When an actor want to generate a certificate, it must first generate a key pair (public and private key). The private key is stored locally and protected(encrypted) as good as possible, while the public key is sent to the CA with some attributes that help identify the actor.
The CA requires that the attributes are both correct and are associated actually with the entity that is in control of the private key, which requires authentication of the actor. For that purpose, if the entity is a human it is usually required to go physically to the CA, even if in some cases a video call may be enough, with the requirement of showing a valid ID.
The Registration Authority will verify the attributes and the identity of the actor, and if everything is correct, it will send a message to the CA to generate the certificate.
At this point the CA generates the certificate, signs it with its private key and sends it back to the actor while also storing it in its repository.
As you may noticed, the verification is only on the attributes, and not of the possession of the private key, which will be discussed later.

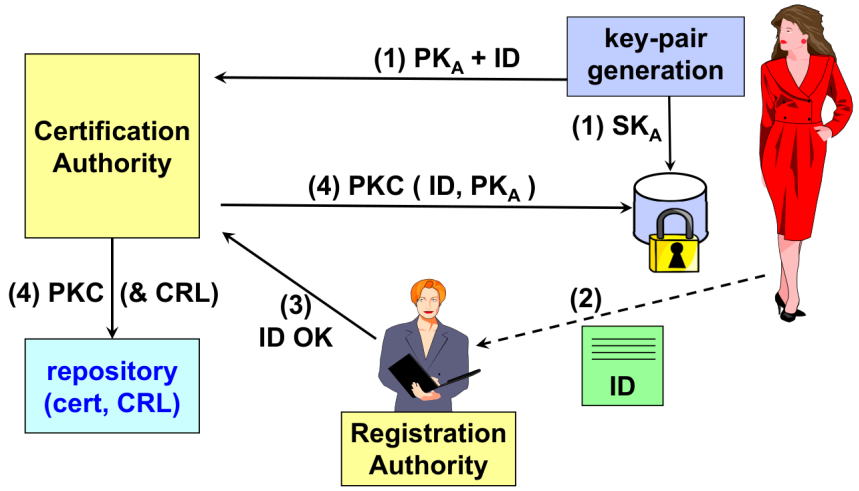


Figure 3.1: Certificate generation

Another certificate generation architecture

In addition to what just discussed, alternative architectures for certificate generation exist:

- **Key Pair Generation by RA**: In some architectures, the Registration Authority (RA) generates the key pair on behalf of the user, obtains the Public Key Certificate (PKC), and securely distributes the key pair and certificate to the user via a secure device, such as a smart card. This approach is often used in large companies where the employees are already known to the organization.
- **User Authentication via Code**: Another common method involves the user visiting the RA to obtain a code that can be used to authenticate their certificate request to the Certification Authority (CA). The code is typically calculated as a Message Authentication Code (MAC) using the shared secret key between the RA and CA:

$$\text{code} = \text{MAC}(K, ID)$$

where K is a shared secret key between the RA and CA, and ID is the user's identity. The user then submits this code to the CA to validate the certificate request.

These alternative methods provide flexibility for organizations with different security needs, allowing for centralized key generation and secure certificate distribution.

3.2 X.509 certificates

X.509 is an ITU-T (international telecommunication unit body) standard that defines the format of public key certificates used to verify the identity of a key owner in cryptographic systems. X509 was developed as a certification technology to protect the x500 directory service, and has undergone several versions:

- **v1 (1988)**: The initial version of the standard.
- **v2 (1993)**: A minor update with small improvements.
- **v3 (1996)**: Added extensions and introduced version 1 of the attribute certificate.
- **v3 (2001)**: Further enhancements, including version 2 of the attribute certificate, which are not used anymore.

X.509 is part of the larger X.500 standard for directory services, often referred to as "white pages," which are used for managing information about entities in a structured way(directory services). X.509 provides a solution to the problem of securely identifying the owner of a cryptographic key.

The certificates and their structures are defined using **ASN.1** (Abstract Syntax Notation One), a standard interface for defining data structures exchanged in networking environments in a neutral and platform-independent way.

3.2.1 PKC Scope

A Public Key Certificate (PKC) contains information that uniquely associates a cryptographic key with an entity. This binding between the key and the entity is ensured by a **Trusted Third Party (TTP)**, typically referred to as a Certification Authority (CA), which digitally signs each certificate to guarantee its authenticity.

The liability associated with a PKC may be restricted to specific applications or purposes, as outlined in the CA's certification policy. This policy defines the intended usage and limits the scope of the certificate, ensuring it is applied within the proper legal and technical contexts.

3.2.2 Certificate Policy (CP) and Certification Practice Statement (CPS)

According to RFC-3647, "Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework," both the Certificate Policy (CP) and Certification Practice Statement (CPS) play key roles in defining the use and management of Public Key Certificates (PKCs):

- **Certificate Policy (CP)**: A CP is a named set of rules that defines the applicability of a PKC to a specific community or class of applications with common security requirements. It establishes the minimum requirements for the issuance and management of certificates and can be followed by multiple Certification Authorities (CAs). For example, a government CP may apply to all certification providers issuing certificates for official use.
- **Certification Practice Statement (CPS)**: A CPS outlines the specific practices that a CA follows when issuing PKCs. While a CP specifies the general rules, the CPS provides detailed implementation procedures. Each CA develops its own CPS, which is tailored to its operations and describes how it meets the requirements set forth in the CP.

3.2.3 X.500 Directory Service

The X.500 directory service was the first system to use X.509v1 certificates. It aimed to manage information about people and entities in a network. However, this early setup had three big issues:

- **No Guarantees on the CA's Quality**: There weren't clear rules or policies to ensure that Certification Authorities (CAs) were reliable or trustworthy. People just had to trust the CA, which caused concern over the security of the certificates.
- **Lack of a proper X.500 Infrastructure**: Even though certificates were supposed to be stored in X.500 directories, the infrastructure to do this was never fully implemented worldwide. This made it tough to access certificates when needed.
- **Hard to Establish Certification Paths**: It was difficult to connect two users with certificates issued by different CAs because the relationships between CAs weren't well-defined. With each CA running its own domain, figuring out how to establish a secure connection between users from different domains was tricky.

These issues slowed down the adoption of X.500 and made it clear that better systems were needed for managing certificates.

Fixing X.509v1 Problems

To solve these problems, X.509v1 was updated with a couple of key changes:

- **Move the semantics Outside the Certificate**: Instead of relying on the certificate itself to define its semantics, they decided to put that responsibility on the application

or some external system. This approach was used in things like RFC-1422 (PEM), where the application takes care of interpreting the certificate.

- **Make Certificates More Flexible**: The original X.509v1 certificates were pretty limited: they only had an identifier and a key. With X.509v3, they added more fields and options to make certificates more flexible and useful for a variety of purposes. This allowed them to handle a wider range of security scenarios.

These updates helped fix the early issues and set the stage for broader adoption of the improved X.509v3 certificates.

3.2.4 RFC-1422

RFC-1422 tried to fix some of the problems with the early X.500 systems by proposing a worldwide certification hierarchy. The plan was to have one main CA at the top: the **IPRA** (Internet Policy Registration Authority). This CA would be responsible for setting the rules and policies for all other CAs in the system, ensuring that everyone followed the same security practices.
Instead of directly issuing certificates to lower-level CAs, the IPRA would issue certificates to **Policy Certification Authorities (PCAs)**.
The IPRA oversaw the entire structure and laid out specific roles for different types of Certification Authorities (CAs) to manage and enforce certificate policies:

- **Policy Certification Authority (PCA)**: PCAs were responsible for setting and enforcing the policies that governed how certificates were issued. They made sure that the CAs under them followed the right security practices.
- **Name Subordination**: CAs had to issue certificates within a defined part of the naming structure, ensuring a consistent and organized system. For example:
  - **CA n.1**: C=IT (country-level CA for Italy)
  - **CA n.2**: C=IT, O=Politecnico di Torino (CA for an organization within Italy)

However, this system wasn't without flaws—it was still based on geographic hierarchies, which didn't always match the needs of modern, global organizations.

The IPRA was established, and four PCAs were created under it. One of these PCAs was designated as **high assurance**, meaning it had to carry out stricter checks before issuing certificates. For instance, BBN, a company heavily involved with the U.S. military, might require things like fingerprints or even DNA tests to verify identities.

In contrast, **mid-level assurance** certificates, like those issued by universities such as MIT, required less rigorous checks—maybe just showing a passport. MIT could even set up its own subordinate CAs, like one for the Laboratory for Computer Science, to handle its internal certificate needs.

What's interesting is how different cultural practices shaped the certification process. In the U.S., where people don't generally carry ID cards, **residential certification authorities** were used. If you wanted to open a bank account, for example, you might have to show an electricity bill to prove your address. While this seems strange in countries with formal ID systems, in the U.S., it was a necessity.

Finally, there were **persona certificates**, which allowed for anonymous certificates. These were for users who didn't want to reveal their identity but still needed secure communications. For example, you could be "anonymous user number 95." Even though no one knew who you were, the security of your communication was still guaranteed.

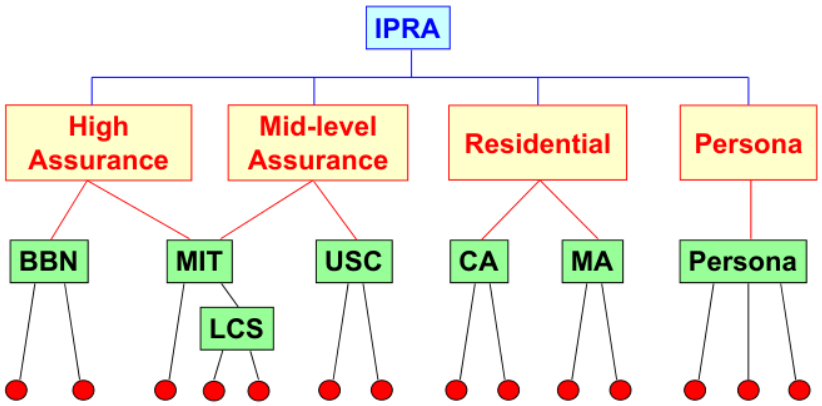


Figure 3.2: Internet PEM hierarchy (RFC-1422)

Unfortunately, this system was a complete failure for several reasons:

- The hierarchical structure severely limited flexibility, much like the issues we saw with X.500. For international companies, this rigid hierarchy simply didn't work.
- Name subordination imposed additional restrictions. You couldn't just assign any name—you had to follow the strict hierarchy, which wasn't always practical.
- The use of the PCA concept lacked flexibility, especially in commercial settings. Instead of automating the process, a human operator was needed to check if the requester was following the policy. This made the system cumbersome and inefficient for businesses.
- Lastly, the biggest issue was trust. Where would the IPRA be placed? Naturally, the U.S. wanted it, but other countries, like Russia, China, or even Japan, wanted control as well. No one trusted a single country to sit at the top of the hierarchy, fearing the possibility of a fake hierarchy being created. As a result, the experiment collapsed. It briefly existed, primarily used by the United States, but it never gained global traction.

3.3 X.509 Version 3

X.509 Version 3 is a standard that was finalized in June 1996 through a collaborative effort between ISO, ITU, and the IETF, with the goal of making certificates suitable for internet applications. This version consolidated all the necessary updates to certificates and Certificate Revocation Lists (CRLs) into a single document. Earlier versions of X.509 (v1 and v2) relied on external semantics, such as the failed attempt with the Internet PEM hierarchy, which proved unsuccessful. X.509v3 addressed these shortcomings by ensuring certificates would be useful for internet applications, particularly as OSI applications became

mostly obsolete.
Key features of X.509 Version 3 include:

- **Types of Extensions**:
  - **Public Extensions**: These are defined by the standard and made publicly available for anyone to use. Because they are standardized, any system or application should be able to understand and process them.
  - **Private Extensions**: These are tailored to specific user communities and are not publicly available. If a system does not understand a private extension, it will treat it as a binary blob and discard it. Private extensions are crucial for certain organizations that require custom functionality.

  The introduction of extensions is a major improvement in X.509v3, adding flexibility without changing the fundamental structure of the certificate (which remains the same as in X.509v1). Extensions are contained within a small field but open up a wide range of possibilities. Public extensions ensure standard compatibility, while private extensions allow customization specific to organizations.

- **Certificate Profile**: This refers to a set of extensions tailored for a specific purpose or application, ensuring that certificates meet particular requirements. For example, RFC 5280, titled "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," provides guidelines on how X.509 certificates and CRLs should be structured for protecting internet applications. Certificate profiles are important because they dictate how certificates should be used for different scenarios, ensuring consistency and security.

3.3.1 Base syntax

The base syntax of an X.509 certificate is defined as follows:

```
Certificate ::= SEQUENCE {
    signatureAlgorithm AlgorithmIdentifier,
    tbsCertificate TBSCertificate,
    signatureValue BIT STRING
}

TBSCertificate ::= SEQUENCE {
    version [0] Version DEFAULT v1,
    serialNumber CertificateSerialNumber,
    signature AlgorithmIdentifier,
    issuer Name,
    validity Validity,
    subject Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    issuerUniqueID [1] IMPLICIT UniqueIdentifier OPTIONAL,
        -- if present, version must be v2 or v3
```

```
    subjectUniqueID [2] IMPLICIT UniqueIdentifier OPTIONAL,
        -- if present, version must be v2 or v3
    extensions [3] Extensions OPTIONAL
        -- if present, version must be v3
}
```

This abstract notation outlines the basic structure of the certificate. The `Certificate` is defined as a sequence, which is a data structure that includes several fields. The first field, `signatureAlgorithm`, is the algorithm identifier used by the Certificate Authority (CA) to sign the certificate.

Next, the `tbsCertificate` (to be signed certificate) is of type `TBSCertificate`, which contains the main information that needs to be signed. Finally, the `signatureValue` is a bit string that holds the actual signature.

The `TBSCertificate` is again defined as a sequence, starting with the `version` field, which begins numbering with zero, indicating version 1. For version 3, this field has a value of two. Following that, we have the `serialNumber` of the certificate and the `signature` algorithm identifier.

You may wonder why the signature algorithm appears twice—once externally and once within the `TBSCertificate`. The external algorithm identifier is crucial for verifying the signature; without knowing the algorithm, you cannot validate the signature value. Having it specified internally helps ensure that the signature algorithm used for verification matches the algorithm used for signing. If they differ, it's a red flag, indicating that the certificate should not be trusted.

The `issuer` field identifies the certification authority by name, and the `validity` field specifies the validity period of the certificate. The `subject` is identified by its name, and the `subjectPublicKeyInfo` field contains both the algorithm and the actual public key of the subject.

In addition, versions 2 and 3 introduce optional unique identifiers for both the issuer and subject, though these identifiers are rarely used today. Finally, the `extensions` field, which is optional, must be present for certificates of version 3, as earlier versions do not support extensions.

3.3.2 Critical Extensions

In the context of X.509 certificates, extensions can be classified as critical or non-critical, each affecting the certificate verification process differently:

- **Critical Extensions**: If a certificate contains an unrecognized critical extension, it **MUST** be rejected during the verification process. This ensures that any essential information required for proper validation is recognized and handled appropriately.
- **Non-Critical Extensions**: Conversely, a non-critical extension **MAY** be ignored if it is unrecognized. This allows flexibility in certificate processing, as non-critical information does not impede the overall verification of the certificate.

The responsibility for handling these extensions lies entirely with the entity performing the verification, referred to as the **Relying Party (RP)**. The RP must implement logic

to correctly interpret and respond to both critical and non-critical extensions in accordance with their definitions.

3.3.3 Public Extensions

X.509 version 3 defines four classes of public extensions that enhance the functionality and applicability of certificates:

- **Key and Policy Information**: Extensions in this class provide additional information regarding the key usage and policies applicable to the certificate, guiding how the key should be used within specific contexts.
- **Certificate Subject and Certificate Issuer Attributes**: These extensions include attributes related to both the subject of the certificate (the entity that the certificate represents) and the issuer (the Certification Authority), enabling more detailed identification and classification.
- **Certificate Path Constraints**: These extensions define rules and limitations regarding the certification path, ensuring that certificates can only be used in certain contexts or under specific conditions, enhancing security within the certificate hierarchy.
- **CRL Distribution Points**: This extension indicates where the Certificate Revocation List (CRL) can be found, providing necessary information for relying parties to check the revocation status of the certificate efficiently.

3.3.4 Key and policy information

There are six public extensions dedicated to this purpose. We won't discuss all of them, but we will focus on the most important ones.

Authority Key Identifier

Key and policy information extensions in X.509 v3 provide crucial details about the public keys associated with certificates. One significant component is the **Authority Key Identifier (AKI)**, which serves the following purposes:

- **Identification of the Signing Key**: The AKI identifies a specific public key used to sign a certificate, ensuring that the verification process can accurately trace the certificate's authenticity back to the correct authority.
- **Identification Methods**: The identification can be achieved through:
  - A **key identifier**, typically represented as the digest (hash) of the public key of the certification authority.
  - The combination of **issuer-name** and **serial-number**, allowing a clear reference to the issuing CA's key.
- **Usage**: The AKI is particularly useful in scenarios where the same CA might utilize multiple keys, such as for different assurance levels, like low assurance for general use and high assurance for sensitive applications.

- **Non-Critical Extension**: While the AKI is classified as non-critical, its presence can be vital in certain applications. In the professor experience, the absence of this extension can lead to many applications rejecting the certificate as invalid.
- **Importance in Certificate Chains**: When an application receives a certificate, it often needs to establish a trust path up to a root CA. This path-building process relies heavily on the AKI field. If a CA neglects to include this extension, it may lead to compatibility issues with various applications, despite being non-critical.

Subject key identifier

The **Subject Key Identifier** (SKI) extension serves to identify a specific public key associated with a certificate. Key features include:

- **Identification of Public Key**: The SKI uniquely identifies a particular public key used in an application. This is especially important in scenarios where the public key may be updated or replaced over time.
- **Being Non-Critical**: The SKI is classified as a non-critical extension, meaning that while it provides valuable identification information, its absence does not necessarily prevent the certificate from being considered valid.

This one is usually not present in the certificate because an identifier is already present in the certificate.

Key usage

The **Key Usage** (KU) extension specifies the application domains in which a public key may be utilized. Key characteristics include:

- **Application Domain Identification**: The KU extension identifies the specific purposes for which the associated public key can be employed, ensuring that it is used appropriately within defined contexts. In case of misuse, the application may refuse to accept the certificate and the CA may refuse liability.
- **Critical or Non-Critical**: The Key Usage extension can be classified as either critical or non-critical:
  - If marked as **critical**, the certificate may only be used for the specific purposes indicated in the Key Usage extension. Any usage outside the defined scopes would render the certificate invalid.
  - If marked as **non-critical**, the certificate can be used more flexibly, potentially allowing usage beyond the defined applications without invalidating the certificate.
- **Permitted Cryptographic Operations**: The following cryptographic operations can be defined within the Key Usage extension:
  - **digitalSignature**: valid in the certificate of a user but also in the certificate of a certification authority.

  - **nonRepudiation**: Specifically permitted for users (non repudiation is only for humans after all).
  - **keyEncipherment**: Permitted for users.
  - **dataEncipherment**: Allowing encryption of data.
  - **keyAgreement**: Involves:
    * **encipherOnly**: Restricting use to enciphering operations.
    * **decipherOnly**: Restricting use to deciphering operations.
  - **keyCertSign**: Specifically permitted for Certificate Authorities (CAs).
  - **cRLSign**: Also permitted for Certificate Authorities (CAs) for signing Certificate Revocation Lists (CRLs).

Private key usage period

The **Private Key Usage Period** extension in X.509 v3 specifies the time frame during which the associated private key may be used. Key features include:

- **Usage Period Definition**: This extension defines the period during which the private key can be actively utilized, helping to enforce time-based restrictions on key usage.
- **Non-Critical Extension**: The Private Key Usage Period extension is always classified as non-critical, meaning that its absence does not invalidate the certificate. However, the information it provides can be important for managing key lifecycles.
- **Usage Discouraged**: While this extension is available, its use is generally discouraged in practice. Organizations often prefer to manage key lifetimes through other means, such as regular key rotation, rather than relying on a specified usage period within the certificate.

Certificate policies

The **Certificate Policies** extension outlines the specific policies that were adhered to during the issuance of the certificate and defines the purposes for which the certificate can be utilized. Key aspects include:

- **Policy Listing**: This extension provides a comprehensive list of the policies that govern the certificate issuance process, ensuring clarity regarding its intended use.
- **Indication Methods**: Certificate policies can be indicated using various formats, including:
  - **Object Identifier (OID)**: A unique identifier for the policy.
  - **Uniform Resource Identifier (URI)**: A link to a location where the policy can be reviewed.
  - **Text Message**: A textual description of the policy.
- **Critical or Non-Critical**: The Certificate Policies extension can be classified as either critical or non-critical:

  - If marked as **critical**, the certificate must only be used in accordance with the specified policies; otherwise, it may be considered invalid.
  - If marked as **non-critical**, the certificate can be used more flexibly, although the policies still provide guidance for its intended use.

- **Support for Authentication and Authorization**: The use of this extension can enhance not only the authentication of users and entities but also facilitate authorization processes, providing a clearer understanding of the permissible actions associated with the certificate.

Policy mappings

The **Policy Mappings** extension in X.509 v3 establishes a correspondence between policies across different certification domains. Key aspects include:

- **Mapping of Policies**: This extension indicates how policies from one certification authority (CA) correspond to policies from another, facilitating interoperability and trust among different certificate frameworks. This can be done automatically, but needs human interaction to be done correctly.

- **Presence in CA Certificates**: The Policy Mappings extension is typically present only in CA certificates, allowing certification authorities to define and communicate relationships between their policies and those of other authorities.

- **Non-Critical Extension**: The Policy Mappings extension is classified as non-critical, meaning its absence does not invalidate the certificate. However, it serves as an important tool for enhancing the understanding and usability of certificates across different domains.

3.3.5 Certificate subject and issuer attributes

As previously mentioned, the X.509 v3 standard includes extensions that provide additional information about the subject and issuer of a certificate. Without this extension, the only possible names are x500 ones, which aren't really an identifier.

Subject Alternative Name (SAN)

The Subject Alternative Name (SAN) extension provides a flexible way to identify the owner of a certificate. This extension allows the use of various formalisms to represent the owner, including e-mail addresses, IP addresses, and URLs.
Importantly, the SAN field is always considered critical when the subject name field is empty. This means that if there is no subject name, the SAN extension must be present to ensure proper identification and functionality of the certificate.

Issuer Alternative Name (IAN)

The Issuer Alternative Name (IAN) extension in enables the use of various formalisms to identify the Certificate Authority (CA) that issued a certificate or a Certificate Revocation

List (CRL). This extension supports different identifiers such as e-mail addresses, IP addresses, and URLs.
The IAN field is always considered critical when the issuer name field is empty. In such cases, the IAN extension must be present to ensure proper identification of the issuing authority.

It's also interesting to see what are the possible identifiers that can be used in the IAN field.
One option is the **RFC 802 Name**, which consists of the standard email addresses used in internet applications. The **DNS Name** identifies a node by its domain name, while an **IP Address** identifies the node by its numerical address. Another alternative is the **Uniform Resource Identifier (URI)**, which serves as an entry point on the web. This means that on the same node, you could have different certificates corresponding to various entry points for the applications.

In addition, you can use a **Directory Name**, which can be formatted according to X.500 or LDAP syntax. For example, the LDAP of the Politecnico di Torino might be represented as `country=T; organization=Politecnico di Torino; organizationUnit=Department`. While there is no formal management of country Italy in this case, the syntax remains consistent.

Moreover, **X.400 Addresses** illustrate that these identifiers were still in use when version 3 was developed in 1996, as there were gateways that transformed mail from X.400 to RFC 802 formats. Thus, having the certificate associated with the X.400 address was essential.

Next, we have the **EDI Party Name**, which stands for electronic data interchange. This standard allows companies to avoid manual processing of data. For instance, a car manufacturer might need to check if a supplier has specific tires available. Instead of making a phone call or sending a fax or email, they could use EDI to make a query through a neutral language that facilitates data requests and responses electronically.

EDI serves as a general standard with various implementations, such as EDIFACT, which Stellantis and many other car manufacturers use to manage the availability of necessary items, check prices, and place orders. The significance of EDI is reflected in the certificates associated with it.

Although EDI may seem older compared to modern standards like XML or JSON, many companies still rely on these established systems to avoid dealing with breaking changes that may occur.

In addition to the EDI Party Name, a **Registered ID**, such as a VAT number for a company or an Italian tax number for individuals, can also be used. This represents any official registered identifier for an entity. Another identifier is the **Unique Identifier**, which signifies a distinct identifier for individuals or organizations. Finally, there is the **Other Name** option. This should be used cautiously, as it lacks a well-defined structure. When using the "Other Name," the syntax and content definition fall to the user, making it non-standard and challenging to interpret.

Subject Directory Attributes

The Subject Directory Attributes extension allows for the storage of specific directory attributes related to the owner of the certificate. For example, the Department of Defense (DoD) utilizes this extension to indicate attributes like "citizenship."
The actual usage of Subject Directory Attributes heavily depends on the application since there are no standard definitions for these attributes. This extension is classified as non-critical, meaning its absence does not prevent the certificate from being valid but may limit its utility in certain contexts.

3.3.6 Certificate path constraints

Let's now go over the chain of trust and the constraints that can be created in the certificate path.
There are three main constraints that can be set in the certificate path.

Basic Constraints

> The correct term for user in the context of certificates is **end-entity** (EE).

Basic Constraints

The Basic Constraints extension indicates whether the subject of the certificate can act as a Certificate Authority (CA). The values for this extension are defined as follows:

- **BC=true**: This indicates that the subject is a CA.
- **BC=false**: This indicates that the subject is an End Entity (EE).
- Additionally, it is possible to define the maximum depth of the certification tree, but only if **BC=true**.
- This extension can be classified as either critical or non-critical, though it is suggested to always mark this extension as critical.

Name Constraints

The Name Constraints extension is applicable only in Certificate Authority (CA) certificates. This extension defines the space of names that can be certified by a CA and follows the same format as the Alternative Names.

Key points regarding Name Constraints include:

At least one specification must be provided between:

- **permittedSubtree**: This serves as a whitelist for names that are allowed.
- **excludedSubtree**: This serves as a blacklist for names that are not allowed.

The whitelist is processed first, ensuring that allowed names take precedence. Caution is advised: An unspecified format in the whitelist (e.g., `directoryName`) is implicitly permitted. This extension can be either critical or non-critical, though it is important to note that there is no non-critical support by Apple.

Policy Constraints

The Policy Constraints extension is used by a Certificate Authority (CA) to specify constraints that may require an explicit identification by a policy or that inhibit policy mapping for the remainder of the certification path.

This extension can be either critical or non-critical.

3.3.7 CRL distribution point

The CRL Distribution Point extension identifies the distribution point of the Certificate Revocation List (CRL) that is to be used in validating a certificate.

The distribution point can be specified in various forms, including:

- **Directory entry**
- **E-mail** or **URL**

This extension can be either critical or non-critical.

3.3.8 Private Extensions

The **Private Extensions** in X.509 v3 enable the creation of extensions tailored to specific user communities, allowing for customized applications within closed groups.

One key aspect of private extensions is their definition. These are custom-defined features intended for use by particular communities or groups of users. This capability provides both flexibility and specificity in the application of certificates, allowing them to meet the unique needs of different organizations or use cases.

Another important consideration is the examples from the Internet Engineering Task Force Public Key Infrastructure (IETF-PKIX), which has defined three notable private extensions for the Internet user community. The first of these is **Subject Information Access**, which provides essential details on how to access data related to the subject of the certificate. The second extension, known as **Authority Information Access**, offers guidance on how to access information pertaining to the certificate authority, ensuring that users can verify and trust the source of the certificate. Finally, the **CA Information Access** extension supplies vital information for accessing resources linked to the certificate authority, facilitating the management and verification of certificates within the relevant community.

Subject Information Access

The **Subject Information Access** extension in X.509 v3 specifies a method for obtaining additional information about the owner of a certificate, particularly useful when a directory service is not employed for certificate distribution. This extension allows users to retrieve extra details regarding the certified end entity, which can be important for verifying credentials or understanding the context of the certificate.

The extension typically includes a method, such as HTTP, HTTPS, or LDAP, which outlines the protocol to be used for accessing the required information. Alongside this method, it provides a name indicating the location or address where the information can be

obtained. For instance, it might direct users to a specific web page related to the individual or point to a local directory entry.

It is important to note that this extension is classified as non-critical, meaning it serves primarily as informational. While it enhances the utility of the certificate by providing additional access methods, it does not impact the fundamental validation of the certificate itself.

Authority Information Access

The **Authority Information Access (AIA)** extension plays a crucial role in X.509 certificates, as it specifies how to access the information and services offered by the certificate authority (CA) that issued the certificate. This extension is applicable to any certificate, whether from a certification authority or an end entity.

AIA contains several important subfields that facilitate various functions. One key subfield is **certStatus**, which may include a URL for an Online Certificate Status Protocol (OCSP) responder. This is particularly useful for checking the real-time status of a certificate, complementing the Certificate Revocation List (CRL) distribution point.

Additionally, AIA may provide pointers for:

- **certRetrieval**: This subfield allows for the retrieval of the certificate itself from a specified location.
- **cAPolicy**: It outlines the policies under which the CA operates, offering insight into the trustworthiness of the certificate.
- **caCerts**: This indicates where to find the CA's own certificates, which is essential for building a certification path.

The presence of AIA in a certificate enables users to locate the services provided by the CA. It acts as a guide for retrieving necessary information, such as following a path to an OCSP responder or accessing the CA's certificates. While the AIA extension can be classified as either critical or non-critical, it is typically recommended for scenarios that require real-time status checks or validation. This extension enhances the functionality of certificates by making it easier to access relevant information necessary for establishing trust.

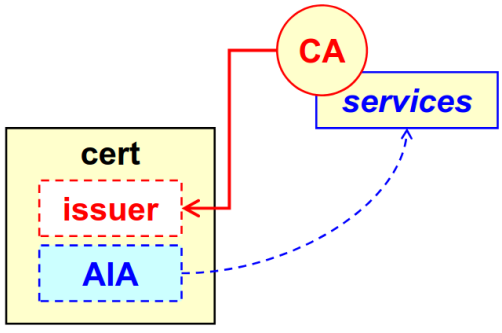


Figure 3.3: Authority Information Access (AIA) extension

CA Information Access

The **CA Information Access (CAIA)** extension serves a specific function in X.509 certificates, as it indicates how to access information and services provided by the certificate authority (CA) that owns the certificate. Unlike other extensions, CAIA is valid only within a CA certificate itself.

CAIA includes several important subfields that facilitate access to the CA's services, such as:

- **certStatus**: This subfield may point to the status of the CA's certificates, typically through an OCSP responder.
- **certRetrieval**: This allows for the retrieval of the CA's own certificate from a designated location.
- **cAPolicy**: This subfield provides information on the policies that govern the CA's operations, helping users understand the trust model.
- **caCerts**: It indicates where the CA's certificates can be accessed, which is vital for validating the certificate chain.

The significance of CAIA lies in its role as a self-pointer for the CA. When you possess a CA certificate and wish to locate its services, CAIA provides the necessary pointers to information about the CA itself, including its policy, OCSP responder, and certificate repository. This self-referential nature distinguishes CAIA from other access methods, which typically point back to the parent CA. As with many extensions, CAIA can be classified as either critical or non-critical, depending on the context in which it is used. However, it is particularly valuable for establishing a complete understanding of the CA's services and trustworthiness.

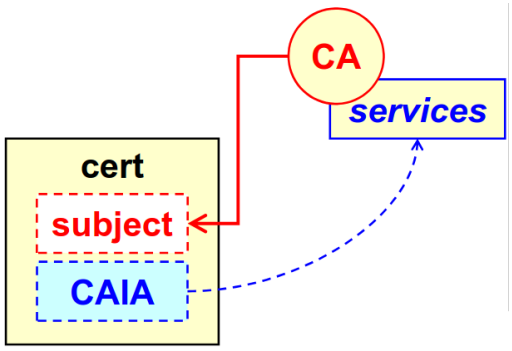


Figure 3.4: CA Information Access (CAIA) extension

RFC-2459

The **RFC-2459** document defines a profile for the use of X.509v3 certificates in Internet applications, such as IPsec, TLS, and S/MIME. This profile was initially suggested by the Public Key Infrastructure (PKIX) working group to promote interoperability and standardization across diverse systems and protocols.

One of the key aspects of RFC-2459 is that it establishes extensions defined by an Object Identifier (OID), specifically using the base `ID-PKIX ::= 1.3.6.1.5.5.7` (which maps to `iso.org.dod.inet.sec.mechanisms.pkix`). OIDs ensure that each extension and algorithm used within the X.509 framework is uniquely identifiable and properly managed. For instance, PKIX requested an OID rooted in the U.S. Department of Defense namespace due to the historical origin of the internet (as DARPA Net), and further sub-trees were established for protocols like IPsec, TLS, and S/MIME.

RFC-2459 also specifies the supported algorithms that should be implemented to ensure secure and reliable communication. Beyond defining supported algorithms, it offers several implementation suggestions to enhance compatibility. A few examples include:

- **UTC Time**: It is mandatory to include seconds in time fields.
- **Zulu Time**: UTC time must be expressed in the Zulu (Greenwich Mean Time) timezone, which avoids issues with software systems that misinterpret time zones. For instance, some software (notably from Microsoft) might default to local time, causing inconsistencies.
- **Two-Digit Year Format**: Although specifying the year in two digits is not recommended, when used, it should be interpreted within the range of 1950-2049. After 2049, certificates must switch to four-digit year formats to avoid ambiguity and incompatibility issues.

By adhering to the guidelines outlined in RFC-2459, certificate issuers and consumers ensure proper functionality across diverse systems, especially in the context of internet security protocols. The use of standardized extensions, supported algorithms, and implementation details like time format further strengthens interoperability and the reliability of X.509 certificates.

Extended Key Usage

The **Extended Key Usage** (EKU) extension in X.509 certificates allows for defining specific uses of a certificate in addition to, or in substitution of, the standard `keyUsage` extension. Unlike the standard key usage, which is focused on cryptographic operations, EKU is oriented towards specific applications. This enables a certificate to be more narrowly defined for its intended purpose, enhancing security and clarity in certificate handling.

EKU can be used simultaneously with the standard `keyUsage` extension, though it is important to consider how conflicts between the two might be handled in practice. EKU serves to tailor certificates for specific tasks within certain application domains, which is particularly useful in environments requiring strict certification boundaries, such as server authentication or email protection.

Some of the possible values for EKU include:

- `serverAuth` (`id-pkix.3.1`): This EKU designates a certificate for server authentication. In cases where serverAuth is used, the equivalent key usages that should be enabled include Digital Signature (DS), Key Encipherment (KE), and Key Agreement (KA).

- `clientAuth` (`id-pkix.3.2`): This EKU is intended for client-side authentication and can be mapped to Digital Signature (DS) and Key Agreement (KA).
- `codeSigning` (`id-pkix.3.3`): This EKU is used for code signing, allowing developers to sign the code they create. The primary associated key usage is Digital Signature (DS).
- `emailProtection` (`id-pkix.3.4`): This EKU provides a certificate for email protection. It includes Digital Signature (DS), Non-Repudiation (NR), Key Encipherment (KE), and Key Agreement (KA) as the relevant key usages.
- `timeStamping` (`id-pkix.3.8`): Used for timestamping, this EKU is associated with Digital Signature (DS) and Non-Repudiation (NR). Timestamping certificates are crucial for certifying when a particular action, such as a signature, took place.
- `ocspSigning` (`id-pkix.3.9`): This EKU is specific to signing OCSP responses. The associated key usages are Digital Signature (DS) and Non-Repudiation (NR), ensuring that OCSP responses are properly authenticated.

In summary, the PKIX group developed EKU to provide more granular control over certificate usage within specific applications. For example, a certificate intended for email may only be valid for email protection, while another for server authentication may only serve that purpose. The flexibility offered by EKU allows for more precise security policies, tailored to the needs of individual systems or applications.

Evolution of RFC-2459

The evolution of **RFC-2459**, which initially defined the profile of X.509v3 certificates for use in Internet applications, reflects a series of updates and refinements to both certificate profiles and cryptographic algorithms.

RFC-2459 was replaced by two key documents:

- **RFC-3280**: This document defines the Internet profile of public key infrastructures (PKIs) based on X.509v3 certificates and X.509v2 certificate revocation lists (CRLs). It was intended to standardize how certificates and CRLs are managed and used in Internet-based PKI systems.
- **RFC-3279**: This document covers the algorithms used in conjunction with the RFC-3280 profile, providing identifiers, parameters, and encodings. It includes, and makes obsolete, the earlier RFC-2528, which addressed the use of the Key Exchange Algorithm (KEA).

The decision to split RFC-2459 into two separate RFCs was driven by the recognition that while certificate profiles tend to remain stable over time, cryptographic algorithms evolve more rapidly. By documenting algorithms separately in RFC-3279, it became easier to update the list of supported algorithms without needing to revise the entire certificate profile.

Later on, **RFC-3280** was itself obsoleted by **RFC-5280**, which further refined the Internet PKI profile and introduced updates to align with advancements in cryptographic practices and PKI implementation. RFC-5280 remains a foundational document in the definition of PKI systems used for securing Internet communication.

RFC-3279 and Supported Algorithms

RFC-3279 specifies the algorithms that *MUST* be supported by applications using the **RFC-3280** profile. It includes some older algorithms, as well as newer ones. The key algorithms defined in this RFC are:

- **Digest algorithms:**
  - MD-2, MD-5, SHA-1 (preferred)
- **Algorithms for signing certificates/CRLs:**
  - RSA, DSA, ECDSA (Elliptic Curve DSA)
- **Subject public key algorithms (SubjectPublicKeyInfo):**
  - RSA, DSA, KEA(a variant of Diffie-Hellman), DH (Diffie-Hellman), ECDSA, ECDH (Elliptic Curve Diffie-Hellman)

The underlined algorithms were introduced in **RFC-3279** in comparison to **RFC-2459**. One important point is that RFC-3279 permits the use of legacy algorithms, though they are not recommended (e.g., SHA-1 is preferred for digest, and RSA/DSA for certificate signing, with ECDSA added for more modern use).

Once the basic definitions are set, optional algorithms can be defined. Notably:

- **RFC-4055:** Adds better specifications for signatures, including the Probabilistic Signature Scheme (PSS), Optimal Asymmetric Encryption Padding (OAEP), and SHA-2 for digest computation.
- **RFC-4491:** Addresses the needs of Russia by providing specifications for the *GOST* algorithm family, widely used in Russian encryption standards.
- **RFC-5480:** Introduces elliptic curve keys of various kinds.
- **RFC-5758:** Adds support for new algorithms in the *SHA-2* family, such as SHA-224 and SHA-256.
- **RFC-8692:** Uses SHAKE128 and SHAKE256 for signatures, both part of the *SHA-3* family of algorithms.

RFC-3280 / RFC-5280: PKI Profile

RFC-3280 and **RFC-5280** specify the profile that defines not only values and fields but also algorithms and procedures. These ensure that certificates are processed consistently across the world. Key aspects include:

- **Path validation algorithm:** Defines how to build or verify the certificate chain, starting from an end entity (EE) certificate, which is issued by a CA, up to a trusted root.
- **Certificate status verification:** Specifies details for verifying the status of a certificate, using methods such as the full CRL or Delta-CRL.

- **Extended Key Usage (EKU):** Adds the *OCSPSigning* extended key usage.
- **Certificate extensions:** Includes important extensions such as:
  - **Inhibit any-policy:** Prevents the use of any policy, ensuring that the CA must specify its own policies.
  - **Freshest CRL:** A pointer to a new type of CRL (Delta-CRL), which contains only the differences with the last full CRL.
  - **Subject information access:** Already discussed previously.
- **Freshest CRL for CRLs:** If there is a Delta-CRL pointer in a certificate or CRL, it is possible to access the differences with the last full CRL. The Freshest CRL is also known as the Delta CRL Distribution Point. It is always marked as non-critical because there are multiple ways to check certificate status (e.g., full CRL, Delta CRL, OCSP), and it is not feasible to make any single method mandatory.

Additional Points

- **Path validation focus:** Due to the sensitive nature of path validation, recent RFCs have paid much more attention to specifying the algorithms for this process, rather than just providing high-level descriptions as in earlier versions.
- **Support for internationalization:** With global use of certificates, it has become important to support different alphabets, such as those used in Japan, China, and India, in fields like URI, DNS names, and email addresses. **RFC-8398** adds support for internationalized email addresses and domain names.
- **Freshest CRL pointer:** A crucial feature is the *Freshest CRL* pointer, which allows a certificate or CRL to reference the latest Delta-CRL. This prevents the need to redownload a full CRL when only small updates are necessary.
- **No Delta of Delta-CRLs:** Delta CRLs cannot be created incrementally (i.e., you cannot generate a Delta of a Delta CRL). Each Delta CRL must reference the base CRL directly, making the process more straightforward.

3.4 Certificate Revocation

A certificate may be revoked before its natural expiration due to various reasons. The revocation can occur under the following circumstances:

- **Upon request of the certificate owner (subject):** This typically happens due to:
  - **Key compromise:** the subject knows that someone else has gained possession of their keys.
  - **Key loss**: the subject cannot locate the key, meaning that the certificate is no longer trustworthy.
- **Upon request of the certificate sponsor (organization):** some cases include:

  - This occurs when an employee leaves the organization, the company goes out of business, or a server is dismissed, among other reasons.
  - Any entity no longer in use should have its certificate revoked to prevent unauthorized usage.
  - This also applies if a company shuts down or transitions services, rendering previous certificates unnecessary or insecure.

- **Autonomously by the issuer**, for a few reasons:
  - **Fraud**: convinces a certification authority to issue a certificate based on false information. For example, a person could impersonate an executive or delegate to receive an unauthorized certificate.
  - **Error**: because mistakes may happen sometimes
  - In such cases, the issuer may revoke the certificate without needing approval from the original certificate owner or sponsor.

Once a certificate is revoked, it is marked as invalid, and any transaction using that certificate must consider it untrustworthy.

Certificate status MUST be checked by the entity that accepts the certificate to ensure that it is still valid and has not been revoked. This is the responsibility of the *relying party (RP)*.

> The RP must always verify the certificate's status to protect any transaction (e.g., a commercial order) relying on the certificate for security.

3.4.1 Mechanisms for checking certificate status

When verifying a certificate's status, it is important to:

1. Consider the entire chain of certificates up to a trusted root.
2. Check if any certificate in the chain has expired.
3. Ensure all certificates are valid.
4. If a certificate has not expired, a PKC (Public Key Certificate) is considered valid unless otherwise stated.

Two possible mechanisms exist to check the validity status:

- **CRL (Certificate Revocation List)**, which is a list of revoked certificates. You check the list manually to see if a certificate has been revoked. The CRL is signed by the issuer or a delegated revocation authority.
- **OCSP (Online Certificate Status Protocol)** provides the status of a specific certificate at the current time. It returns an answer whether the certificate is valid at the moment of the request. The response is typically signed by the OCSP server, which may raise trust concerns.

3.4.2 X.509 CRL

The **Certificate Revocation List (CRL)** is a mechanism used to track revoked certificates. It contains:

- A list of revoked certificates.
- The revocation date and reason, indicating when the certificate ceased to be valid. Those details are not present in the OCSP response.

CRLs are issued periodically and maintained by the certificate issuer (CA), even if no additional revocations have occurred, a CRL must be reissued to ensure its "freshness" and prevent replay attacks with outdated CRLs.

RLs are digitally signed by:

- The Certificate Authority (CA) that issued the certificates.
- A revocation authority delegated by the CA. When signed by the revocation authority, the CRL is referred to as an **indirect CRL (iCRL)**.

For instance, at Politecnico, a CRL is generated every 30 days to ensure the CRL remains current, even if no new revocations have taken place. This practice prevents ambiguity about whether a CRL is outdated or if no certificates have been revoked.

x.509 CRL version 2

```
CertificateList ::= SEQUENCE {
    tbsCertList          TBSCertList,
    signatureAlgorithm   AlgorithmIdentifier,
    signatureValue       BIT STRING
}

TBSCertList ::= SEQUENCE {
    version                 Version OPTIONAL,
                           -- if present, version must be v2
    signature               AlgorithmIdentifier,
    issuer                  Name,
    thisUpdate              Time,
    nextUpdate              Time OPTIONAL,
    revokedCertificates     SEQUENCE {
        userCertificate       CertificateSerialNumber,
        revocationDate        Time,
        crlEntryExtensions    Extensions OPTIONAL
    } OPTIONAL,
    crlExtensions           [0] Extensions OPTIONAL
}
```

The CRL structure consists of a sequence containing the list to be signed (`tbsCertList`), the algorithm, and the signature value. The version field determines whether it's version 1 (if absent) or version 2. The issuer can be the Certificate Authority (CA) or a revocation authority.

The `thisUpdate` field indicates the time of the latest update, while the optional `nextUpdate` field suggests when the next CRL will be created, serving as a promise to issue a new CRL by that date. If the next update has passed, a new CRL should be obtained.

The `revokedCertificates` field lists revoked certificates, providing the serial number, revocation date, and optional extensions. These extensions may apply to individual revoked certificates or to the entire CRL, and are optional in both cases.

Extensions of CRLv2

In CRLv2, there is an extension for each entry of the CRL or a global for the whole CRL.

- **crlEntryExtensions**:
  - *Reason Code*: Helps clarify the cause of revocation (e.g., key compromise vs. key destruction).
  - *Hold Instruction Code*: Marks a certificate as temporarily suspended, not permanently revoked. Though deprecated, because it forces the validators to keep a lot of copies just for the suspended certificates.
  - *Invalidity Date*: Specifies when the certificate became invalid.
  - *Certificate Issuer*: Identifies the issuer in case of an indirect CRL (i.e., the CRL is signed by a revocation authority rather than the CA).
- **crlExtensions**:
  - *Authority Key Identifier*: Supplemental information about the key used to sign the CRL.
  - *Issuer Alternative Name*: Provides additional issuer identification, supporting internet-based identifiers.
  - *CRL Number*: Tracks different versions of the CRL.
  - *Delta CRL Indicator*: Specifies if the CRL is a delta CRL (shows only differences from the base CRL). If it's 0 it's not a delta CRL.
  - *Issuing Distribution Point*: A pointer to the location where the latest CRL can be found.

Certificate revocation timeline

A key issue arises with the *revocation date*(CRLv2) and *invalidity date*(CRLv2 Extention), which are both included for revoked certificates. Although they might seem redundant, they serve to solve a problem.
Take a look at figure **??**, we have a point in time (before the red part) in which everithing works fine and the CLR number n has been issued. At a certain point the private key has been compromised, meaning that someone can impersonate someone else. A certificate

revocation in requested after the user became aware of the issue, which would like to revoke the certificate with a prior date or time of the one that would be issued by the CA. Now the two dates come into play. The *revocation date* is the date certified by the CA, while the *invalidity date* is the date claimed by the ee.
The risk is not yet over, because the CA, to issue a new certificate needs some time(the one in yellow), meaning that every request made in that time slot will still get the old certificate back. For this reason any good relying party should wait to get the next issued CRL for any important operation, while also having a strong proof of the time when the key was used.

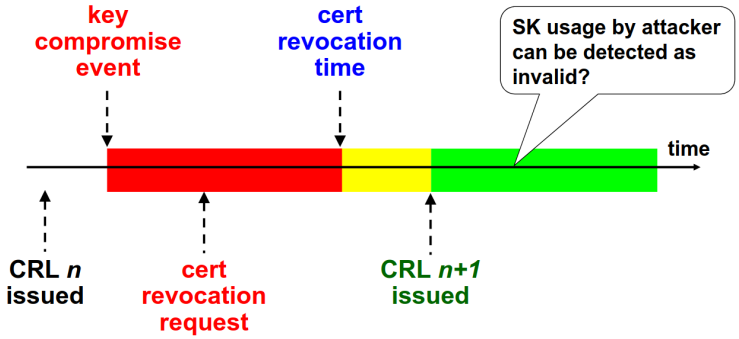


Figure 3.5: Certificate revocation timeline

Efficient management of CRLs

The major issue with CRLs is their size. They can become quite large over time, especially in large organizations. For this reason many solution have been proposed.
One of those is to eliminate the revocation following the first CRL issued after the expiry date of the certificate, but required an archive in which all the past CRLs are stored.
Another solution is to use the *Delta CRL*, by publishing the base CRL and, from that one on, only the changes. This solution is good for the download time, but still doesn't solve the problem of the storage.
The best solution is probably the **partitioning of the CRLs**. For example, one CA could partiotion them in buch of 1000 certificates each, and to download one you need to download the whole partition.
In the end, a perfect solution does not exist.

3.4.3 OCSP

OCSP (as defined in RFC-6960) is an alternative to CRLs and is used to verify if a certificate is **valid** at the **current moment**. The protocol follows a client-server model where the server, known as the responder, returns one of three possible responses: *good* (the certificate is valid), *revoked* (the certificate has been revoked), or *unknown* (the status of the certificate cannot be determined).

Responses from the OCSP responder are digitally signed to prevent fake responses. Optionally, the responder may accept only signed requests from clients to provide authentication and restrict access.

> Keep in mind that, because the OCSP response is signed with the certificate of the OCSP server, it is not possible to verify the signature using OCSP itself.

As a result, OCSP responders usually operate with short-lived certificates, typically valid for 24 hours, requiring automation to manage frequent certificate requests.

It is implemented as a binary protocol and does not have a default port. The URI, provided in the Authority Information Access, specifies the port on which the OCSP responder is available. OCSP can be encapsulated in various transport protocols such as HTTP, HTTPS, LDAP, and SMTP, which act as wrappers for the binary data exchange.

OCSP provides a real-time mechanism for checking the validity of certificates, but managing the responder's certificates, especially with short lifetimes, often requires automation to ensure continuous availability and security.

OCSP source of information

OSCP, to provide an answer, must have a source of information. This data can be aquired in different ways:

- **download i form CRL repositorie**
- **querying the CA database**
- **ask another OSCP server(chaining)**

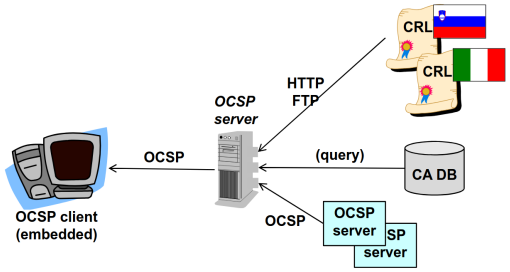


Figure 3.6: OCSP source of information

The protocol

Lets now go over how the protocol works.

An OCSP **request** contains three parameters:

- the **protocol version**
- the **target certificate identifiers**
- the **extension**, which are optionals

In the request, each certificate is identified by its certID:

- hashAlgorithm (AlgorithmIdentifier)
- issuerNameHash (hash of Issuer's DN)

- issuerKeyHash (hash of Issuer's public key)
- serialNumber (CertificateSerialNumber)

As you can see, in this section, no explicit information about the certificate is provided, and this is done for privacy reasons.

The response contains:

- the version of the response syntax.
- the name of the responder itself.
- one response for each certificate in the request, allowing the status of multiple certificates to be requested with one query.
- the extension, optional again
- The OID of the signature algorithm for the signed response.
- The signature, which is computed across the hash of the response.

The SingleResponse of each certificate contains:

- The certificate identifier
- the certificate status, and if revoked, the revocation time and optionally, also the revocation reason.
- the thisUpdate field, which indicates the time of the response to indicate which is the knowledge of the responder.
- the nextUpdate field, which is optional

Models of OCSP responder

There are some types of models:

- **CA Responder** (operated by the CA): the CA signs the response with its own private key, which is quite risky because the private key of the CA must be online, so it is rarely adopted. It is possible to mitigate this problem by using a key dedicated only to OCSP signing (remember ExtendedKeyUsage). Even if the OCSP responder is operated by the CA, it can use different keys: one for creating certificates and one to sign them. That goes also in the direction of "Authority Key Identifier" because a CA could have 3 keys: certificate sign, CRL sign, OCSP responder sign. This because they are different functions and maybe different machines. This last method is the most used one.
- **Trusted responder**: the OCSP server signs the responses with a pair key:cert independent of the CA for which it is responding. The company responder is something like that. It can be also a trusted third party (TTP) paid by the user.

- **Delegated responder**: the OCSP server signs the responses with a pair key:cert which is different based on the CA for which it is responding. It's a TTP paid by the CA for who it is responding. An external server is delegated to provide the answer on the CA behalf by providing a pair key:cert where the cert contains OSCP responder as extended key usage, and this will show that the responder is delegated by the CA to provide OSCP answers. There could be one responder that answers for multiple CAs.

As we have seen, OSCP is faster but for real time transactions they are quite the same. For delayed verifications we have problems because we need to store the information at the time of the signature.

Attacks against OCSP

There are two main attacks possible against OCSP.

The first one is obviously a **replay attack**: one can ask for the validity of a certificate, store the response and replay it when its not valid anymore by being faster then the OCSB server or stopping the real response via a denial of service. For this reason, optionally but encouraged, the client could send a **nonce**(`id-pkix-ocsp-nonce`) in its request to the server, which will be included in the response signed by the server.

Another possible attack is a **denial of service** one, accomplished by flooding the OCSP server with many requests. One little detail is esclusive of an OCSP dos attack: requests have to be digitally signed, which takes a lot of time to verify for the server and makes the attack easier. The obvious defence is to remove the real-time signature and replace it with a pre-computed responses, which requires three timestamps:

- thisUpdate – the response is based on revocation information available now
- nextUpdate – is again a promise.
- producedAt – this answer was created in this moment.

These answers are created even if no question is made. Since we are creating the answer without waiting for the request there is no nonce. In order to protect from DoS attack a problem about the replay attack is created. What some companies do is to use pre-computed responses and try to make them fast enough (e.g, the response is valid only for 30 minutes. Again there is the need to look at the policy).

3.5 Proof of possession(POP)

One of the issues we discussed in this chapter is the fact that, in the registration protocol that we explained, the identity of the ee is verified by the CA, but the CA is not sure that the private key is really in its possession.

> Having a Proof of possession means that the certification authority has a guarantee that the private key is in possession of the subject requesting the certificate.

If a certification authority creates a certificate without proof of possession, then there are various attacks possible, meaning that it is imperative to achieve non-repudiation. On the other hand, POP is not critical for encryption.

3.5.1 Risks in the absence of POP

Let's consider the case in which we don't have proof of possession.

Alice creates a private key and stores it locally then sends a public key and an identifier to the CA. The CA verifies Alice's identity and creates a certificate associating Alice to that public key. Then there's Bob who sends to the same CA the same public key of Alice (copied from another certificate) and his identifier. If the CA doesn't perform POP, it will create another certificate with the same public key associated to Bob and here's the problem. When Alice signs a document, it may happen that when someone is contesting that document, she could claim that Bob signed it, so there is no non-repudiation. Another case may be that Alice claims that she signed a document, but Bob may say it was him signing it. So, Alice can deny a signature or Bob can claim his signature. That means that POP for any case of non-repudiation and attribution, especially in legal matters, is a very important thing.

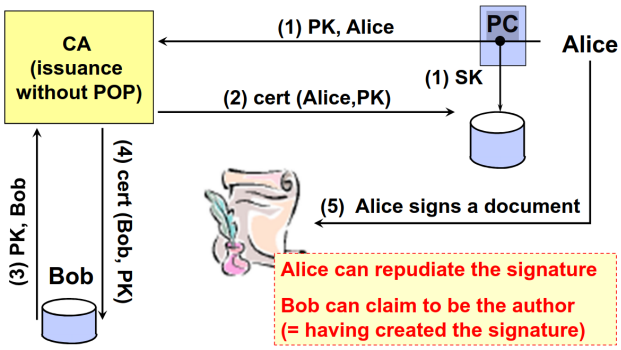


Figure 3.7: Risks in the absence of POP

Countermeasures

> The best solution to demostrate the POP is at signing time.

The signer inserts a reference to the certificate (e.g. a hash) among the signed information, thus the signature value is a function of the certificate too (depends on it). Unfortunately, most of the security protocols do not perform this kind of task. There are some custom protocols designed for protecting electronic documents that keep this into consideration.

Another solution is quite simple: the CA issues a certificate only if it has the proof that the certificate requester owns the private key. POP with this method can be achieved in two methods:

- Out-of-Band: the keys are not generated by the user, but are generated by the CA or the array and delivered in a secure token, for example a smart card. (This is what

happens at politecnico). The possession of the token is proof enough

- In-Band/Online: the CA will keep a copy of all the private keys, but the problem is how to protect them efficiently. This is better but it requires a secure device or online methods. If the key is used for both signature and encryption, then could be possible to use the self-signed formats like PKCS#10 or SPKAC. When a request is performed there must be also a signature (which means using a private key), even if there is no certificate. the CA verifies the signature before creating the certificate. If on the contrary, the key that you are trying to certify is a purely encryption key, then the requester cannot sign anything because that is not a valid usage. In that case, you can use a challenge response protocol. The CA sends a challenge to the requester, sending an encrypted certificate with the public key of the requester.

3.6 PKCS#10

Focusing on PKCS10, it represents one of the most commonly used formats today. It is specified in two RFCs: one detailing the basic syntax, and the other focusing on its application. Despite its status as an RFC, the format retains its original numbering and is also referred to as a Certificate Signing Request (CSR).

A typical request includes:

- The distinguished name
- The public key
- Several optional attributes

One of the notable optional attributes is the *challenge password*. This serves as a one-time password that may be provided by a registration authority and can be utilized during both registration and revocation processes. Revocation, in particular, involves both administrative and technical aspects.

For instance, if a smart card is lost while in Torino, the individual might visit the office to have their identity verified and initiate revocation. However, if the loss occurs while abroad, such as in Japan, physical presence at the office may not be feasible. In such situations, an online method for initiating revocation is necessary.

While it may seem that simply making a phone call or using a website to request revocation would be straightforward, this approach could lead to unauthorized individuals blocking certificates. Therefore, when submitting a revocation request online, there must be some means of verifying that the request comes from the legitimate certificate owner. This is often done through the use of the challenge password, providing proof of ownership.

In addition to these elements, the request may also contain further attributes and information pertaining to the requester.

3.6.1 PKCS#10 format

The format of a PKCS#10 request is shown in figure **??**. The data to be certified, made up of the DN, the public key and the optional attributes, is inserted at the beginning of the certificiate. We know that the certificate contains a signature generated by the private key

of the requester, and this is computed over the data to be certified. The signature is then inserted at the end of the certificate. This is also a proof of possession.

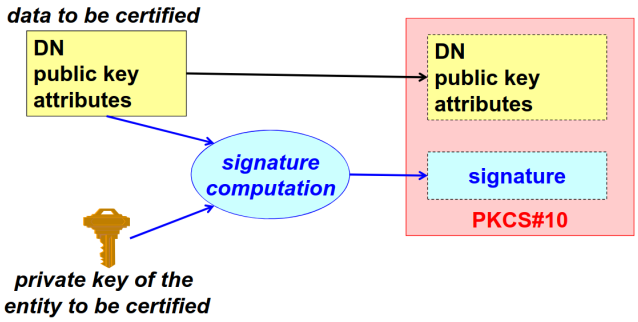


Figure 3.8: PKCS#10 structure

3.6.2 Time-stamping

In certificates, time handling is one of the most important aspects. For that purpose we use time-stamping.

> A time-stamp is the proof of creation of certain data **BEFORE** a certain point in time.

It demonstrates that data existed in that moment, but it doesn't tell anything about when it was created, certainly was created before but it is not known when, and this is wrongly assumed by many people.

Time-stamping is normally performed by a **time-stamping authority** (TSA) that uses a specific protocol and data format for its operation. The RFC-3161 is defining the request protocol (TSP, **Time-Stamp Protocol**) and the format of the proof (TST, Time-Stamp Token).

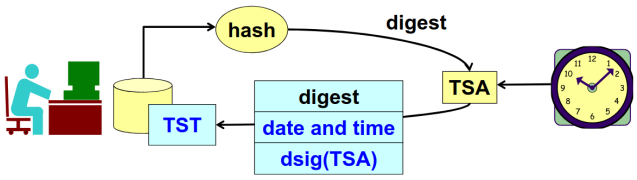


Figure 3.9: the time-stamping process

The user has some data created at some point in time, which he wants to time-stamp to demonstrate that those data existed in that moment. The user computes a hash (it does not send data for privacy reason) and sends it to the TSA. The authority receives the digest and consults a very precise clock(timing is fundamental for this procedure) and will create the TST, a document that will contain the digest, the date and time and the digital signature of this data performed by the time-stamping authority.

This is demonstrating that it exists in this moment, or at least that it was created before.

As we just sait, the timestamp does not certify a moment in time, but it is possible to do so with a little trick. Assume that one wants to certify the time of a digital signature, with one timestamp T_1 we can only certify that it was applied before the timestamp, but we can add another timestamp T_2 to certify that the signature was not applied before T_1 to narrow the time frame, meaning that $T_2 <$ signature time $< T_1$. If this time frame is small enough, it can be very accurate.

3.7 Personal Security Environment (PSE)

Let's talk for a second about something else than the public key. Most people remember that the private key should be protected and be secret, of course, but they forget that also the certificate of the trusted root CA should be protected, not for confidentiality but for authenticity. After all, a common attack is to substitute the certificate of the trusted root CA with another one. As an additional security features, the private key should not be exportable, meaning that the device in which the private key is stored should be the one used to do the operations.

For all those reasons, the protection mechanism are performed trough a **Personal Security Environment (PSE)**.

It is possible to implement PSE in software: typically it's an encrypted file because it contains also the private key and typically also the root CAs. Root CAs would not require encryption but it is done anyway since there is also the private key.

PSE can be also hardware. There are here two options: passive system which is only a memory (like a USB pen) so it is the same as a software PSE but hardware; active system which means that not only protects the keys but also performs cryptographic operations using those keys.

3.7.1 Cryptographic smart-card

Typically, the used device is a **cryptographic smart-card**, which is a chip card with memory and autonomous cryptographic capacity. The keys need to be managed by a trusted system, if there is an encrypted file but then the CPU is used to perform the computation it means that the key must be put in clear in RAM for the CPU to be used and if there's a malware then the key could be stolen. On the other hand, if there is a secure smart card, when the signature is needed the CPU is sending the hash and the smart card is returning the signature, because the smart card has the capability to perform the computation. That is a cryptographic smart card and the difference from a smart card for collecting points, for example, is that those are only memory card, unlike the Polito smart card which is a cryptographic smart-card.

These cards have microcontrollers (CPU+IO), RAM and especially an E^2PROM and a cryptographic coprocessor. The E^2PROM is a protected permanent memory where is typically stored the private key, while the cryptographic coprocessor is the one that can use the key to perform the computation. The card will never give out the key but instead will perform the computation itself. Depending on the costs, it is possible to have various algorithms of various lengths, the keys can be generated on board or can be generated

outside and then injected. Typically, the memory has not lot of space. Nowadays we're moving from smart card to smartphones, which has both pros and cons. The smart-card is a dedicated device that is not affected by malwares, while smartphones are used also for other purposes and might be affected by viruses.

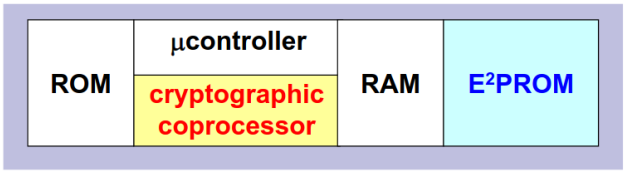


Figure 3.10: Cryptographic smart-card

A smart card is typically used by an individual and is quite slow, it takes 1-2 seconds to perform one signature because of the small CPU inside that works at MHz speed and because the I/O interface is serial, one bit a time, so usually is like 9000 bits per second. This is good to perform just a signature, but for example, a server that needs to perform a signature needs an HSM.

3.7.2 Hardware Security Module (HSM)

A cryptographic accelerator, commonly known as a hardware security module (HSM), provides enhanced security by offering protected memory, where cryptographic keys are stored securely. Even if an attacker were to access the memory, the keys cannot be extracted. In addition to protected storage, these modules typically feature a cryptographic coprocessor, which is responsible for performing cryptographic operations. These are especially useful in servers for handling both asymmetric encryption (e.g., RSA) and symmetric encryption, thanks to their high-speed I/O capabilities.

HSMs can come in various forms, such as PCI boards, external devices (e.g., USBI, SCSI), or network devices (e.g., netHSM). They are essential in environments that require strong cryptographic protections, such as electronic commerce, where TLS servers benefit greatly from having an HSM.

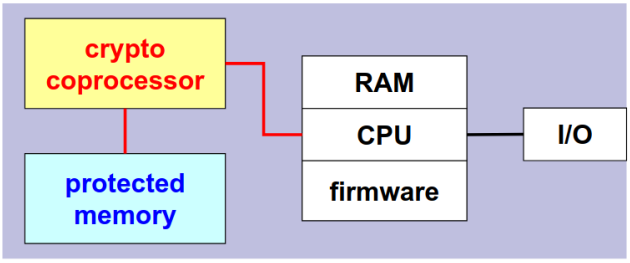


Figure 3.11: Hardware Security Module (HSM)

However, integrating HSMs into cloud infrastructure presents challenges. Cloud servers, being virtual, can be moved between physical nodes, but HSMs are tied to specific hardware. This raises the question of how to ensure secure key management for virtual servers. Different cloud providers have developed various solutions, such as virtualized HSMs, to address this issue and maintain the necessary security level in cloud environments.

3.7.3 ISO 7816-x standards for smart-card

The ISO 7816-x standards define the physical and logical characteristics of smart cards, there are many standars:

1. Physical format
2. Contact characteristics
3. Electrical signals and protocols
4. Inter-industry commands
5. Application identifiers
6. Inter-industry data elements
7. SCQL (SC Query Language)
8. Inter-industry security commands
9. Commands for card management
10. Electronic signals and answer to reset for sync cards
11. Personal verification through biometric methods
12. Cards with contacts — USB electrical interface and operating procedures
13. Commands for application management in multi-application environment
14. Cryptographic information application

The most important ones are 8, 11 and 14.

3.8 PKCS#12

If it is wanted to implement PCI in software, one of the solutions is PKCS#12, also called Security Bag. It is defined in RFC-7292 and it is used to transport/store cryptographic material among different applications and different devices in a neutral way.
It contains a private key and one or more certificates to transport the digital identity of a user. It's very commonly used, for example by Java, Microsoft, Mozilla, Google, etc.
Beware that the extension is ".p12" but for Microsoft the files are named ".pfx", which has the same content with just a difference. Microsoft has preferred speed over security, so since PKCS#12 has a certain number of rounds to make exhaustive attacks impossible, the Microsoft implementation is using the lowest possible number of rounds (more easily attackable). The suggest is to create PKCS#12 with another system (Mozilla, Google) and then use it on Microsoft (for importing) but do not export from Microsoft because they will create a lower security version of the PKCS#12.

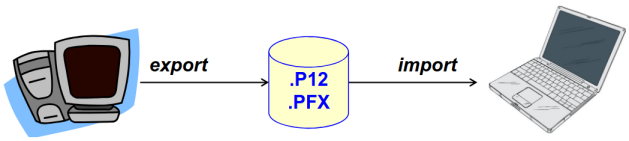


Figure 3.12: PKCS#12

3.8.1 How-to display a X.509 certificate

The following are some useful commands to manage certificates with OpenSSL:

- **openssl asn1parse**
  - Displays the actual ASN.1 structure in abstract syntax notation.
  - May convert from the input format (PEM or DER) to DER.
- **openssl x509**
  - Signs/verifies a certificate.
  - The **-text** option displays the content of the certificate in text format.
- **dumpasn1**
  - It is not part of OpenSSL but displays the content of the certificate in a similar way to **asn1parse**.

Note that when managing certificates, it is possible to have the ASN.1 format of the certificate stored in two different ways:

- **DER** (Distinguished Encoding Rules), which is a binary encoding of ASN.1 (a subset of **BER**, Basic Encoding Rules).
- **PEM** is the armoured base64 encoding of DER.

3.9 PKI organization

PKI is a complex system that requires careful organization, in fact it must be organized in such a way that allows for verification of the CRCA (Certificate Revocation Certificate Authority) trustworthiness.

3.9.1 Hierarchical PKI

A hierarchical Public Key Infrastructure (PKI) is structured as a tree rooted at a self-signed root Certificate Authority (CA). This root CA provides the foundation for trust within the hierarchy, but its self-signature introduces a degree of risk due to its central role as the root of trust.
One of the primary advantages of a hierarchical PKI structure is the ease with which certification paths can be established between any two End Entities (EEs). By traversing up the tree to a common point, a certification path can be constructed, making this model

highly compatible with applications that natively support it. Applications that use X.509 certificates are referred to as using PKI.
However, due to a range of legal, commercial, and political challenges, there is no single global PKI hierarchy. Instead, the PKI ecosystem is better described as a forest, with multiple independent hierarchies or "trees." Consequently, systems often require several root CAs, each serving as a distinct root of trust and creating its own separate hierarchy.

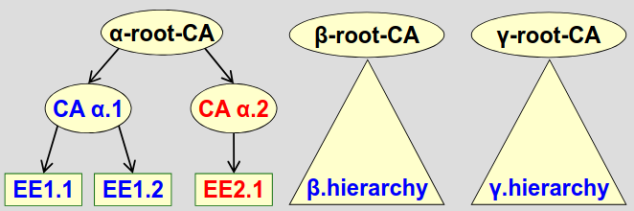


Figure 3.13: Hierarchical PKI

3.9.2 Mesh PKI

Mesh PKI is a model designed to enable trust between entities in different PKI hierarchies. In a standard PKI setup, different hierarchies usually don't trust each other, which can be a hassle when entities from one hierarchy need to communicate securely with those in another. To solve this, Mesh PKI introduces the concept of a **cross-certificate**, a special X.509 certificate that allows two PKI hierarchies to trust each other either unilaterally or bilaterally. Essentially, a root CA in one hierarchy can issue a certificate for the root CA in another. For example, if the root of hierarchy H3 issues a certificate for H2, then H2's root has two certificates: one that's self-signed and another from H3. In this case, when someone in H2 wants to be trusted by H3, they'd ideally use the certificate issued by H3 instead of the self-signed one.
This setup requires applications to handle cross-certificates and pick the right certificate chain based on who they're communicating with. So, if an entity in H2 knows it's sending information to someone in H3, it can use the H3-issued certificate for smoother verification. However, if it doesn't know the verifier's hierarchy, it might need to include multiple certificates, hoping one matches the verifier's trust path.
Although Mesh PKI seems practical, it has some drawbacks. Applications generally don't recognize cross-certificates automatically, so they may struggle to pick the correct chain. Plus, if there are a lot of hierarchies, achieving universal trust would require a ton of cross-certificates—about $\frac{N(N-1)}{2}$, where N is the number of hierarchies. Due to these limitations, especially in applications, Mesh PKI is mostly theoretical and isn't widely used.

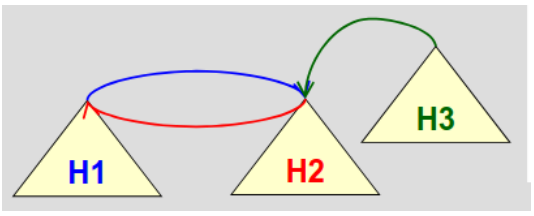


Figure 3.14: Mesh PKI

3.9.3 Bridge PKI

In certain sectors, some organizations have opted for the bridge PKI model over a traditional forest of hierarchies. The bridge PKI approach introduces a special entity called the "bridge CA," which serves as a central hub to simplify management tasks—such as adding or removing a CA—and streamline trust transitivity across organizations. In this model, the bridge CA is trusted by all the root CAs and establishes cross-certification with each one. Importantly, the bridge CA doesn't certify any other CA or EE; it only manages trust relationships between root CAs.
Adding a new CA in this model is straightforward: only two certificates are needed, one issued by the bridge CA for the new CA (e.g., H3) and another by the new CA for the bridge. This arrangement enables bidirectional trust, allowing all users within the network to reach any other user. However, like with mesh PKI, standard applications often don't recognize the bridge PKI model automatically, so modifications to applications may be required to manage these cross-certifications.
A prominent example of bridge PKI in practice is the U.S. Federal PKI. In this setup, each department or local government manages its own hierarchy, and a federal bridge CA issues the necessary cross-certificates to link them. Additionally, some browsers, such as modified versions of Chrome and Edge, support this model to enable compatibility with the federal PKI. You can learn more about it at `https://fpki.idmanagement.gov/`.

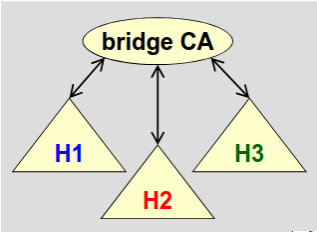


Figure 3.15: Bridge PKI

Handling Public Key Certificates Issued Mistakenly or by Compromised CAs

Detecting fraudulently issued certificates for a domain is challenging for domain owners. Imagine managing servers for an institution like Politecnico di Torino. Under normal circumstances, a valid certificate ensures secure communication. However, if an attacker manages to obtain a certificate for this domain from a different CA, and this in some instances is unexpectdly easy (think about the Common Law and its trust system), this fraudulent certificate would still appear legitimate, as no notification mechanism exists to alert the domain owner of unauthorized certificate issuance. This poses a risk, as clients connecting to a compromised or fake server using this certificate would see it as trustworthy.
Browsers are also not well-equipped to detect these malicious certificates in real-time. Specifically, in a TLS connection, a browser may not flag:

- Certificates mistakenly issued to an unauthorized entity
- Certificates issued by a compromised CA

When such certificates are detected, revocation is required to alert browsers. However, this process takes time, leaving a window during which a browser may continue to trust a compromised certificate. This vulnerability can lead to severe attacks, such as:

- Connections to a fake server masquerading as the legitimate site
- Man-in-the-middle (MITM) attacks, where an attacker intercepts and manipulates data in transit

Some examples of mistakenly issued certificates

Instances of mistakenly issued certificates highlight the risks of unauthorized certificate issuance by certificate authorities (CAs). Notable cases include:

1. **2011: DigiNotar Incident**
   In July 2011, an intruder acquired a valid certificate for the domain `google.com` and its subdomains from the prominent Dutch Certificate Authority DigiNotar. This certificate may have been used maliciously for weeks before its detection on August 28, 2011, enabling large-scale MITM attacks against users in Iran. Some suspect the involvement of state actors aiming to spy on users of Google services. By setting up a fake website, attackers could intercept traffic to legitimate Google servers, facilitating covert surveillance.

2. **2011: Comodo Group Incident**
   Also in 2011, the Comodo Group suffered a security breach that resulted in the issuance of nine fraudulent certificates for domains owned by Google, Yahoo!, Skype, and others. This attack underscored the vulnerability of even well-established CAs to unauthorized certificate issuance.

3. **2014: India CCA Incident**
   In July 2014, a sub-CA of India CCA, specifically the Indian NIC (Network Information Center), misissued a substantial number of certificates. Given the incident's scope, with numerous unauthorized certificates generated, the sub-CA was fully revoked. Additionally, India CCA was restricted to issuing certificates solely for specific domains within India's namespace. For example, it could no longer issue certificates for general domains ending in ".in" but only for designated domains within that scope.

3.10 HTTP Key Pinning (HPKP)

HTTP Key Pinning (HPKP) is a security feature designed to protect HTTPS websites from impersonation attacks by modifying the HTTP protocol. In this approach, a site specifies the digest of its own public key and/or one or more Certificate Authorities (CAs) in its chain, excluding the root CA.
When a user agent (UA), typically a browser or an application, accesses the site, it caches the specified key(s) and subsequently refuses to connect to the site if it presents a different key. This mechanism employs a Trust On First Use (TOFU) technique, meaning that trust is established during the initial connection. However, if the first connection is made to a fake site, the UA may refuse to connect to the legitimate site afterward.

While HPKP provides some protection, it also poses several risks and challenges:

- Losing control of the key can lead to significant security vulnerabilities. For instance, if a hardware security module (HSM) is broken and the old private key is lost, no one will trust the new private key.
- Key updates can be problematic, especially if keys are changed periodically. If the current key is lost without a backup, it can lock users out from connecting to the server.
- To mitigate these risks, it's advisable to always include at least two keys: the current key and a backup key that is already provided for future use.
- A URI for reporting violations can be specified, allowing the site to operate in either enforcement mode (reporting refused connections) or report-only mode (notifying about connections to servers with different keys).

The primary purpose of HPKP is to protect against fake websites attempting to impersonate legitimate sites using fraudulently obtained public key certificates (PKCs). However, due to its limitations and potential risks, HPKP has been deprecated in favor of Certificate Transparency, which offers a more robust mechanism for monitoring and validating the authenticity of certificates.

3.11 Certificate Transparency (CT)

Certificate Transparency (CT) is an open, global auditing and monitoring system designed to enhance the security of digital certificates. More information can be found at `https://www.certificate-transparency.org/`. CT is based on a public log of issued certificates, allowing domain owners, such as Politecnico Torino, to verify that no fraudulent certificates have been issued for their domains.

In the past, certificates issued were only visible by scanning various repositories. Now, with CT, there is a publicly available log that enables domain owners to scan and check if any certificates have been issued for their domain. This transparency helps prevent issues arising from maliciously issued certificates.

Originally proposed by Google, CT was later standardized by the IETF Public Notary Transparency Working Group. Key features of Certificate Transparency include:

- The issuance and existence of TLS certificates are made open to analysis by domain owners, Certificate Authorities (CAs), and domain users.
- Web clients, including browsers and apps, should only accept certificates that are publicly logged in CT.
- It should be impossible for a CA to issue a certificate for a domain without it being publicly visible in the CT logs.

Implementing Certificate Transparency requires modifications in both browsers and apps. When a web client receives a certificate, it must:

- Cryptographically validate the certificate chain.
- Check the revocation status for each element in the chain.
- Verify that the certificate is present in the public log.

These requirements necessitate changes in the issuance procedures of CAs to ensure that no certificate can be issued without being logged publicly, thereby mitigating the risk of fraudulent certificate issuance and enhancing overall web security.

> The main idea of certificate transparence is to make impossible (or at least very difficult) for a CA to issue a PKC for a domain without making it visible to the domain owner.

It also aims provide an external system to the CA that lets any domain owner or CA determine if these fraudulent certificates have been created while also protecting the users from being given fraudulent certificates.

3.11.1 CT – Log Servers

Log servers are the core of the Certificate Transparency (CT) system. They play a crucial role in maintaining a secure log of TLS certificates. Key characteristics of log servers include:

- **Append-only**: Certificates can only be added to a log; they cannot be deleted, modified, or retroactively inserted. This design ensures the integrity of the log.
- **Cryptographically assured**: Log servers utilize cryptographic techniques to ensure that the logs are tamper-proof.
- **Merkle Tree Hashes**: These hashes are employed to prevent tampering and misbehavior within the log, providing a structured way to verify the integrity of the data.
- **Publicly auditable**: Anyone can query a log using HTTPS GET and POST requests. This feature allows users to verify that the log is functioning correctly and to check that a specific TLS certificate has been legitimately appended to it.

CT – Log Signature and Support

Log entries must be digitally signed to ensure their integrity and authenticity. The specifications for log signatures include:

- **Signature Algorithms:**
  - For version 1.0: NIST P-256 or RSA-2048
  - For version 2.0: NIST P-256, Deterministic ECDSA, or Ed25519
- **Deterministic ECDSA:**
  - Based on RFC-6979, titled “Deterministic Usage of the DSA and ECDSA.”

- If the K value used in the computation is not random, the secret key (SK) can be computed from the signature, which poses a problem particularly in embedded systems.

- **CT Support in Browsers:**
  - In Chrome/Chromium/Safari:
    - 1 Signed Certificate Timestamp (SCT) from a currently approved log.
    - Duration $<$ 180 days: Requires 2 SCTs from once-approved logs.
    - Duration $>$ 180 days: Requires 3 SCTs from once-approved logs.
  - In Firefox: CT support is pending implementation.

3.11.2 CT – Operations

In the Certificate Transparency (CT) system, anyone can submit a certificate to a log server, although most submissions will typically come from Certificate Authorities (CAs) and server operators. Upon submission, the logger provides each submitter with a commitment to log the certificate within a specified time frame. This commitment is known as a Signed Certificate Timestamp (SCT), which accompanies the certificate throughout its lifetime, ensuring ongoing verification.

The SCT, created by the log servers, is delivered alongside the certificate issued by the CA through various mechanisms. One common method is using an X.509v3 extension to include the SCT directly in the certificate. Alternatively, the SCT can be integrated as a TLS extension during the handshake process, or it can be stapled to the OCSP response, providing real-time validation of the certificate's status.

SCT via X.509v3 extension

Before issuing a certificate, the CA contacts the Log servers and sends a pre-certificate to it. Log server accepts the pre-certificate and returns the SCT by logging that the CA promised to issue a certificate for a web server. The SCT is ready, then the CA will attach the SCT to the pre-certificate as an X.509v3 extension, then it is signed and sent to the web server. From this moment, the web server when performing the TLS handshake will send its certificate together with the SCT. That means that the browser does not have to look for the SCT.

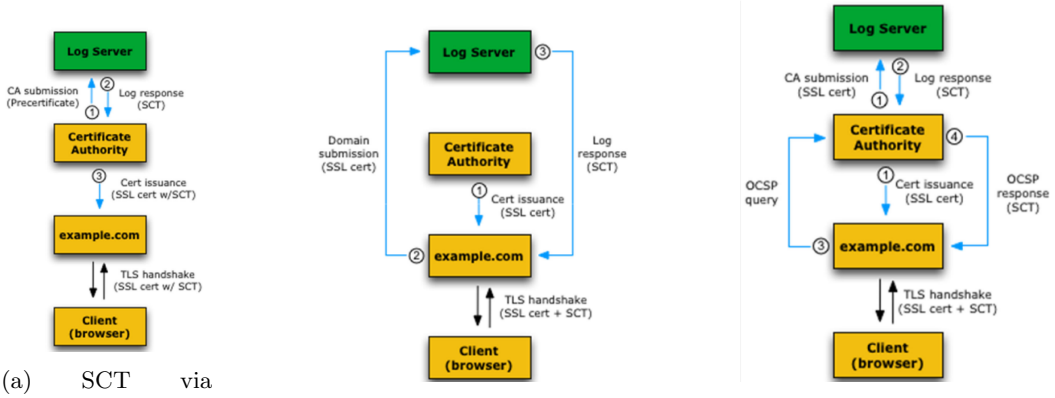
SCT via TLS extension

In this situation, CA issues a normal certificate to the server, and server operator (the owner of a website) submit it to the log server. Log server sends the SCT directly to the server operator and now the web server can deliver it to the client, this time separately, using the `signed_certificate_timestamp` TLS extension and it is delivered during the TLS handshake.

SCT via OCSP Stapling

In this case, the CA informs the Log server of the creation of the certificate and will get the log response, but this SCT is not part of the certificate, since it has already been issued

to the web server. When the website will perform OCSP stapling (when it will require an OCSP response from the CA) the OCSP response from the CA will contain also the SCT. Now, the SCT will be transmitted to the browser as part of the OCSP response. They are different strategies that depend on the specific situation. There is no reason to use the TLS approach if the CAs is providing the service, for instance.



(a) SCT via X.509v3 extension (b) SCT via TLS extension (c) SCT via OCSP Stapling

Figure 3.16: SCT delivery methods

3.11.3 Submitter and Monitors

In the Certificate Transparency (CT) ecosystem, submitters are responsible for submitting certificates, or partially completed certificates, to a log server, after which they receive a Signed Certificate Timestamp (SCT). This SCT serves as proof that the certificate has been logged.

On the other hand, Certificate Monitors, which are public or private services, play a critical role in maintaining the integrity of the system. These monitors, not previously detailed in the schemas we've discussed, actively watch for misbehaving logs or suspicious certificates. They periodically contact log servers to download the latest information and inspect new entries. Additionally, they maintain copies of the entire log and verify the consistency between the published revisions of the log. This vigilant monitoring helps identify potential issues, ensuring trust in the certificate issuance process.

3.11.4 Submitter and Monitors

In the Certificate Transparency framework, submitters play a crucial role by submitting certificates, or partially completed certificates, to a log server in exchange for a Signed Certificate Timestamp (SCT). This SCT serves as proof that the certificate has been logged properly.

Certificate Monitors, which may be public or private services, are responsible for ensuring the integrity of the system. These monitors actively watch for misbehaving logs or suspicious certificates. They periodically contact log servers to download the latest information and inspect new entries, while maintaining copies of the entire log. They verify the consistency between published revisions of the log.

Auditors in this context are typically lightweight software components rather than individuals. They function by verifying the overall integrity of the logs, periodically checking log proofs. A log proof consists of a signed cryptographic hash of the log, ensuring that a particular certificate appears in the log. This is essential since the CT framework mandates that all certificates must be registered; a certificate that hasn't been registered is deemed suspect.

While TLS clients are not required to enforce certificate transparency, it is advisable for them to provide warnings when connecting to a site lacking transparency. For instance, a warning might state, "Beware, you are connecting to a website that cannot provide certificate transparency. Do you want to proceed?" If a TLS client, through an auditor, determines that a certificate is absent from the log, it can use the SCT from the log as evidence that the log has not behaved correctly. Furthermore, auditors and monitors exchange information about logs through a specific protocol known as a gossip protocol.

3.11.5 A Possible CT System Configuration

A Certificate Transparency (CT) system can be configured in various ways, but a possible configuration includes the following components:

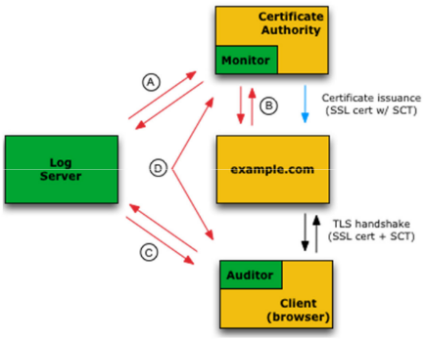
(A) Monitors continuously observe logs for suspicious certificates and verify that all logged certificates are publicly visible.

(B) Certificate owners can query these monitors to ensure that no illegitimate certificates have been logged for their domain.

(C) Auditors are responsible for verifying that the logs are functioning correctly and can also confirm that a particular certificate has been logged.

(D) Monitors and auditors exchange information about logs to detect any forked or branched logs, ensuring consistency within the system.

While CT does not prescribe any specific configuration, a possible setup might involve the following steps:

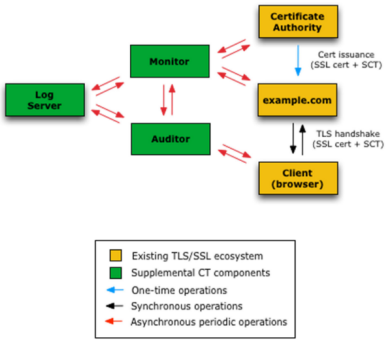


1. The Certificate Authority (CA) obtains an SCT from a log server and incorporates it into the TLS certificate using an X.509v3 extension.
2. The CA then issues the certificate, with the SCT attached, to the server.
3. During the TLS handshake, the TLS client receives the TLS certificate along with the SCT.
4. The TLS client validates the certificate and checks the log signature on the SCT to confirm that the SCT was issued by a legitimate log and corresponds to that certificate. If any discrepancies are found, the TLS client may reject the certificate.

In another configuration, monitors may be operated by the CA, while auditors can be built directly into the browser. In this case, the browser periodically sends a batch of SCTs to its integrated auditing component, requesting verification of whether these SCTs have been legitimately added to a log. Auditors then asynchronously contact the logs to perform the necessary verification.

Overall, monitors and auditors can operate as standalone entities, offering either paid or unpaid services to CAs and server operators.

Another possible configuration is shown in figure **??** , and involves placing the monitor and auditor outside the Certification Authority and the browser. In the previous setup, these components were integrated within the CA and the browser, which necessitates certain changes. However, this is not mandatory. By having the monitor and auditor operate as independent entities, we can minimize modifications to the existing certification authorities and browsers.



3.11.6 Current Log Servers

As of October 2023, the availability of log servers in the Certificate Transparency (CT) ecosystem is dependent on browser support. There are currently six valid log servers: CloudFlare, DigiCert, Google, Let's Encrypt, Sectigo, and TrustAsia. For a complete list, visit `https://certificate.transparency.dev/logs/`.

Additionally, there are ten public monitors that actively observe these logs: Censys, CloudFlare, crt.sh (by Sectigo), DigiCert, Entrust, Facebook, KEYTOS, Hardenize, sslmate, and ReportURI. It is worth noting that many of these monitors focus primarily on the domains they serve, such as CloudFlare, which monitors only its own domains. For more information on public monitors, refer to `https://certificate.transparency.dev/monitors/`.

3.12 ACME Protocol

The Automated Certificate Management Environment (ACME) protocol, detailed in RFC-8555 from March 2019, simplifies the management of public key certificates (PKCs) between End Entities (EEs) and Certificate Authorities (CAs). Created by the Internet Security Research Group (ISRG) for their CA service, Let's Encrypt (`https://letsencrypt.org`), ACME aims to make it easier for everyone to adopt Transport Layer Security (TLS) without the cost.

One major challenge in the world of certificates is the push for short-lived ones. Imagine needing to create a new certificate every 24 hours or even more frequently—that sounds like a nightmare if done manually! This is where automation comes into play. With ACME, you can automatically request and issue PKCs without needing a person to intervene each time. This is especially useful since some folks advocate for certificates valid for as little as five minutes. If a key gets compromised, the risk is limited because the certificate isn't valid for long.

Here's how it works: an ACME agent (or client) is installed on your web server. This client proves to the CA (Let's Encrypt) that it controls a domain. Once the CA verifies this, it allows the client to request and install certificates as needed. The client can even be programmed to perform certificate operations at fixed intervals—no more manually generating individual PKCS#10 requests or proving domain ownership every single time. You can also skip the hassle of downloading and configuring server certificates with ad-hoc procedures.

There are over 100 open-source ACME clients available, making it very easy to integrate this process into your workflow. Some popular ones include Certbot (from Let's Encrypt), GetSSL, Posh-ACME, Caddy, and ACMESharp. For a full list, check out `https://letsencrypt.org/docs/client-options/`.

By minimizing human involvement and keeping costs low, Let's Encrypt makes it possible for anyone to get free certificates, helping to spread TLS adoption far and wide. With ACME handling the heavy lifting, securing your web presence has never been easier!

If you want to use ACME for your own certificates, first of all, you must create an account at Let's Encrypt or the CA that supports ACME. Then you must perform domain validation once and for all, by demonstrating that you have the right to request certificate for a domain website. And then you will get a certificate and you will be able also to manage the revocation of certificates.

3.12.1 Account Creation

Account creation in the ACME system is a one-time process that must be completed before you can issue or revoke any certificates. During this phase, the ACME client generates an asymmetric key pair, which is used for both authentication and authorization purposes, although it's referred to as the authorized key pair.

The authorized public key is linked to the account registered at Let's Encrypt, while the authorized private key is used to sign your certificate requests. When you send a request to Let's Encrypt, they'll validate it using your public key.

It's important to note that a single account can be associated with one or more domains. While you only need one domain to start, you can also manage certificates for additional domains, such as those belonging to the University of Torino, under the same account.

3.12.2 Domain Validation

Domain validation is the trickiest part of the ACME protocol because the client must prove control over the domain without human intervention. For example, if the client claims ownership of `Polito.IT`, the Certificate Authority (CA) will issue a challenge. The CA might say, "If you control `Polito.IT`, please place this value in a specific path on your web server," meaning the client must create a web page with that value. Alternatively, the client can create a DNS record with the value, providing two options for validation.

In addition to placing the value, the client needs to sign a nonce with its authorized private key to associate the proof. After the client solves the challenge, it sends the signed nonce to the CA to initiate the validation process. The CA then downloads the response from the domain and verifies both the correctness and the signature.

If everything checks out, the CA approves the domain ownership by recording it in a database, enabling the client to request certificates for that domain. For instance, if the

client claims control of `example.com`, Let's Encrypt might challenge, "Please place the value `ED98` on the page at `https://example.com/80303`." The page should contain only that value.

The client, using its private key, signs the nonce and sends it alongside the value to the CA. Once the administrator places `ED98` on the specified page, Let's Encrypt verifies the response. This process links the client's public key to `example.com` in the CA's database, allowing it to make future requests for certificates. Notably, the private key used to sign the message is that of the client software, not the web server. The client acts like a chatbot, managing certificate requests for domains it controls, such as `example.com`.

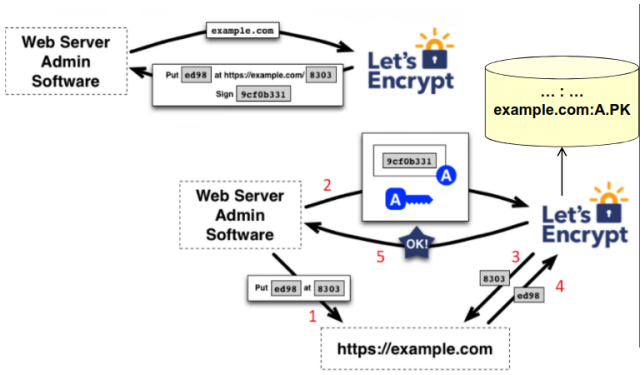


Figure 3.17: ACME Domain Validation

3.12.3 Certificate Request and Issuing

Once the ACME client has been registered, it can use the authorized key pair to request, renew, and revoke certificates for its domain. The client constructs a PKCS#10 Certificate Signing Request (CSR) automatically, so there's no need for manual intervention. It submits this request to the Certificate Authority (CA) to issue a certificate containing a specified public key.

In this process, the client takes the public key for a server and includes it in the CSR. The CSR includes a signature generated by the private key (SK) corresponding to the public key (PK) included in the CSR. Essentially, this means the request is double-signed: it is signed by the server's private key and also signed with the authorized private key associated with the domain.

For example, when requesting a certificate for `example.com`, the PKCS#10 structure remains unchanged, but the request is validated by the client's authorized key, demonstrating that it is a legitimate request. Once the CA (e.g., Let's Encrypt) verifies the request, it will return a signed certificate for `example.com` that corresponds to the specified public key, completing the issuance process.

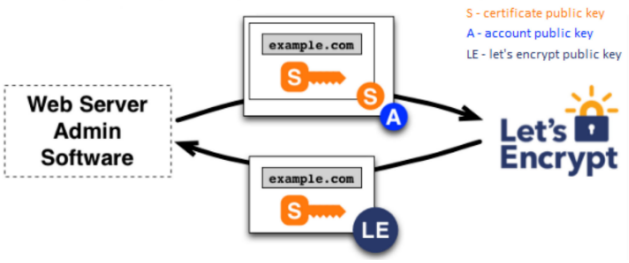


Figure 3.18: ACME Certificate Request and Issuing

3.12.4 Certificate Revocation

For certificate revocation, it is the ACME client, not the server, that signs a revocation request using the authorized private key (SK) associated with the corresponding domain. This ensures that the client has the right to initiate the revocation process. The Certificate Authority verifies the signature of the request to confirm the authorization.

Once verified, the CA revokes the specified certificate, which was initially issued by the CA. This revocation information is then included in the appropriate revocation mechanisms, such as the Certificate Revocation List and/or the OCSP. Consequently, when relying parties check the status of the certificate, they will receive accurate and up-to-date information regarding its validity. The use of the authorized key for this request is crucial, as it demonstrates the client's entitlement to perform the operation, ensuring that the revocation process is secure and properly managed.

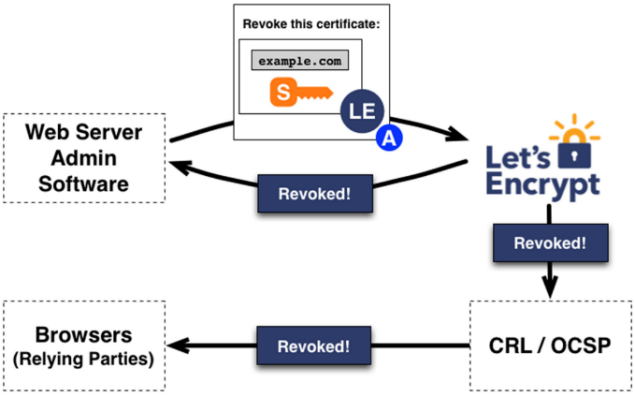


Figure 3.19: ACME Certificate Revocation

Chapter 4

Trusted computing

Nowadays, the modern computing system is made up of many highly distributed components.

Let's take into consideration the typical distributed infrastructure, it's made up of a cloud for storage and computation, which communicate with other devices outside its cluster, such as IoT devices, via edge routers. Securing all those components it's a real challenge, because each of those devices could be different, as well as their communication technologies(Wired, Wireless, etc.).

While in the past we had physical components, nowadays the trend is **softwarization of the components** on top of a generic commodity hardware, with "tools" such as SDN, NVF, etc.

> As a consequence, systems **more flexible** but **more vulnerable**.

After all, the larger the software base, the higher the probability of bugs, especially in the software, but hardware bugs are to be taken into consideration as well. All this without even considering the issue of software updates

One of the main issues nowadays is **trustworthiness**, that being if something behaves exactly as expected. However, there are several problems related to trust in this context:

- Trust in the cloud provider(s)
- Trust in the network/edge provider(s)
- Low or no access control for edge- and end-devices
- Low-cost IoT devices (which typically imply low security)
- Personal devices (often managed by users with limited security knowledge)

If possible, it is recommended to protect the infrastructure by avoiding or blocking all the possible attacks, which is a basically impossible task. If protection is not feasible, one should do the next best thing, that being monitoring the state of the system for early detection and reaction to attacks. IDS can help to this end, but they can be eluded by the attacker, so the best solution is to provide **integrity verification**, meaning that the system has not been tampered with, both of the software component as well as their configuration.

Integrity concerns can be categorized into hardware and software aspects:

- **Hardware:**
  - Am I communicating with the correct (intended) node?
  - Does it host the expected (physical) components? For this reason, each contry is developing their own components
- **Software:**
  - Am I communicating with the correct (intended) software component?
  - Is it correctly configured?
  - Is the baseline software the expected one?

As we just explained, trust is a big issue in modern computing, and in order to answer all those questions, we need to consider some solutions.

4.1 Trusted Execution Environment (TEE)

Systems are complex, and its difficult to trust every single component, but we can create a small environment that we can trust. But first, let's give again a definition of what trust is.

> Something sor someone is **trusted** if one can **rely** upon to **not compromise your security**, without any guarantees.

Similarly, something is **trustworthy** if it **will not compromise your security**, thinking about whether it is safe to use something or not.

If something is trustworthy it is trusted, but not vice versa.

Trusted Execution Environment is what you may choose to rely upon to execute sensitive tasks, which are called **Trusted Applications** (TA), and which one hope to be trustworthy.

TEE were originally developer for smartphones, which made them necessary due to the high number of apps running on the same environment, some of those with critical data(CC numbers, etc.), but nowadays they are used in a variety of devices.

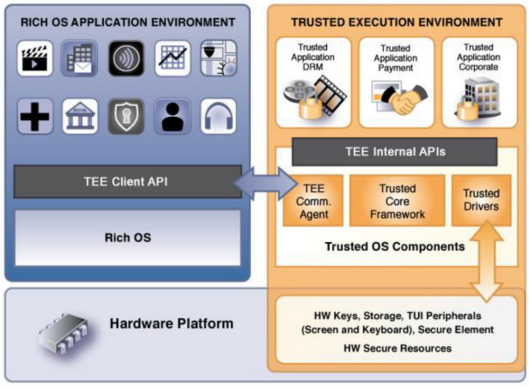


Figure 4.1: Trusted Execution Environment and Rich Execution Environment

As you can see from figure **??**, one one device can run different enviroments at the same time On of those is a **Rich Execution Environment** (REE), which is the normal environment in which one runs any applications or OS. On the other hand, the TEE is a separate environment isolated from the rest of the system to run critical tasks and has access to some trusted ankles like the hwardware keys, the secure storage, peripherals, etc. TEE can only run on some specific hardware platform made up of trusted components(not trustworthy, but trusted): the trusted drivers, the core framework(the execution core) and a set of API accessible only from trusted applications, the TEE communication agent

In modern computing, TEE has become an important subject, especially in the field of "confidential computing," which is actively promoted by the Confidential Computing Consortium (CCC). The primary goal of TEE is to protect "data in use," ensuring that nobody else can read or write the data. Only authorized applications are allowed to process the data. This approach contrasts with various cryptographic techniques used to protect "data at rest" and "data in motion."

A key component of TEE, necessary to provide it's services, is the **Root of Trust** (RoT), which is an element whose misbehavior cannot be detected during runtime. The RoT must be **both trusted and trustworthy**. It is also part of the Trusted Computing Base (TCB), which is the set of hardware, firmware, and software components critical to the system's security. Any vulnerability within the TCB poses a significant threat to the system's overall security.

4.1.1 TEE Security Principles

The security principles of a Trusted Execution Environment (TEE) involve the following:

- Being part of the device's secure boot chain (based on a Root of Trust) and verifying code integrity during each device boot.
- Hardware-based isolation from the device's rich OS environment to execute sensitive code.
- Isolation of Trusted Applications (TAs) from each other.
- Secure data storage, using a hardware-unique key accessible only by the TEE OS to prevent unauthorized access, modification, and any possibility of data exploitation on other devices.
- Privileged and secure access to peripherals (trusted path).
- Hardware isolation of peripherals (e.g., fingerprint sensors, displays, touchpads) from the rich OS environment, controlled only by the TEE during specific actions, with no visibility or access by the Rich Execution Environment (REE), including malware.

4.2 Some TEE Implementations

4.2.1 Intel Identity Protection Technology (Intel IPT)

Intel Identity Protection Technology (Intel IPT) is a security feature that operates on a dual-CPU system within Intel processors. This design leverages the Management Engine

(ME), a dedicated CPU that exists alongside the primary CPU in Intel processors. While the primary CPU executes user tasks, the ME, integrated into the chipset, can perform various management and security functions independently. Typically, the ME is unused in consumer devices, but in corporate environments, it can be activated even when the device is off via features like wake-on-LAN, allowing remote management over Wi-Fi or Ethernet.

Intel IPT uses the ME to run a **Java applet** isolated from the main CPU, offering enhanced security for tasks such as cryptographic key generation and storage, which integrates seamlessly with the Windows Cryptographic API. Additionally, Intel IPT supports secure One-Time Password (OTP) generation, as seen in applications like VASCO's MYDIGIPASS.COM, which use the ME to securely store secrets for OTPs. Another significant feature is secure PIN entry, where the chipset manages video output to ensure the security of PIN input by isolating it from the main system.

The ME's integration into the hardware, running on separate CPU architecture, exemplifies a physically separated Trusted Execution Environment (TEE), offering distinct security advantages by isolating critical operations from the main processing tasks.

4.2.2 ARM TrustZone

ARM TrustZone is a Trusted Execution Environment (TEE) implemented in certain ARM CPUs, designed to provide a secure and a normal mode within the same processor. To achieve this, TrustZone extends the CPU's bus with an additional "33rd bit" that signals whether the processor is in secure or normal mode. This signal is exposed outside the CPU, which enables secure peripherals and secure RAM by allowing the system designer to control access to memory and devices based on the security mode.

While the TrustZone framework is open and well-documented, it has some limitations. TrustZone can only support a single secure enclave at a time, and this security relies on software-based separation between applications rather than on distinct hardware boundaries, making it somewhat less secure than fully hardware-isolated environments. ARM is actively working to expand TrustZone by introducing a third operational mode to accommodate specific features like attestation. However, adding this capability presents challenges due to backward compatibility concerns, as ARM aims to preserve the success of its platform without disrupting existing applications.

4.2.3 Trustonic

The main issue with TrustZone is that it only one secure enclave. To address this issue, Gemalto developed the Trusted Foundations system, and Giesecke+Devrient (G+D) created MobiCore. Both solutions effectively divide the single secure enclave into multiple enclaves by leveraging a smart-card operating system. Trustonic's development is based on MobiCore, requiring license fees for implementing the code. Trustonic's TEE OS, named "Kinibi," includes enhancements such as version 500, which supports 64-bit Symmetric Multiprocessing (SMP) for embedded systems. Samsung Knox presents a similar approach but additionally incorporates secure boot functionality.

4.2.4 Intel SGX

Intel Software Guard Extensions (SGX) are tightly integrated with the CPU, providing a hardware-based TEE by modifying memory management to enhance security. SGX enables the creation of secure enclaves, which are isolated execution environments protected from access by other processes, even those with high priority. These enclaves achieve memory isolation, ensuring that only the code within an enclave can access its data, providing a high level of hardware-protected separation.

When an SGX enclave is created, the Intel SGX architecture performs a measurement, similar to Trusted Platform Module (TPM) practices, by computing a hash of the executable loaded into the enclave. This measurement-based security model is essential to SGX's integrity, ensuring that only approved code can run within an enclave.

For extended capabilities, SGX can be paired with Intel Identity Protection Technology (IPT), allowing features such as a trusted display. However, SGX itself is limited to CPU and memory protection and does not provide secure input/output channels. When trusted input-output is required, pairing with Intel IPT enables trusted interaction with the display.

Intel initially made SGX-1 available across both consumer and enterprise CPUs, but later revisions—specifically SGX-2—shifted focus towards server-oriented environments. SGX-2 is now mainly available on high-end CPUs, such as Intel's Xeon series, used in data centers. Furthermore, enclave creation within SGX requires special permissions and the use of Intel-specific libraries, and all enclave-bound code must be signed by Intel. This signing requirement means executable code must be submitted to Intel to receive a signature, which may be a barrier for some users.

4.2.5 Keystone

Keystone is an open-source framework designed for building Trusted Execution Environments (TEEs). It allows developers to select only the necessary features, which helps minimize the Trusted Computing Base (TCB), because smaller the trust "surface" the better. The architecture consists of an untrusted environment, such as a general-purpose operating system, combined with multiple trusted and segregated enclaves. Keystone is built on top of RISC-V, which offers customizable open-source hardware options, including Field-Programmable Gate Arrays (FPGAs) or System-on-Chip (SoC) designs. The framework incorporates core components along with cryptographic extensions and supports various execution modes, including Machine(M), Supervisor(S), and User(U) modes. Additionally, it features Physical Memory Protection (PMP), which has its hardware-based access control to different pages of memory. That avoids that one process can access the memory of another process, either in general or specifically during a period of time. This helps to safeguard memory and I/O operations, which are memory mapped.

Usually, TEEs are rigid and un-customizable, with many design implementation dictated from the underlying hardware, for example Intel SGX has a large software stack that is not customizable which means a larger TCB, which is undesirable, AMD SEV have the same issue and ARM TrustZone has a single TEE.

The architecture is shown in figure **??**. Keystone is structured to have a **Security Monition** as the only program running in Machine mode(the highest privilege mode), which provides which provides access control between any call coming from the upper layers

towards the hardware. It also provides an **untrusted domain** where one can run any OS compatible with the RISC-V architecture(even Linux), which will run in Supervisor mode. Furthermore, to minimize the TCB, no hypervisor or base operating systems are required.

On the other hand, one can have many **Enclaves**, which are isolated from the untrusted domain and from each other. Since they dont need a general purpose OS, they run on a **Keystone Runtime**, which is a small OS that provides the necessary services to the specific application. On top of this, the **Keystone Application** runs, which is the actual application that the user wants to run.

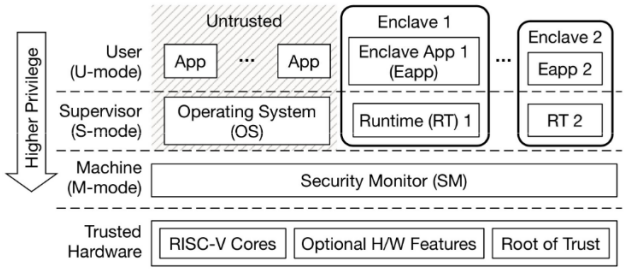


Figure 4.2: Keystone architecture

> For the exam, read the bloody papers.

4.3 Trusted computing and remote attestation

Attacker would obviously like to inject malwares at the lowest level possible, this is to remain undetected while still having access to the largest part of the system. For this reason the ideal scenario is to modify the OS is possible, or, alternatively boot an alternative one under the control of the attacker by modifying the boot sequence or even the bootloader.

It comes natural after this that the boot system and the OS should be protected in order to trust the system: for boot sequencing we once had the BIOS (Basic Input Output System), developed by specific hardware vendors(no general purpose one available) which was very difficult to protect, then we had UEFI (Unified Extensible Firmware Interface) , which is more secure with native support for firmware signature and verification. After this has been initialized, the OS can be verified before activation.

So the manufacturer of the platform will sign the firmware, the UEFI, and the hardware at boot will verify the signature. IF it fails, the firmware has been tampered with, and the system will not boot. This step is necessary to guarantee trust in the loaded OS.

4.3.1 Rootkits

This is the generic name for a tool that allows to have root access to a machine. Of those, there are many kinds.

Firmware Rootkits Firmware rootkits function by overwriting the BIOS/UEFI or other hardware firmware. This allows the rootkit to start running before the operating system even begins to load.

Bootkits Bootkits replace the bootloader of the operating system, ensuring that the bootkit is loaded first when the node is booted, which occurs before the OS itself initializes.

Kernel Rootkits Kernel rootkits manipulate a section of the OS kernel, enabling them to start automatically whenever the operating system loads, embedding themselves within the core processes of the OS.

Driver Rootkits Driver rootkits disguise themselves as trusted drivers used by the operating system (e.g., Windows) to interact with hardware. By mimicking a legitimate driver, these rootkits gain access to hardware resources while avoiding detection. Don't confuse drivers rootkits with firmware ones, the firmware is permanently stored in read-only memory, while drivers are part of the OS and loaded at boot-time.

4.4 Root of trust

In general, protecting software with software can not always be the best idea, as it may fail or have bugs. For this reason, we need hardware support to protect the software. At this end, we have a **Root of Trust**, which is a hardware component that is trusted to behave as expected, and is the foundation for the chain of trust. The RoT should be always part of the Trusted Computing Base because hardware is anyway operated by software or firmware

4.4.1 SW root-of-trust

An example from HP Enterprise illustrates a method for firmware self-protection, or software root-of-trust. HP Enterprise machines implements a designated "signature" region at a small fixed location within the final BIOS image, which is typically 16MB in size. Keep in mind that this was developed before UEFI.

During manufacturing, the SHA-256 hash of the custom BIOS regions is calculated. These regions include static code, BIOS version information, and microcode, but not the hash itself, as its yet to be initialized, at its only based on components that will never be updated. The computed hash is then sent to an HPE signing server, which returns a signed hash image (32 bytes) that includes the signature and certificate information. This signed hash image is stored in the BIOS "signature" region.

On power-up, the early BIOS code calculates a hash from the specified valid BIOS regions and verifies the validity of the stored "signature" contents. If the calculated hash matches the stored hash, the boot process proceeds; otherwise, the system halts, preventing further booting.

Of course, this method is not perfect, as it is still software-based and can be compromised. For example an attacker could replace the chip, which is soldered on the board, with a compromised one, or add another memory chip on the free region of the board, which code will be executed after the BIOS but before the OS without the integrity protection of the signature.

4.4.2 HW root-of-trust

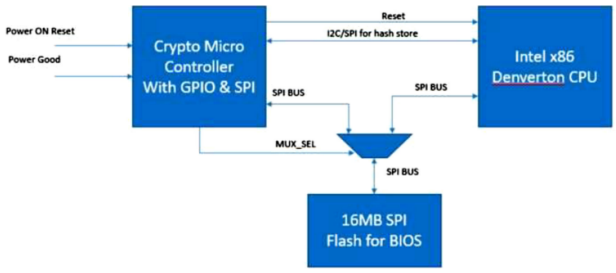
The mechanism just described tries to verify firmware from the firmware, but can we do better? Yes, we can use a hardware root of trust to verify the firmware.

A hardware root-of-trust for firmware protection typically includes self-verification, where a static portion of the firmware authenticates the updatable section. This approach allows for an internal method of validating firmware integrity directly within the firmware itself. Alternatively, firmware verification can be managed by an external chip, offering a secure, independent means of affirming firmware authenticity.

For example take a look at figure **??**, which shows a x86 Denverton CPU(which is quite old) and implements a 16MB SPI flash memory, which stores the BIOS. It presents a multiplexer in front of it that connects to both the CPU and an external cryptographic microcontroller which can both use the SPI bus and drive the multiplexer. So the decision of which chip can use the SPI bus is taken by the external cryptographic microcontroller, which will verify the integrity of the BIOS before allowing the CPU to boot.

Upon successful validation, this chip allows the x86 CPU to exit its reset state; otherwise, the CPU remains in reset, ensuring security, meaning that the chip is the true hardware root-of-trust.

The external chip may also include a fusing option, allowing a public key hash, which is smaller than the public key, to be securely fused and later used to verify the signature of the hash stored in the BIOS's signature region. This validation process is similar to the internal BIOS self-integrity check, with the added benefit of relying on an external chip, making it the true hardware root-of-trust.



4.5 Boot Types

Once we have been able to start the BIOS, the OS can be started. Different types of boot processes provide varying levels of security during system startup:

Plain Boot A plain boot involves no security checks, leaving the platform vulnerable to unauthorized modifications.

Secure Boot In a secure boot process, firmware verifies a signature before proceeding. If the verification fails, the platform halts, ensuring security from the earliest stages. This process is primarily hardware-based and verifies components up to the OS-loader.

Trusted Boot A trusted boot involves the operating system verifying signatures of critical OS components, such as drivers and antimalware software. If any verification fails, system operations are halted. You could have noticed that it assumes that the first part of the boot up to the firmware was OK and performs verification only of the operating system components. This type of boot is mainly software-based and extends verification up to the operational state of the OS.

Secure boot vs Trusted boot With secure boot, if the firmware(BIOS/UEFI/whatever) is compromised, it will not be loaded, whereas with trusted boot, if the firmware is compromised, the OS could still be loaded because the firmware is not verified.

Measured Boot During a measured boot, the system measures each component executed from boot through a defined point, denoted as X. Unlike other boot types, measured boot does not halt operations if verification fails. Instead, it can securely report these measurements to an external verifier, providing ongoing insight into system integrity.

What has been just explained is shown in figure **??**. Notice that the trusted drivers and the anti-malware are loaded before the OS.

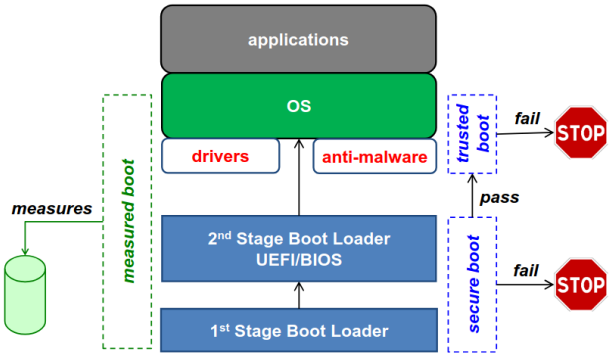


Figure 4.3: Boot types

Windows Boot protection

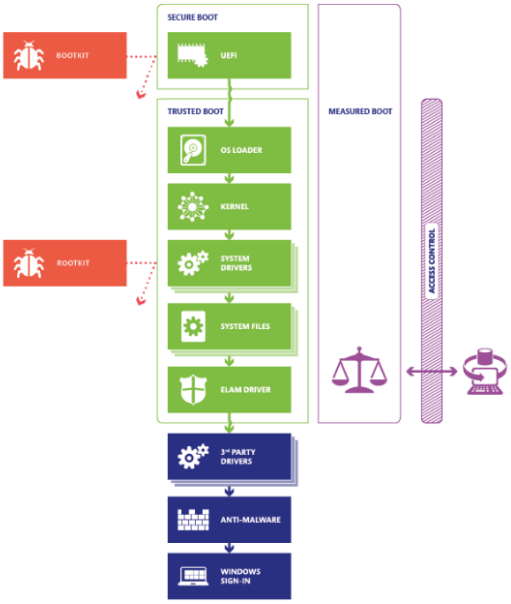
Windows 10 and Windows 11 implement a robust boot protection scheme that combines Secure Boot, Trusted Boot, and Measured Boot to secure the startup process against malicious interference, as shown in figure **??** . Secure Boot, which is managed by the hardware manufacturer (e.g., Dell, Lenovo, HP), is the first line of defense and prevents unauthorized firmware and bootloaders from running. This step effectively blocks bootkits and ensures that only trusted firmware initiates the boot sequence.

Following Secure Boot, Windows executes Trusted Boot, which verifies and loads essential operating system components in a specific order: first the OS loader, then the kernel, followed by system drivers, critical system files, and Early Launch Anti-Malware (ELAM) drivers. ELAM is a fundamental part of anti-malware solutions, as it starts before any user-space process, offering early protection against rootkits. The ELAM component must be submitted to Microsoft for signature to ensure its integrity, enabling a secure foundation for user-level anti-malware processes to operate later.

In conjunction with Secure Boot and Trusted Boot, Windows also utilizes Measured Boot, which records each step in the boot process for verification by an external verifier. If access is granted, this verifier can query the system to confirm its boot integrity and detect any potential tampering from the start. This integrated approach—Secure Boot to lock down firmware, Trusted Boot to load and verify core OS elements, and Measured Boot to maintain verifiable logs—creates a strong defense against unauthorized modifications during the boot process.

4.6 Trusted Computing

Trusted Computing encompasses systems and components designed to behave predictably and as expected. In this context, a component or platform is considered *trusted* if it consistently performs according to predefined expectations, though this does not inherently mean it is secure or "good." Trust in such systems requires verification against an expected behavior, rather than an implicit assumption of reliability.



A key concept within Trusted Computing is *attestation*, which provides verifiable evidence of a platform's state, allowing assessment against a known standard. Another fundamental concept is the *Root of Trust*, an inherently trusted component within the system that forms the basis for verifying trustworthiness.

Trusted Computing schemes establish trust in a platform by identifying its hardware and software components, often through a *Trusted Platform Module* (TPM). The TPM plays a crucial role in collecting and reporting component identities, offering a way to assess hardware and software configurations. This enables determination of whether a system's behavior aligns with expected standards, thus supporting the establishment of trust in the platform's integrity.

4.6.1 Trusted Computing Base (TCB)

> The Trusted Computing Base (TCB) refers to the **collection of system resources**, including both hardware and software, that is essential in upholding the security policy of a system.

An essential aspect of the TCB is its resilience against compromise; it must be able to protect itself from threats posed by any hardware or software outside of the TCB itself.

It is important to note that the TPM **is not** synonymous with the TCB of a system. Instead, the TPM serves as a tool for an independent entity to assess whether the TCB has

been compromised.

In specific implementations, the TPM may also play a preventative role, ensuring that the system does not start if the TCB fails to initialize correctly, thus providing an additional layer of security assurance.

4.6.2 Root of Trust (RoT)

> The RoT refers to a component within a system that must reliably act in an expected manner, as any deviation in its behavior cannot be detected.

The RoT is essential in establishing trust within a platform and comprises several foundational components.

The **Root of Trust for Measurement** (RTM) is responsible for taking integrity measurements and transmitting these to the **Root of Trust for Storage** (RTS). Typically, the CPU executes the **Core Root of Trust for Measurement** (CRTM) at boot, serving as the first element of BIOS/UEFI code to initiate the chain of trust.

The RTS provides a shielded(no one can modify it) and secure storage environment for critical integrity measurements. The RTR securely reports the content stored in the RTS, thus enabling verification of the system's integrity.

4.6.3 Chain of Trust

Our aim is, starting from the lowest level of the firmware, create a **chain of trust**.

In a Chain of Trust, each component verifies the integrity of the next component in sequence. Component A measures Component B and stores this measurement in the RTS. Then, Component B measures Component C and similarly stores the measurement in the RTS, continuing this process down the chain.

Typically, Component A is the CRTM, which is part of the TCB. By using the **Root of Trust for Reporting** (RTR), a verifier can securely retrieve the measurements of Components B and C from the RTS. For Components B and C to be trusted, Component A must itself be trustworthy.

4.6.4 Trusted Platform Module Overview

The Trusted Platform Module is an inexpensive component, typically costing less than 1 dollar, and is available on most servers, laptops, and PCs. It is designed to be **tamper-resistant**, although it is important to note that it is not entirely tamper-proof, meaning that it is difficult to compromise it, but not impossible.

While the TPM provides security features, it is not a high-speed cryptographic engine; in fact, it is relatively slow in terms of processing speed, m.t. doing RSA signatures with it will take forever. The TPM have to be certified with a **Common Criteria Evaluation Assurance Level**(EAL) of 4 or higher, which indicates a certain level of security assurance, which is quite good indeed.

As a passive component, the TPM requires the CPU to drive its operations. It does not have the capability to prevent the boot process; however, it can protect sensitive data and securely report this information. Consequently, the TPM functions as both the RTS and

the RTR, but it does not serve as the RTM, because we'll perform the measure and we'll store the results inside the TPM.

The TPM provides secure storage capabilities, functioning as a secure storage component (RTS) with an extend-only approach. It can report the content of this secure storage using a digital signature, thus acting as a reporting entity (RTR). This means that every time that memory is read outside the TMP, a digital signature computed with a asymmetric key pair stored inside the TPM is attached to the data.

One of the critical features of the TPM is its hardware random number generator, which supports various cryptographic algorithms, including hashing, Message Authentication Code (MAC), and both symmetric and asymmetric encryption. However, it is important to clarify that the TPM is not a crypto accelerator, as its performance is relatively slow.

The TPM also facilitates the secure generation of cryptographic keys for limited use cases. It supports **binding**, which encrypts data using the TPM bind key—a unique RSA key derived from a storage key. What does that mean? Binding means that if you want to decrypt this data, you can do that only on the platform that contains that TPM. Because if you move the data to another platform, they are encrypted with the key which is not there. This also means that if the platform broke down, you can't decrypt the data anymore.

Additionally, the TPM allows for **sealing**, a process similar to binding, but it also specifies the TPM state required for the data to be decrypted, or unsealed. This means that the data can be decrypted only if the TPM is in a specific state, which is useful to avoid tampering.

Furthermore, computer programs can utilize a TPM to authenticate hardware devices. Each TPM chip is manufactured with a unique and secret **Endorsement Key** (EK) that is burned in during production, ensuring that the authenticity of the hardware can be verified.

4.6.5 TPM 1.2

TPM 1.2 features a fixed set of cryptographic algorithms, which include SHA-1, RSA, and optionally AES. This version of the TPM provides a single storage hierarchy specifically for the platform user, ensuring that key management and storage are centralized.

At the core of TPM 1.2 is a single root key known as the Storage Root Key (SRK), which is typically an RSA-2048 key. This root key serves as the foundation for other keys within the TPM, facilitating secure storage and cryptographic operations.

Additionally, TPM 1.2 incorporates a built-in Endorsement Key (EK) (RSA-2048) that provides a hardware identity for the platform. This EK is essential for establishing the authenticity of the TPM and the device it resides in.

Another important thing of TPM 1.2 is its capability for sealing only against the PCRs value, where the measurements are collected.

4.6.6 TPM 1.2

TPM 1.2 features a fixed set of cryptographic algorithms, which include SHA-1, RSA, and optionally AES. This version of the TPM provides a single storage hierarchy specifically for the platform user, ensuring that key management and storage are centralized.

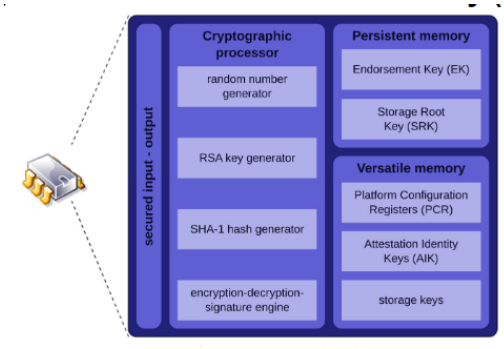
At the core of TPM 1.2 is a single root key known as the Storage Root Key (SRK), which is typically an RSA-2048 key. This root key serves as the foundation for other keys

within the TPM, facilitating secure storage and cryptographic operations.

Additionally, TPM 1.2 incorporates a built-in Endorsement Key (EK) that provides a hardware identity for the platform. This EK is essential for establishing the authenticity of the TPM and the device it resides in.

Another important feature of TPM 1.2 is its capability for sealing data to a Platform Configuration Register (PCR) value. This sealing process ensures that the data can only be decrypted when the platform is in a specific, verified state, enhancing the security of sensitive information.

4.6.7 TPM 2.0



TPM 2.0 introduces advanced cryptographic flexibility and supports a range of algorithms, including SHA-1 for backward compatibility, SHA-256, RSA, ECC-256, HMAC, and AES-128, providing robust options for secure operations. This version organizes keys into three main hierarchies: platform, storage, and endorsement. Each hierarchy can support multiple keys and algorithms, enhancing security management and allowing for versatile key applications based on specific needs.

A significant feature of TPM 2.0 is policy-based authorization, enabling more sophisticated and adaptable access controls (in previous version just one password was required). This policy framework allows security measures to be tailored according to diverse requirements and provides a structured way to enforce access restrictions.

Additionally, TPM 2.0 includes platform-specific specifications tailored to various application areas, including PC clients, mobile devices, and automotive environments. Each of these specifications addresses the unique security requirements of its respective platform, ensuring that TPM 2.0 can meet the needs of a broad range of use cases. However, most of the hardware manufacturers implements their TPMs directly in the CPU or the chipset, meaning that one should check when buying new hardware: in some cases the TPM is software based on even virtualized on a dedicated VM thanks to the hypervisor.

Implementations of TPM 2.0

TPM 2.0 has several implementation forms, each designed to fit different hardware and software architectures. The Discrete TPM is a dedicated chip that implements TPM functionality within its own tamper-resistant semiconductor package, providing robust security.

In contrast, the Integrated TPM is part of another chip and is not required to implement tamper resistance. For example, Intel has integrated TPMs in some of its chipsets, balancing functionality with space and cost considerations.

The Firmware TPM is a software-only solution that operates within a CPU's trusted execution environment. Major manufacturers such as AMD, Intel, and Qualcomm have implemented firmware TPMs, allowing for flexibility in system design.

Another form is the Hypervisor TPM, which offers a virtual TPM managed by a hypervisor. This runs in an isolated execution environment and is comparable to a firmware TPM in terms of security and functionality.

Finally, the Software TPM serves as a software emulator of a TPM, primarily useful for development purposes. This allows developers to test and implement TPM functionalities without needing dedicated hardware.

TPM 2.0 Three Hierarchies

TPM 2.0 defines **three key hierarchies**, each serving distinct purposes and managing different aspects of security and key storage.

The first hierarchy is the **Platform Hierarchy**, which is dedicated to managing the platform's firmware. This hierarchy utilizes non-volatile (NV) storage for keys and data, ensuring that critical firmware-related information is securely maintained.

The second is the **Endorsement Hierarchy**, which primarily caters to the privacy administrator. But why we need a privacy admin? In the early days of the TPM and the Trusted Computing Group (TCG), there was concern over privacy implications. One issue was that every time a device's TPM measured and reported its state, it signed those measurements with its unique RSA private key. This approach ensured that the measurements were authentic and uniquely tied to the device, which was valuable for verifying the integrity of the platform. However, this method also had a drawback: it revealed the device's identity. If each attestation came from a unique RSA key, it could be traced back to the same machine each time, compromising privacy. For this reasons we use different keys for different purposes(key of the manufacturer, key to identify the owner, ...).

Similar to the Platform Hierarchy, this one also provides storage for keys and data, emphasizing the importance of privacy in the overall security architecture.

Lastly, the **Storage Hierarchy** is designed for the platform's owner, who typically also acts as the privacy administrator. This hierarchy features NV storage for keys and data, allowing for efficient management of cryptographic assets.

Each of these hierarchies comes with dedicated authorization mechanisms, such as passwords, and specific policies to govern access and usage. Additionally, each hierarchy utilizes a unique seed for generating the primary keys, further enhancing the security and integrity of the TPM's operations.

4.6.8 Using a TPM for Securely Storing Data

Utilizing a TPM for securely storing data involves several key considerations to ensure both security and accessibility.

One primary advantage of using a TPM is its physical isolation. The storage occurs within the TPM itself, specifically in NVRAM, which helps safeguard sensitive information. This storage is very small, so ut usually stores primary keys and permanent keys

To enhance security, Mandatory Access Control (MAC) mechanisms are employed, which govern how data and keys are accessed within the TPM. This cryptographic isolation ensures that even if data is stored outside the TPM, such as on a platform's hard drive, it remains protected.

When storing keys or data outside the TPM, it is crucial that the information is encapsulated in a secure format, often referred to as a blob. This blob must be protected, typically by encrypting it with a key controlled by the TPM. The use of MAC further reinforces the security measures, ensuring that access to the stored data is tightly regulated.

4.6.9 TPM Objects

TPMs utilize various objects to manage cryptographic keys and secure data. One of the primary types of objects within a TPM is the **primary keys**, which includes endorsement keys and storage keys. These keys are derived from one of the primary seeds stored within the TPM. Notably, the TPM does not return the private value of these keys; instead, they can be re-created using the same parameters, assuming that the primary seed remains unchanged.

In addition to primary keys, TPMs also handle keys and sealed data objects (SDOs). These objects are protected by a Storage Parent Key (SPK), which is necessary within the TPM to load or create a key or SDO. The randomness required for key generation and other cryptographic functions is provided by the TPM's built-in Random Number Generator (RNG).

When a key is generated, the TPM returns the private part, which is protected by the SPK. However, it is essential to note that this private part must be stored securely, as its integrity is critical to maintaining the overall security provided by the TPM.

4.6.10 TPM Object Areas

Trusted Platform Modules (TPMs) categorize the storage structure of objects into distinct areas, each serving a specific purpose:

- **Public Area:** This area is utilized to uniquely identify an object, providing essential information for the management of keys and data.
- **Private Area:** Containing the object's secrets, this area exists solely within the TPM. It is crucial for maintaining the confidentiality and integrity of the keys and data stored within the module.
- **Sensitive Area:** This consists of the encrypted private area, specifically designed for use when storing sensitive data outside of the TPM. It ensures that even when data is not housed within the TPM, it remains secure through encryption.

4.6.11 TPM Platform Configuration Register (PCR)

The Platform Configuration Register (PCR) serves as the TPM implementation of Root of Trust for Storage (RTS) and is a core mechanism for recording platform integrity. PCRs maintain their values and can only be reset during a platform reset or through a hardware signal, which ensures that any malicious code cannot manipulate or retract its measurements.

PCRs are extended using a cumulative hash, following the formula:

$$\mathrm{PCR}_{\mathrm{new}} = \mathrm{hash}(\mathrm{PCR}_{\mathrm{old}} \,||\, \mathrm{digest\_of\_new\_data})$$

Basically, the new PCR value is the hash of the old PCR value and the new data. This process is repeated for each new piece of data that needs to be added to the PCR.

This process is commonly referred to as the EXTEND operation (keep in mind that it's not commutative). Additionally, PCRs can be utilized to gate access to other TPM objects. For example, BitLocker uses PCR values to seal disk encryption keys, ensuring that the keys can only be accessed when the platform is in a known and trusted state.

4.6.12 Measured Boot

Measured boot builds on secure and trusted boot principles by using TPM functionality to verify system integrity throughout the boot process. The key here is that a TPM is available—whether as a separate chip or firmware module—enabling attestation at each step.

The process begins with the Core Root of Trust for Measurement (CRTM) located in the boot ROM's first-stage bootloader. This trusted component takes the first measurement by capturing the state of the next component in line, the second-stage bootloader, and stores this in the TPM's Platform Configuration Registers (PCRs). This baseline measurement is critical, as it establishes initial trustworthiness for everything that comes afterward.

Once the second-stage bootloader is loaded, it continues the process by measuring the operating system, which can also measure subsequent applications if needed. The idea is simple: measure each part of the boot sequence and store these measurements in the PCRs, creating a chain of trust from the very start.

The accumulated values in the PCRs provide a snapshot of the system's integrity, which an external verifier can check to confirm everything is as expected. Though an internal verifier is possible, an external one is often more reliable in case an attacker has compromised the internal components. This setup allows measured boot to act as a strong line of defense, verifying each stage and ensuring a trusted environment.

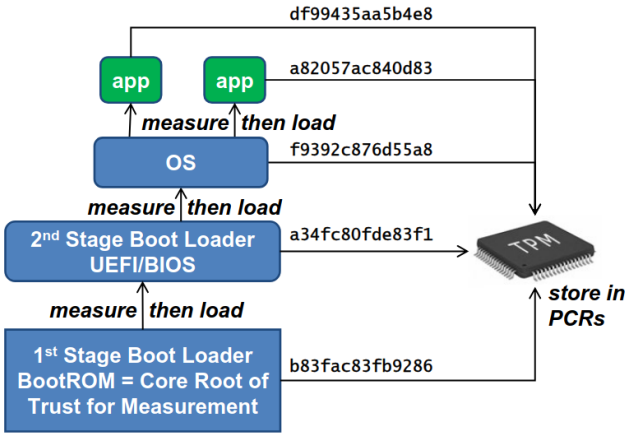


Figure 4.4: Measured boot

4.6.13 Remote attestation procedure

Remote attestation leverages the values stored in the TPM's PCRs during the measured boot process, enabling an external verifier to confirm the system's integrity. Here's how it

works:

First, an external verifier, which could be a trusted system on the network, sends a unique challenge (often a nonce) to the device holding the TPM. This challenge prevents replay attacks by ensuring that each attestation session is fresh and unpredictable. The platform then gathers the values of the PCRs requested by the verifier and packages them along with the challenge.

This package of data—containing the PCR values and the response to the nonce—is signed using a key unique to the device. This signed response assures the verifier that it's seeing legitimate measurements tied to that specific device.

When the verifier receives the signed package, it performs several checks:

1. **Signature Validation**: It verifies the signature to ensure the data hasn't been tampered with and is genuinely from the device.

2. **Identity Validation**: It checks that the device's identity matches expected values, using a stored list of authorized machines to confirm that the device is part of the network.

3. **Measurement Validation**: Finally, the verifier compares the PCR values against known "golden values" stored in a database. These golden values reflect the expected configurations for each device type—taking into account variations in hardware, firmware, and software.

If the PCR values match the golden values, the verifier can confidently conclude that the system is in a known and trusted state. If there's a mismatch, an alert is raised, signaling that the device may have been compromised or misconfigured.

This remote attestation process provides a robust way to ensure system integrity across a network by verifying both the software status and identity of each device, making it a valuable tool for network security.

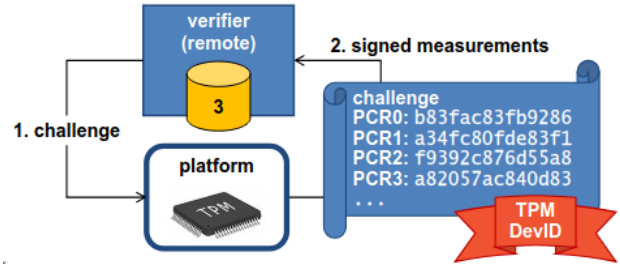


Figure 4.5: Remote attestation procedure

4.6.14 Management of Remote Attestation

Remote attestation has been performed, and it has failed. What now? Management of remote attestation is most important when the attestation process fails.

One critical aspect is whether to implement only boot attestation, which is static, or to include periodic (dynamic) attestation as well. This decision should take into account the attack model, especially with respect to runtime vulnerabilities that could be exploited.

The **periodicity of the attestation** operation is another important consideration, as it must align with the speed of potential attacks. A cycle of operations which include the time required for signature verification, protocol execution, and database lookups, are typically in the range of several seconds due to the inherent slowness of TPMs, and for some cases is acceptable, but for others a window of exposure of even 5 seconds is excessive. And that is a limit, not of the attestation, but of the physical implementation.

Another challenge in remote attestation management is **whitelist generation**. This process can be complex in general but may be less difficult in limited environments, such as Internet of Things (IoT) devices, edge devices, Software-Defined Networking (SDN), and Network Function Virtualization (NFV).

Furthermore, it is essential to label components appropriately—such as good, old, buggy, or vulnerable—and to include configurations from sources like Management and Orchestration (MANO) or network management tools. While generating labels can be straightforward if the data is file-based, it becomes significantly more challenging when dealing with memory-based data.

4.6.15 TCG PC Client PCR use (architecture)

According to the Trusted Computing Group for a PC client the PCRs are used for various purposes:

- PCR0 is measuring the Platform firmware from system board ROM and it is a fixed value given one version of the firmware. Typically, in a database there are different values for PCR0 according to the version of the firmware and the version running in a device could be deduced by these values. It also stores the UEFI boot services, UEFI runtime services.
- PCR1 contains the hash of various extensions of the firmware (ACPI, SMBIOS, OTHER).
- PCR2 drivers loaded from the disk.
- PCR3 is not here. That means it's not used for PC.
- PCR4 is the part of the UEFI OS loader and of the Legacy OS Loader
- PCR5 is for the Platform hardware, for example it is reading and computing the hash of the partition table.

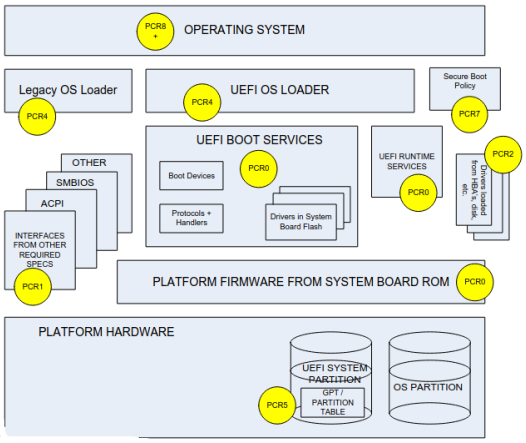


Figure 4.6: TCG PC Client PCR architecture

That is important if someone manipulated the hardware.

- PCR7 contains the Policy for Secure Boot.
- All registers from PCR8 to above are given to the OS to decide what will be used for.

For all the registers up to PCR7 the values can be predicted. They will depend on the kind of platform, version of the firmware or driver. These values are in the golden: if there are wrong values in those registers the system cannot be trusted.

PCR Index	**PCR Usage**
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and PI Drivers
1	Host Platform Configuration
2	UEFI driver and application Code
3	UEFI driver and application Configuration and Data
4	UEFI Boot Manager Code (usually the MBR) and Boot Attempts
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code) and GPT/Partition Table
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8-15	Defined for use by the Static OS
16	Debug
23	Application Support

Table 4.1: PCR Index and Usage

4.6.16 Measured Execution

Previously, we discussed measured boot, which verifies the integrity of the system during the boot process. We would like to extend this functionality to the application that runs on the OS, but to do so we would have to store the measurements in PCRs, which are limited in number.For this reason we can simply use one PCR and extend the measurements of each application.

Another issue also arises: the value of the PCR depends on the execution order, in fact starting an application A before an application B will result in a different value of the PCR than starting the application B before the application A. For this reason, the OS can measure the application before loading it and perform the extend operation of the PCR, but not all OS are able to do so(the superior OS can).

Linux's IMA

The **Integrity Measurement Architecture** (IMA) in Linux is a standard component of the Linux kernel(one needs to enable it) that extends attestation capabilities to dynamically

executed elements, such as applications. It provides mechanisms for collecting, storing, appraising, and protecting measurements to ensure the integrity of files accessed on the system. It can perform various operations:

- **Collection**: IMA measures a file before it is accessed, creating an integrity measurement to track any potential alterations.
- **Store**: The measured data is added to a kernel-resident list, known as the Measurement List (ML), and the IMA PCR (specifically PCR 10) is extended to reflect these measurements, making them available for attestation purposes.
- **Appraise** (optional): IMA can enforce local validation of a file's measurement by comparing it to a "good" value stored in the file's extended attributes. This step helps to ensure that the file remains in a trusted state, but is a feature which is not required for measured operations
- **Protect** (optional): IMA can secure a file's extended security attributes, including the appraisal hash, against offline attacks. This prevents unauthorized modifications to these attributes when the system is not active.

The appraise feature is most interesting: lest's suppose that one think that, for example, a python executable has been compromised. One can use an extended attribute to store the hash of the python executable and compare it with the hash of the python executable measured by the IMA. This happends before it's been executed, automatically by the OS. If the hashes are different, the python executable is not executed.

When the IMA starts, it extends UEFI's measured boot concept to the OS and applications by using PCR 10. It begins by storing a "boot aggregate," which is a hash of all PCR values from 0 to 7—these PCRs capture UEFI-related measurements. This boot aggregate establishes a continuity between the boot measurements and IMA's ongoing monitoring, preventing certain types of attacks.

IMA is configurable through an "IMA template," which defines what gets measured. A common template, "IMA-NG," is often used, but the template can be customized to fit specific needs. IMA then logs these measurements in the kernel security filesystem, where they're stored as ASCII runtime measurements, allowing access to a detailed record of measured files and executables. For example, each entry includes the PCR 10 value, a template hash, and hashes for each file, showing the order in which each executable or library was measured.

PCR	**template-hash**	**template**	**filedata-hash**	**filename-hint**
10	91f34b5[...]ab1e127	ima-ng	sha1:1801e1b[...]4eaf6b3	boot_aggregate
10	8f16832[...]e86486a	ima-ng	sha256:efdd249[...]b689954	/init
10	ed893b1[...]e71e4af	ima-ng	sha256:1fd312a[...]6a6a524	/usr/lib64/ld-2.16.so
10	9051e8e[...]4ca432b	ima-ng	sha256:3d35533[...]efd84b8	/etc/ld.so.cache

Table 4.2: Example of IMA Measurement List

To verify these measurements, a verifier can request the current PCR 10 value. The machine responds with this value along with the detailed measurement list. To validate, the verifier starts with a zero value, incorporates the boot aggregate by combining PCRs 0

```
myPCR10 = 0
myPCR10 = extend(boot_aggregate)

foreach measure M of component C
    if (C not authorized) then raise alarm
    if (M != gold_measure(C)) then raise alarm
    myPCR10 = extend(M)
end foreach

if (myPCR10 == PCR10) OK else raise alarm
```

Listing 2: IMA verification

through 7, and then processes each measurement in the order they were originally extended into PCR 10. This step-by-step verification ensures that the system's state aligns with expected integrity values.