ADVANCED INFORMATION SYSTEM SECURITY

Hoping to get a better grade this time around.

Fabio Lorenzato

November 24, 2024

Contents

Chapter 1

Transport Layer Security

TLS, or Transport Layer Security, was originally proposed by Netscape 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 always 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 transmitted 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.

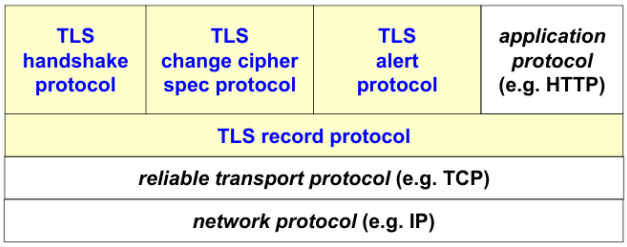


Figure 1.1: TLS packet structure.

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).

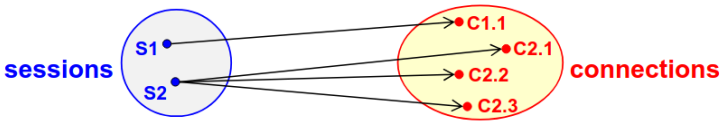


Figure 1.2: TLS session and connection.

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 achieved in TLS using **authenticate-then-encrypt** approach, 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)

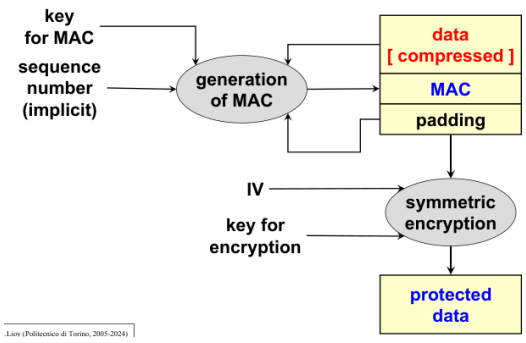


Figure 1.3: TLS data protection.

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.

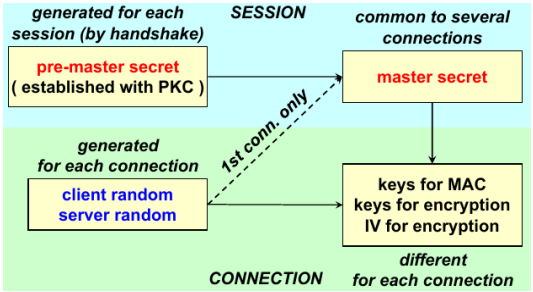


Figure 1.4: Relationship among keys and sessions.

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.

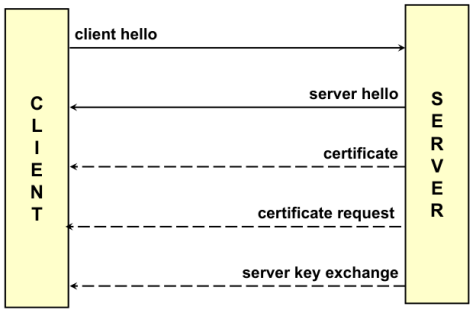
1.6.1 Client Hello and Server Hello

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

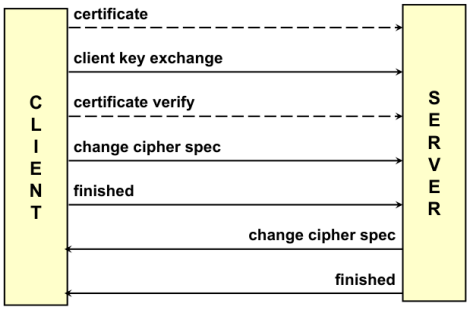
- the SSL version preferred by the client, and the highest supported(2=SSL-2, 3.0=SSL-3, 3.1=TLS-1.0, ...)
- a 28 bytes pseudo-random number, which is the client random
- a session-id, which is *0* if the client is starting a new session, and different from it if the client is trying to resume a previous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

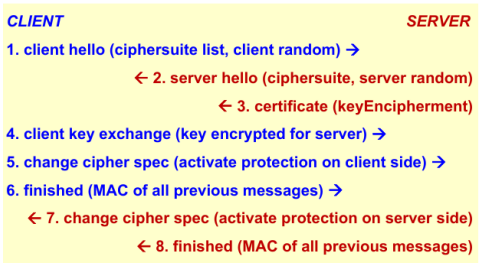
1.6.6 Change cipher spec

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.

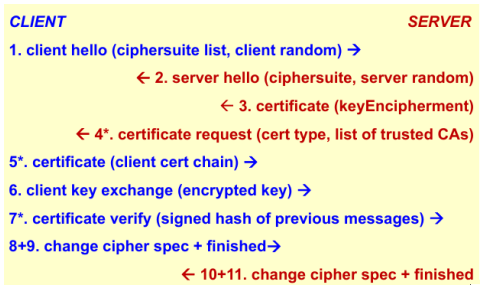
1.6.7 Finished message

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.

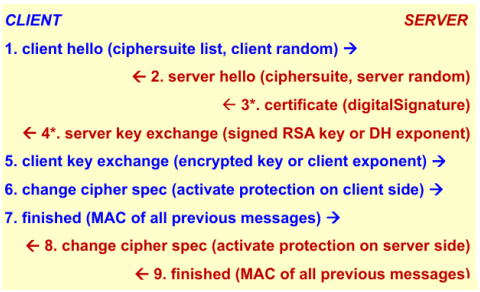
This allows to prevent rollback man-in-the-middle attacks (version downgrade or ciphersuite downgrade)



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



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



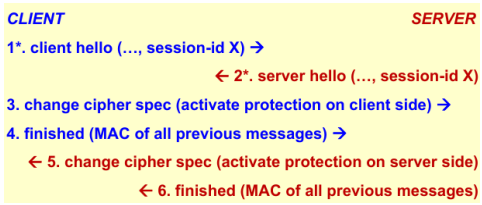
(c) TLS handshake(ephemeral, no client authN).

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 that 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.

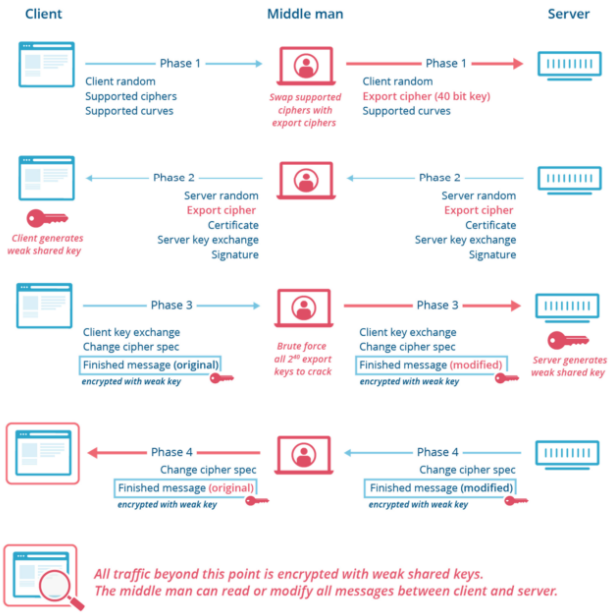


Figure 1.8: FREAK attack.

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).

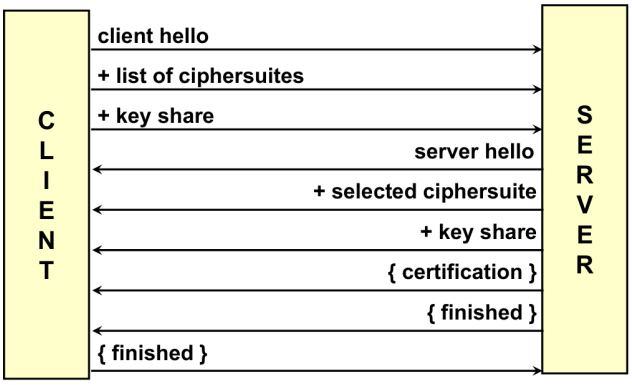


Figure 1.9: TLS 1.3 handshake.

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.

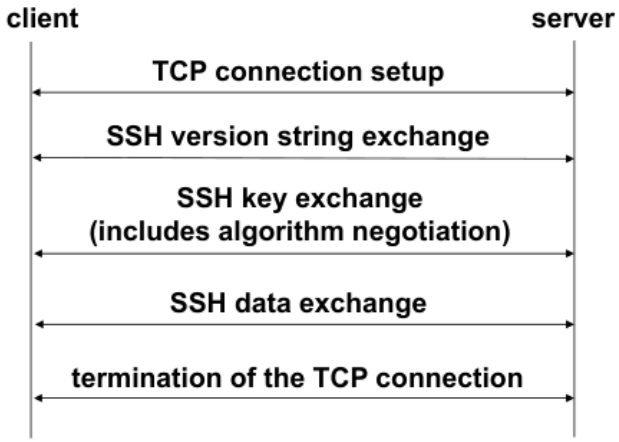


Figure 2.1: SSH Transport Layer Protocol

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)

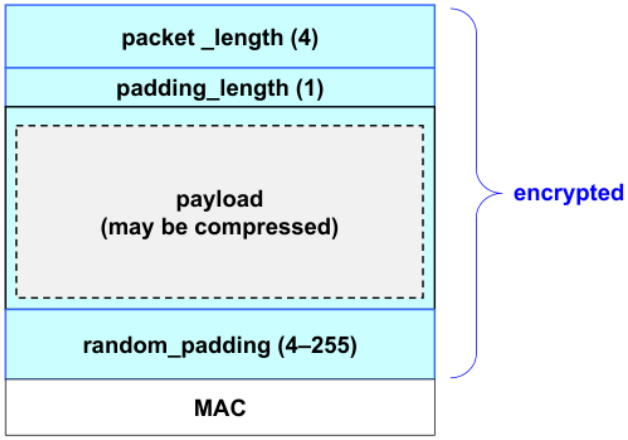


Figure 2.2: Binary Packet Protocol

- 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**

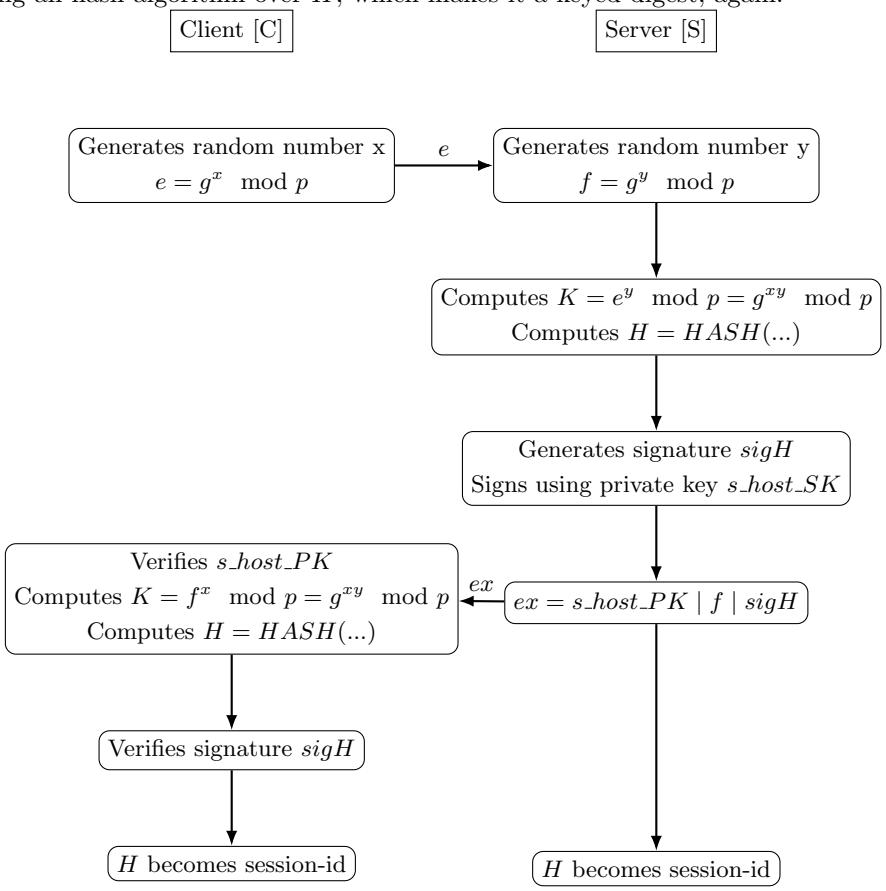
Diffie-Hellman key agreement

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.

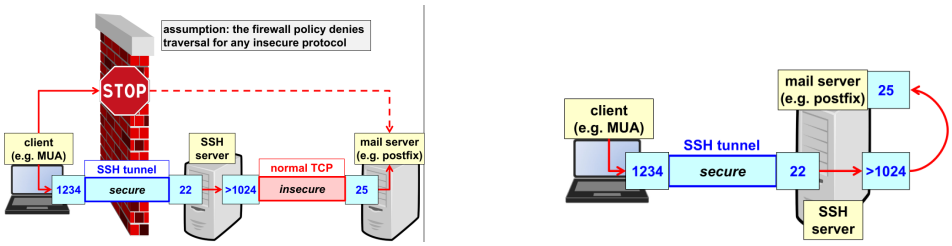
2.2.1 Local port forwarding

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.

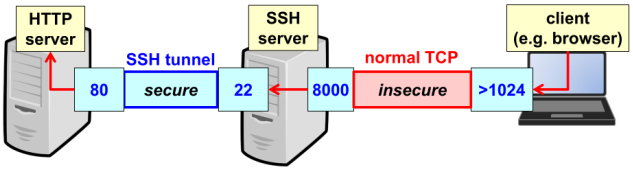


2.2.2 Remote port forwarding

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.

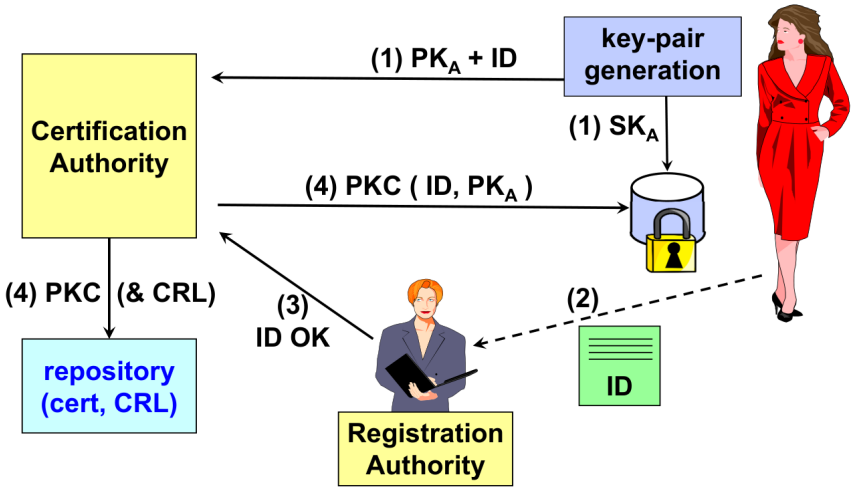


Figure 3.1: Certificate generation

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.

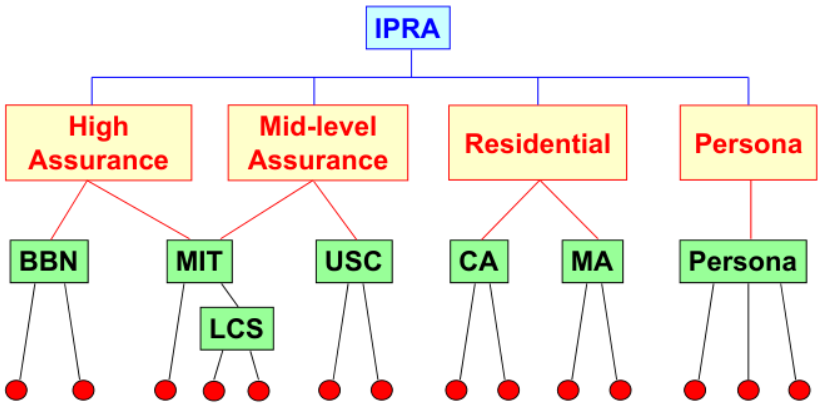


Figure 3.2: Internet PEM hierarchy (RFC-1422)

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.

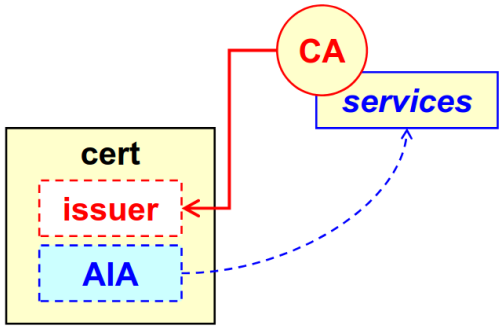


Figure 3.3: Authority Information Access (AIA) extension

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.

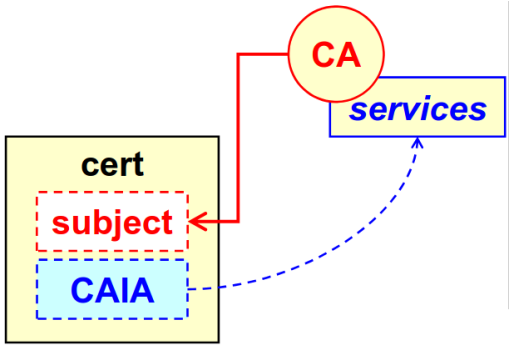


Figure 3.4: CA Information Access (CAIA) extension

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 3.5, 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.

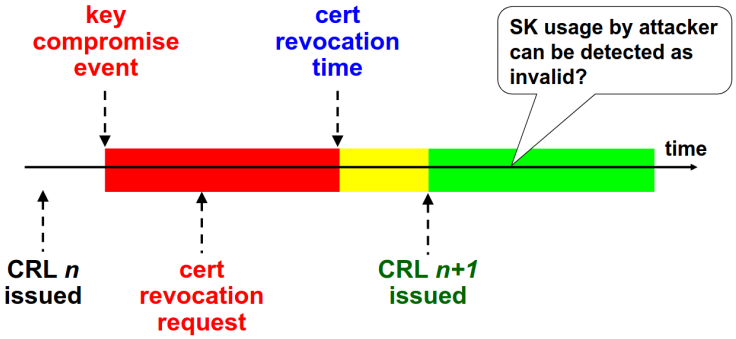


Figure 3.5: Certificate revocation timeline

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)**

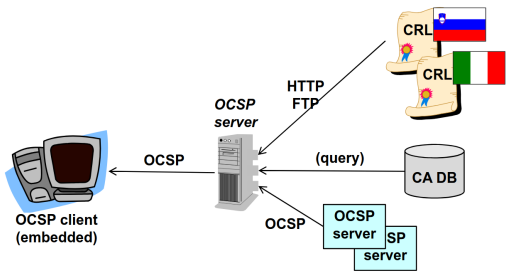


Figure 3.6: OCSP source of information

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.

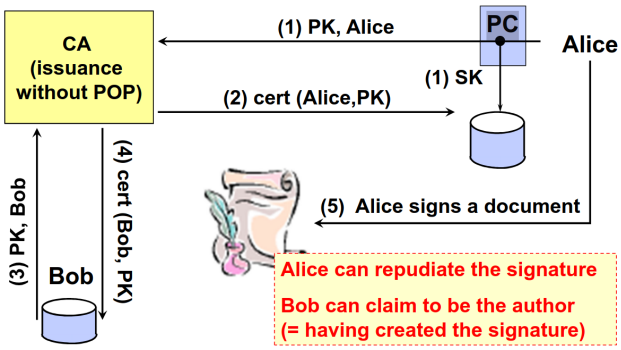


Figure 3.7: Risks in the absence of POP

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 3.8. 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.

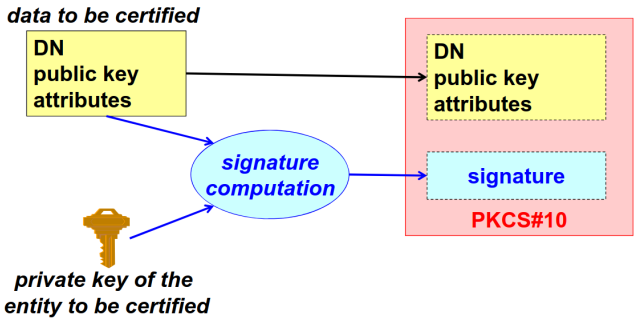


Figure 3.8: PKCS#10 structure

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).

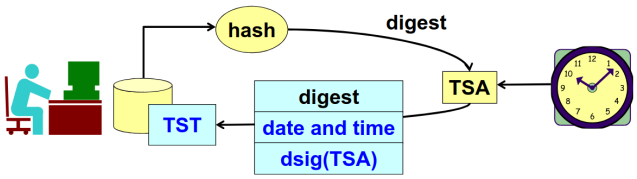


Figure 3.9: the time-stamping process

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.

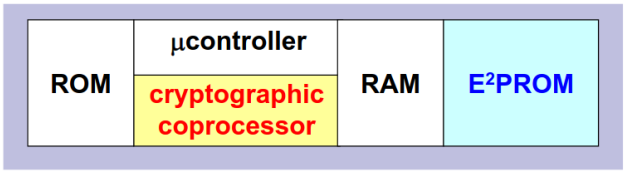


Figure 3.10: Cryptographic smart-card

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.

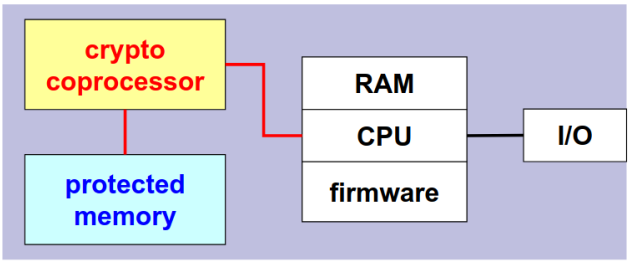


Figure 3.11: Hardware Security Module (HSM)

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.

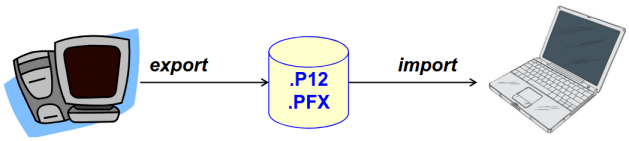


Figure 3.12: PKCS#12

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.

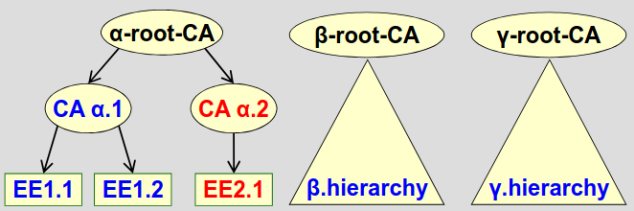


Figure 3.13: Hierarchical PKI

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.

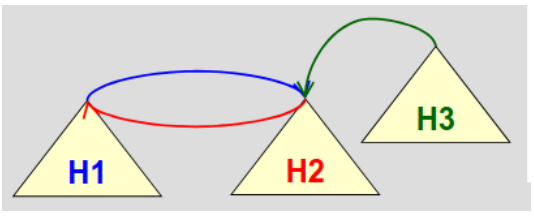


Figure 3.14: Mesh PKI

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/`.

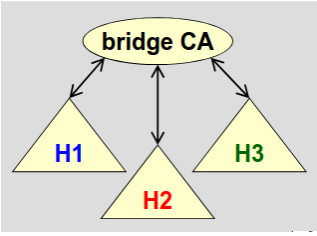


Figure 3.15: Bridge PKI

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.

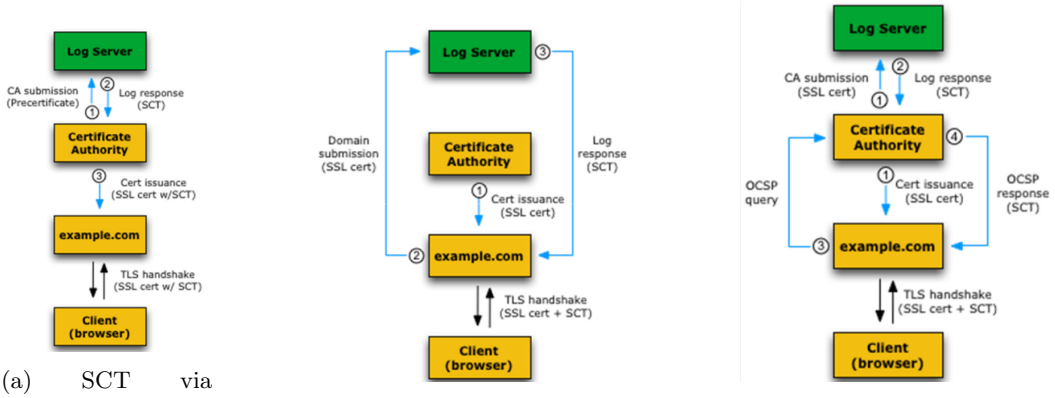
SCT via TLS extension

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:

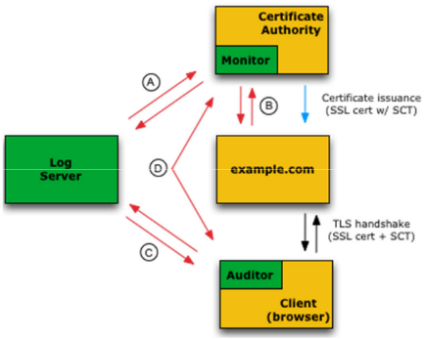
(A) Monitors continuously observe logs for suspicious certificates and verify that all logged certificates are publicly visible.

(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 3.17 , 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.

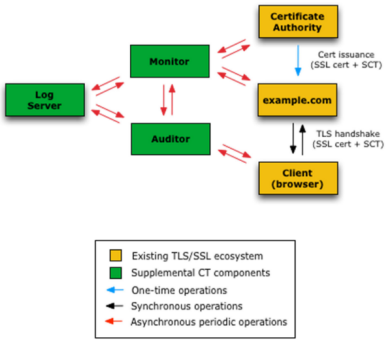


Figure 3.17: CT System Configuration

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`.

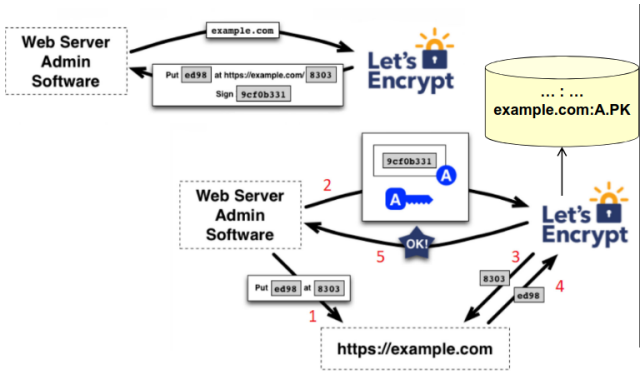


Figure 3.18: ACME Domain Validation

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.

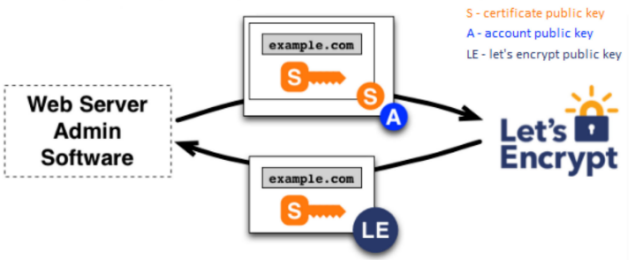


Figure 3.19: ACME Certificate Request and Issuing

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.

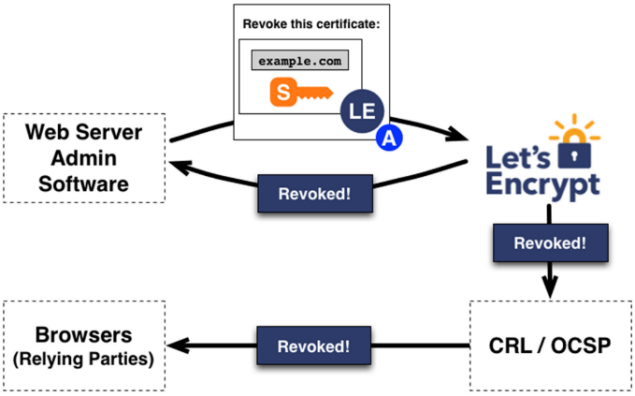


Figure 3.20: ACME Certificate Revocation

Chapter 4

Trusted computing

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

For instance, 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 are **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 or 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.

> A **Trusted Execution Environment** is what one 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.

As you can see from figure 4.1, one one device can run different environments at the same time. One 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 hardware 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

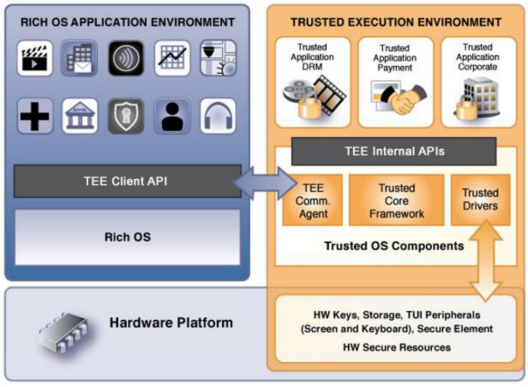


Figure 4.1: Trusted Execution Environment and Rich Execution Environment

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 4.2. 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.

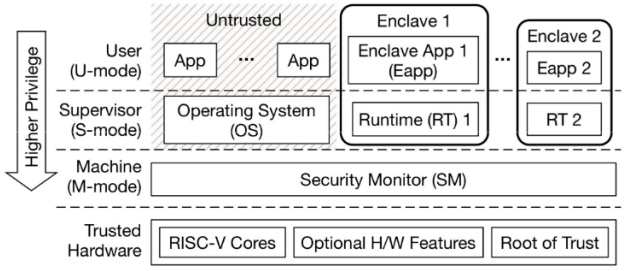


Figure 4.2: Keystone architecture

> 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 4.3, 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.

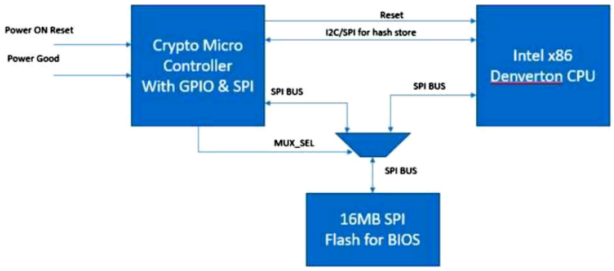


Figure 4.3: Hardware Root of Trust verification

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 4.4. Notice that the trusted drivers and the anti-malware are loaded before the OS.

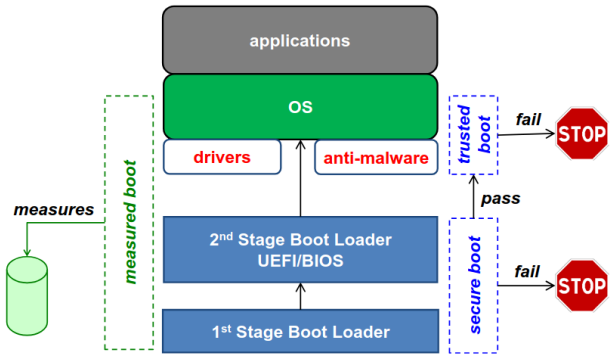


Figure 4.4: Boot types

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 4.5 . 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.

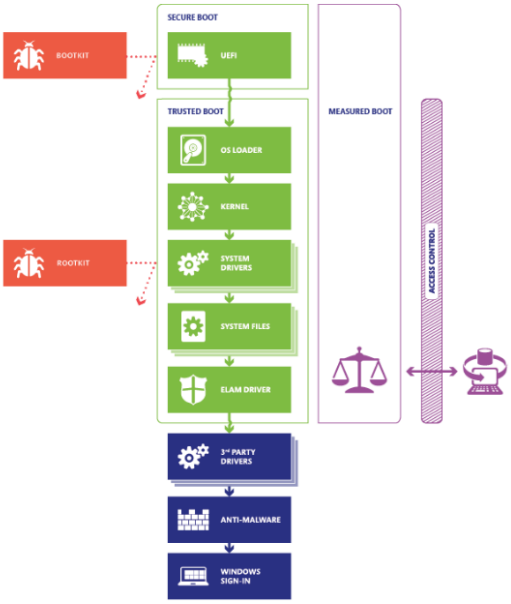


Figure 4.5: Windows Boot protection

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

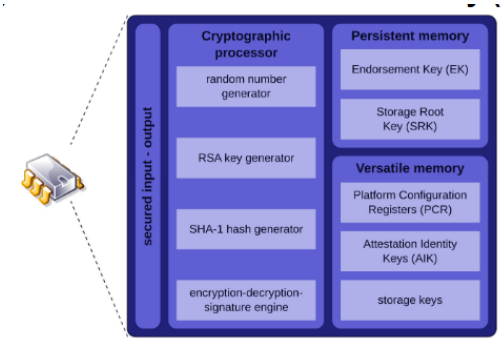
TPM 1.2 features a fixed set of cryptographic algorithms, which include SHA-1, RSA, and optionally AES. This version of the TPM provides a single storage hierarchy specifically for the platform user, ensuring that key management and storage are centralized.

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:

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

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.

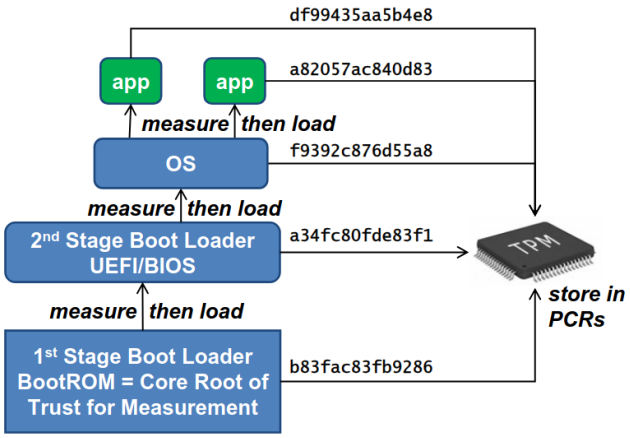


Figure 4.6: Measured boot

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.

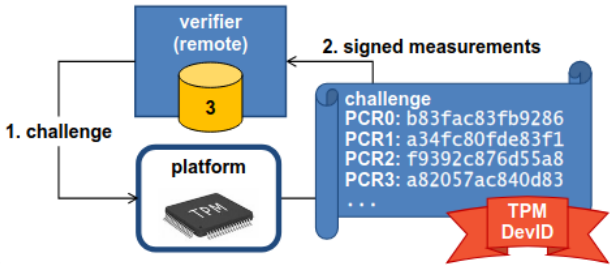


Figure 4.7: Remote attestation procedure

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.

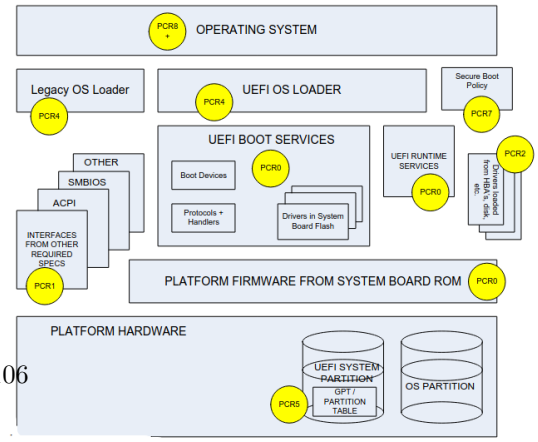
Another challenge in remote attestation management is **whitelist generation**. This process can be complex in general but may be less difficult in limited environments, such as Internet of Things (IoT) devices, edge devices, Software-Defined Networking (SDN), and Network Function Virtualization (NFV).

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. 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

```
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

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 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.

4.6.17 Size and Variability of the TCB

The TCB is as large as the smallest set of code, hardware, personnel, and processes necessary to be trusted to meet specific security requirements. A reliable TCB is essential for achieving a secure system, as it represents the foundation upon which all security assurances are built.

To enhance confidence in the TCB, several methods can be applied:

- **Static Verification:** Analyzing the TCB without executing it to ensure it behaves as expected.
- **Code Inspection:** Manually examining the TCB code for potential vulnerabilities.
- **Testing:** Running the TCB through various scenarios to check for security flaws.
- **Formal Methods:** Applying mathematical proofs and models to verify the TCB's behavior under different conditions.

These methods, however, are often expensive and labor-intensive, making it essential to minimize the complexity of the TCB.

> A smaller, more streamlined TCB reduces the attack surface, yet simplification alone is insufficient if variability is not controlled.

The TPM aims to establish a reliable TCB via the CRTM. However, the TCB has become increasingly large and dynamic, making consistency more difficult to maintain. . For example, two "identical" computational nodes might report different TCB measurements due to slight variances, but be completely fine.

4.6.18 Dynamic Root of Trust for Measurement

The solution to the variability and complexity of the TCB is the Dynamic Root of Trust for Measurement (DRTM).

It is a technique that establishes a trusted environment on demand by resetting the CPU and starting measurements from a controlled point, rather than relying on the entire system boot sequence from BIOS onward. After all, the secure boot can't be tampered with, so we can start the measurements after that.

For that purpose, since TPM version 1.2, some dynamic PCRs were added from 17 to 23. Those are set to -1 at boot(the others are usually set to 0) and can be reset to 0 by the OS(this operation is not allowed for the other PCRs, they can be only reset by hardware resets).

PCR 17 is unique in that it is used specifically by DRTM commands, ensuring the integrity of this late-launch measurement. By allowing the TCB to start fresh from a known, controlled point, DRTM enables flexible and secure initialization of trusted environments.

In any case, whatever is the real operation being executed, they have the same effect. They disable direct memory access, disable all the interrupts and disable any debugging mode. That means that the program executing that special instruction is the only thing being executed in this moment on this computer, cannot be interrupted, no one else can have access to the memory and no one else can trace it or modify it via debug. The purpose is to permit to this program to measure and then execute the secure loader block.

DRTM leverages special processor commands to initiate a secure environment. These commands include:

- `SENTER` for Intel's Trusted Execution Technology (TXT), which executes the SINIT binary module.
- `SKINIT` for AMD's Secure Virtual Machine (SVM).

When either of these commands is executed, all processing on the platform is temporarily halted. DRTM then:

- Hashes the contents of a specified memory region, storing this measurement in a dynamic PCR.
- Transfers control to a specific memory location, where the Secure Loader Block (SLB) is measured and executed. this is also called **Late Launch** because it's not the original

launch of the system, but it's something that happens after the normal launch. This helps in avoiding the problem with the PCR values that are incorrect when firmware is updated. And the consequent problem with the sealed data.

- Disables Direct Memory Access (DMA), interrupts, and debugging to secure the environment during this process.

One last thing to help understand why firmware updates are a problem for sealing. We have mentioned the fact that sealing is like encryption, but it is attached, it's related to a specific system state. So imagine that you have sealed data. Then your system say, okay, I'm performing an update, firmware update. Now you've got a new bias. Next time that you start your system, even if that bias is correct, so the measurements are perfect, the sealing will not work. Because they all, if you want to unseal, you must have that version because of that was the state. So it means that now we should seal the data, not against the firmware, but upon some operational state started after the dynamic route of trust for measurement was executed. Okay, so it's not only a problem of not having the correct measures because creating a very big database, it is of course possible to maintain a lot of measurements. But sealing is one of the worst thing because sealing requires exactly the same state.

4.6.19 Hypervisor TEE

The DRTM was designed to facilitate the secure loading of a hypervisor, such as Xen or VMware ESX, which in turn manages and isolates VMs. When initialized through DRTM, the TPM can attest to the integrity of the hypervisor, ensuring it has loaded correctly and securely. Only after a proper load can TPM-sealed storage be accessed by the hypervisor, thus providing a robust mechanism for securing sensitive data in cloud computing environments.

Although this approach enhances security, validating the hypervisor's integrity remains challenging due to its substantial codebase. For instance:

- Xen hypervisor includes a complete copy of the Linux kernel.
- VMware hypervisors are similarly large in scale.

The extensive code size of these hypervisors necessitates rigorous validation to maintain trust, underscoring both the capabilities and challenges of using DRTM for hypervisor-based TEEs.

4.6.20 Remote Attestation in Virtualized Environments

In virtualized environments, having a hardware RoT remains crucial for ensuring security. However, full virtualization, as seen in VMs, often provides only a software-based RoT, such as virtual TPMs (vTPMs) implemented by platforms like Xen, Google, and VMware. To strengthen security, a strong link between the virtual TPM (vTPM) and the physical TPM (pTPM) is necessary.

This approach involves *deep attestation*, which is basically a validation of the vTPM by the pTPM. Furthermore, by rooting sealed objects in the physical TPM, the vTPM gains added protection. This setup requires an extension of the standard TCG-defined interfaces, which is an area of ongoing development.

As a result in cloud environments, one should investigate if they have really access to a pTPM, because "equivalents" are not really the same.

Alternatively, in lightweight virtualization environments, such as Docker containers, a different approach is feasible. Here, because the hardware resources, including the TPM, are shared directly with the host (the virtualization layer is not completely separated, they use namespaces after all), the complexity of managing separate attestation layers can be reduced.

4.6.21 Remote Attestation for OCI Containers

Remote Attestation for Open Container Initiative (OCI) containers is designed to be flexible and is not dependent on any specific containerization technology. This approach is transparent both to the container runtime and to the containerized workloads, allowing seamless operation without requiring modifications to the container environment.

The RA process verifies both the host and its containers, leveraging a hardware-based RoT to ensure the integrity of the system components.

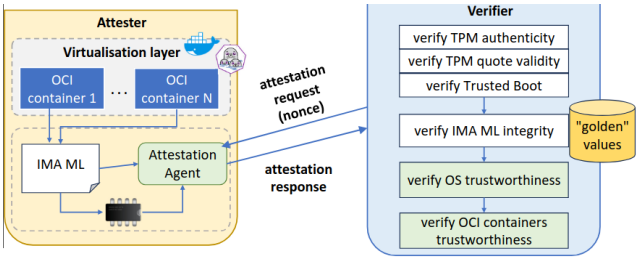


Figure 4.9: Remote attestation for OCI containers

This process works seamlessly with the container runtime and workloads, enabling remote attestation of both the host and containers. Attesting the host is essential since it's the base layer; if compromised, the integrity of all containers would be at risk. Once the host is verified, we can choose to attest containers individually or in specific groups, depending on our needs.

The architecture includes a hardware root of trust, an attestation agent, and a virtualization layer tailored for container environments. When any operation is measured, it's labeled to indicate where it was executed—either on the host or within a specific container. This labeling ensures clarity, as operations that appear to be running inside containers are actually executed on the host OS.

During attestation, the verifier sends a request containing a nonce, and the platform responds with the measurement list, nonce, PCR values, and a TPM-signed response. The process first involves checking that the TPM's digital signature matches the expected node certificate. Then, we verify the TPM quote to confirm that the signature aligns with the PCR values and that the PCRs tied to trusted boot contain the expected values. Following this, we assess the IMA measurement list for both the host and containers, comparing these against known good values, or "golden" values.

Implementation

The implementation of RA for OCI containers extends IMA template called `ima-dep-cgn`. This template is designed to enhance the attestation process by addressing the following key aspects:

- **Dependencies:** The template helps to determine whether an entry belongs to the host or to a specific container, establishing a clear relationship between the two.
- **Control-Group Name:** This identifier specifies the particular container associated with the entry, allowing for accurate tracking and management.
- **Template Hash:** The digest for the entry is calculated using an algorithm other than SHA-1, with options such as SHA-256 or SHA-512, ensuring a more secure and robust hashing process.

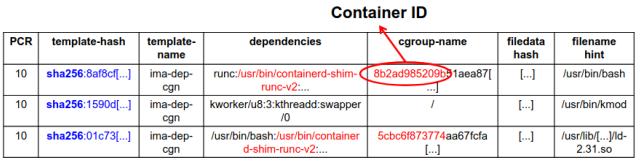


Figure 4.10: Implementation of remote attestation for OCI containers

4.6.22 Credentials chain of trust

Another main issue is privacy: in fact, the TPM is signing the measurements with its own private key, and if you move that machine to different environments you will be able to trace that it is always the same machine because the signature will be created always with the same public key. So we would like to create a chain of credentials that are trusted but that protect privacy.

In the end we would like to move from TPM vendor issued keys, which are traceable, to a customer-usable certificate. There's also a specific standard for this, IEEE 802.1AR, which is a standard for securing a device identity using a TPM, whicle also allowing zero touch management of a platform, but in order to do that you need the identity of that element, a trusted identity

Inside the TPM, there's a Endorsement Key (EK) which is unique to each TPM, with the respestive EK certificate issued by the TPM vendor(this is also a proof that the component is genuine). We dont use that to avoid tracing by the vendor. Instead, because the TPM is usually enbedded on another device, the manifacturer additionally provides a IDevID based on the EK and the TPM vendor certificate, which, top maintain chain of trust, is linked to the TPM vendor one and cannot be moved to another machine. This IDevID is still inside the TMP, but we dont use that either to avoid tracing by the vendor. Instead, in the moment in which you deploy this machine inside your IT infrastructure another key pair whose certificate depends on the IDevID and the OEM certificate is created. This is the local device identifier (LDevID) which is based on the IDevID and the OEM certificate, and one can use this key to perform operations without being traced by the vendor, while still demostrating that it's genuine.

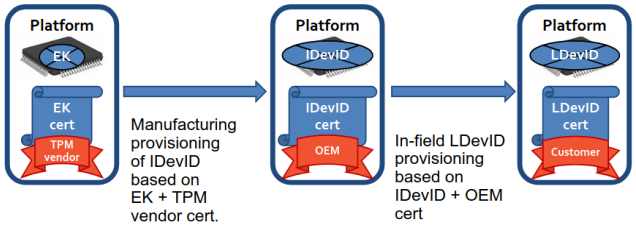


Figure 4.11: Credentials chain of trust

Let's go through how the sequence for creating or activating a credential works. We have three main parts: the TPM itself, the host platform that contains the TPM, and an external certification authority. The process begins with the host platform asking the TPM to create a new key. This applies to any keys created after the endorsement key, like for IDevIDs or LDevIDs. The TPM responds with the new key, providing only the public part.

Next, the host platform requests a certificate for this new key, and to prove its legitimacy, it includes its TPM endorsement credential. This credential demonstrates that the key request is coming from a trusted machine. The certification authority then provides a certificate for the new key, encrypting it with the TPM's endorsement key. This encryption serves as proof of possession since only the TPM that holds the endorsement key can decrypt it.

The host platform then instructs the TPM to decrypt the certificate and verify that it indeed certifies the new key. Once decrypted, the certificate is confirmed to be tied to the same TPM that holds the endorsement key, establishing a genuine pairing. Now, we have a new key that's verified and securely linked to the same TPM, ensuring its authenticity and security.

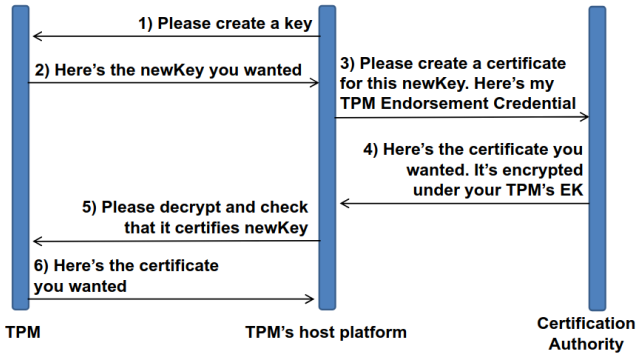


Figure 4.12: The TPM credential activation process

TPM Basic Authorization Mechanism

The TPM implements a basic authorization mechanism designed to ensure secure command execution and object usage.

The mechanisms available are:

- **Direct Password-Based Authorization:** The TPM supports direct password-based authorization for executing single commands, enhancing security by requiring valid credentials.

- **Password-Based HMAC:** Commands and responses are authenticated using a password-based Hash-based Message Authentication Code (HMAC), ensuring the integrity and authenticity of the communication. The mechanism utilizes both caller_nonce and TPM_nonce values to prevent replay attacks, ensuring that each command and response pair is unique and cannot be reused maliciously.

But the TPM is aware of various platform states and can be configured to enforce specific policies regarding object usage, including:

- **PCR Value Dependency:** Object usage can be restricted based on the values of selected Platform Configuration Registers (PCRs), ensuring that operations are only performed when the platform is in a trusted state.
- **Time-Based Restrictions:** The TPM can prevent object usage after a specified time, enhancing security by limiting the validity of operations to a defined time window.
- **Multi-Entity Authorization:** Usage of certain objects may require authorization from multiple entities, such as key holders, ensuring that a consensus is reached before critical operations are performed.

4.6.23 Trust perimeter and attestation considerations

Now that we have a functioning TPM and certificates for various keys, there's a critical issue around trust that we need to address, specifically concerning when and how to perform attestation. Many market solutions perform attestation when a software component is installed. If the component's hash doesn't match the expected value, the installation (or in a cloud context, the download) is blocked. At this stage, the signature from the developer is often verified. Additionally, attestation may be conducted at load time, when the component is loaded into memory for execution. This approach ensures that components are measured at runtime.

However, the concept of "executable" is nuanced. For compiled binaries, attestation is straightforward: we can measure the binary's hash. But for interpreted languages like Python, the executable is actually the interpreter, while the script itself becomes an input to the interpreter, creating challenges for accurate measurement. In these cases, measurements must extend beyond the interpreter to the script or input itself, if it exists as a file.

Moreover, even if a verified binary is running, its behavior might be influenced by configuration files. For instance, an application like IP tables reads configurations from files that can be modified during runtime. Therefore, it's crucial to monitor configuration files in addition to binaries. Generally, attestation systems like IMA focus on the file name for efficiency, but this can lead to issues if the file contents change without a name change. To account for modifications, attributes like the last modification date should also be considered.

The challenge intensifies with applications that store configurations in memory rather than on disk, as is common with routers, switches, and some firewalls. These devices are often configured through protocols like SNMP or netconf, which send UDP packets containing configuration data. Such configurations reside in memory, making disk-based attestation approaches ineffective. Hewlett-Packard, for instance, has implemented custom firmware for certain routers that can attest to specific memory regions where routing and filtering tables

reside. This approach allows for memory-based attestation by hashing designated memory segments, but it requires that these memory addresses remain constant to avoid issues like memory randomization.

For attestation to be complete, measurements need to be compared against "golden" or expected values. This becomes complex for in-memory configurations, as they vary over time. Here, the network or security management system becomes essential for providing the current expected values. Such integration allows the attestation system to verify in-memory configurations dynamically, rather than relying on static, file-based measurements.

This approach to attestation is increasingly critical for audit and forensic purposes. When an incident occurs, attestation provides a way to verify the system's state at the time, allowing us to determine whether an incident was caused by a design flaw or a security breach. This capability is essential in areas like autonomous vehicles, where accidents might involve significant legal or ethical implications. By attesting to the platform state, network path, and specific functions at any given time, attestation can support not only regular audits but also forensic investigations, helping identify the root cause of failures and assigning accountability.

In addition to software attestation, there's an emerging field called network path attestation. This aims to verify the exact path network packets took across routers, particularly in sensitive international contexts. Ensuring that packets didn't pass through unauthorized regions, such as certain countries, could be essential in security-sensitive scenarios. This opens new avenues for research and potential thesis topics, particularly for those interested in enhancing trust and accountability in modern, distributed networks.

In summary, attestation serves as a foundation for verifying what was executed and how it was configured, enabling effective auditing and forensic analysis. This capability is indispensable for confirming software integrity, assessing system states, and potentially informing legal or corrective actions when failures or breaches occur.

4.6.24 SHIELD project

To illustrate how attestation can support practical applications, let's look at an example from a European project we participated in, which focused on "security as a service." In this approach, security functions are deployed as needed over the network infrastructure used by the user. This is typically achieved by creating and deploying virtualized network security functions (VNSFs) where they are needed. In our project, we used a trust monitor to perform attestation on the infrastructure, which was primarily software-based.

In the setup, the trust monitor worked in conjunction with an NFV (Network Function Virtualization) infrastructure. This infrastructure, managed by an orchestrator, was the target for deploying various security functions. Additionally, a store provided network security functions as software objects, while a security dashboard displayed the system's status. If any security misconfiguration or integrity issue was detected, there was also a reaction mechanism to address it. This framework was part of the Shield Horizon 2020 project, an NFV-based security infrastructure aimed at monitoring, enforcing security actions and reactions, and using remote attestation to verify the integrity of the entire infrastructure.

The trust monitor received two key inputs. The first input came from the infrastructure itself, allowing the monitor to assess whether the system was correctly configured and

running the expected components. This involved periodic requests to each component for attestation results, ensuring that the trust monitor could check each component's current measurements. The second input came from the VNSF store, providing "golden" measurements—known values for expected configurations and software hashes, which serve as benchmarks for verifying each component's integrity.

When there was a mismatch between a component's actual measurement and the golden measurement from the store, an alert was triggered. This alarm was then analyzed, allowing the system to identify what went wrong and initiate appropriate remediation actions. Through this combination of measurements and golden benchmarks, the trust monitor maintained a secure infrastructure by continuously verifying the integrity and correct configuration of deployed security functions.

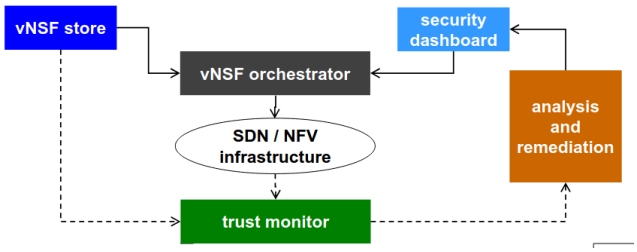


Figure 4.13: SHIELD project

Golden value creation

Let's begin with creating the "golden values." Here, we have the store of network security functions. When the trust monitor starts, it will request the security manifest. Every piece of software comes with a manifest, which, among other things, specifies hash values. If a virtualized network security function (VNSF) consists of multiple components, the manifest includes a hash for each part.

The trust monitor retrieves the VNSF's security manifest, extracting the measurements for each component. If a particular component's hash is already in the database (perhaps because it's used by another VNSF), the trust monitor will skip it. However, if the measurement is new—meaning this component hasn't appeared in any other VNSF yet—it will add this hash to the whitelist database. This list, also known as a "whitelist" or "accept list," identifies the components that are authorized to run on the system.

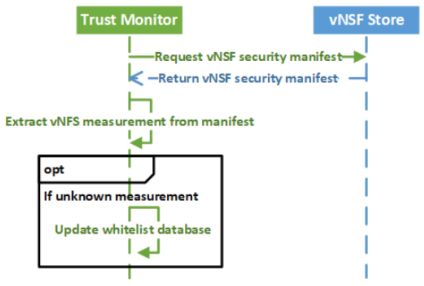


Figure 4.14: Golden value creation

Initial deployment of a security function

When deploying a security function, the orchestrator initiates the process by requesting the trust monitor to verify the state of the host, or "middle box," where the function will run. Before placing a virtual network security function (VNSF) on a host, it's essential to confirm that the host system is in a secure and trusted state. This includes checking that the operating system, Docker container runtime, and boot process are intact and secure.

To do this, the trust monitor initiates a remote attestation by sending a nonce to the target host. In response, the host provides a TPM quote and event log as proof of its current integrity. If the attestation results match the expected, secure values, the trust monitor returns a "success" message to the orchestrator. This validation allows the orchestrator to include the middle box in the list of authorized nodes for deployment.

However, if the attestation fails to verify, the trust monitor returns a "failure" message, and the host is excluded from further orchestration. An alarm is also triggered, notifying the security manager to investigate the physical node's status. Until resolved, the node remains isolated and is not considered for any subsequent deployment steps.

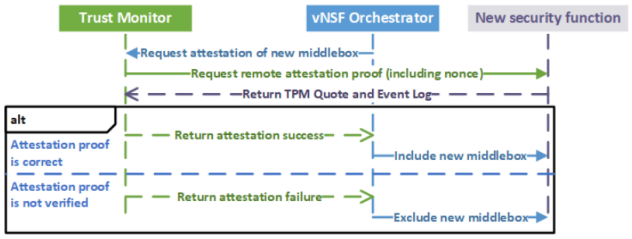


Figure 4.15: Initial deployment of a security function

Periodic attestation of security functions

Next, there is the process of periodic attestation, which the trust monitor performs automatically at intervals determined by the security manager. These intervals are typically based on the expected speed of potential attacks and are usually set in the range of minutes—such as every 10, 30, or 60 seconds—rather than hours.

For each cycle, the trust monitor requests the current network state from the orchestrator, which identifies the specific nodes currently involved in security functions. This ensures that only relevant nodes are attested, rather than the entire network, focusing on the nodes that a particular customer or application may be using (e.g., nodes 2, 7, and 35).

With the list of active nodes, the trust monitor requests a remote attestation proof from each middle box in this subset. Each node provides a TPM quote and event log as proof of its current integrity. If the attestation result is positive, the network continues to operate as normal. However, if a failure is detected, the trust monitor triggers an alert through the security dashboard, notifying the operator of a potential issue with the affected node.

At this point, the operator has the option to manually intervene, such as by terminating, isolating, or reconfiguring the problematic node. While automated responses could be implemented with AI, this project favors a "human-in-the-loop" approach, leaving the final decision to the operator after receiving the alert.

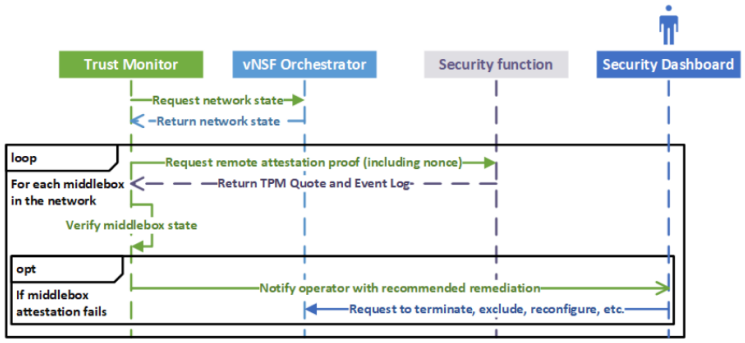


Figure 4.16: Periodic attestation of security functions

4.7 Keylime

Keylime is an open-source remote attestation project hosted at the Cloud-Native Computing Foundation (CNCF) that is designed with a focus on cloud-oriented environments. It leverages a physical TPM to create a highly scalable RA framework with a straightforward architecture.

Keylime includes several key features: remote boot attestation, which ensures the integrity of the boot process by verifying the platform's state at boot time; Linux IMA support, allowing for periodic runtime attestation through IMA to verify system integrity during operation; registration of multiple agents to a central verifier, enhancing flexibility and scalability in attestation processes, in fact, an agent is the piece of software that you need on the node to create the attestation port, m.t. one can have as much as needed ; and a additional certificate infrastructure to facilitate secure communications and verifications within the framework. Why it's part of the standard? Because at least initially, Keylime gave to each agent also a public key certificate in order to perform operations like signatures, but it's rarely used nowadays.

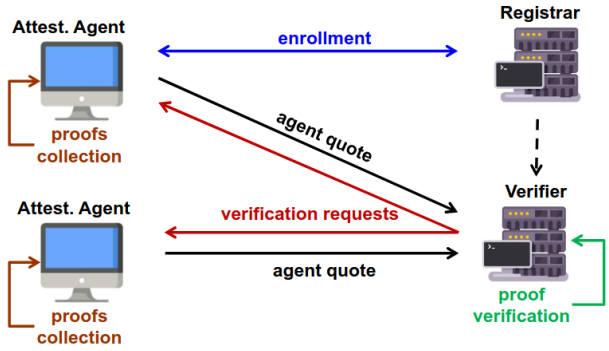


Figure 4.17: Keylime schema

4.7.1 Keylime Structure

The Keylime framework is composed of several key components that work together to enable remote attestation of nodes.

The **attester agent** is a software component installed on each node that needs to respond to remote attestation requests. Its primary function is to retrieve the TPM quote, which includes the PCR values signed by the TPM's internal key. In addition to the quote, the agent gathers other necessary data, such as the IMA measurement list for PCR 10. Optionally, the agent can also listen for revocation messages if certificate-based attestation is used and a certificate has been revoked.

The **registrar** manages the enrollment of attester agents. When a node with an agent is deployed, it must be registered with the registrar, which assigns it a unique user identifier. The registrar also handles the management of agent keys, including the agent's signing key and the endorsement keys stored in the TPM.

The **verifier** is responsible for attesting the nodes by communicating with the attester agents. It requests and validates the TPM quotes and can send revocation messages if an agent is determined to be in an untrusted state, especially in cases where certificate revocation is utilized.

Finally, the **tenant** represents the entity that owns or manages a specific part of the infrastructure. It provides a unified interface for managing the agents across the infrastructure, enabling consistent and centralized management of attestation operations.

4.7.2 General Schema

This is the schema we are discussing. In our network, we have two nodes that require attestation. Each of these nodes needs to run an attestation agent. Additionally, there will be a machine designated as the registrar and another as the verifier. While it's not mandatory to have them separate, they are distinct processes. You can run both the registrar and the verifier on the same machine.

The registrar is responsible for enrollment. When an agent is deployed, it attempts to enroll itself, and the registrar notifies the verifier of this new registration, instructing it to perform periodic attestations for that node. The verifier will then regularly send verification requests to the various agents. Each agent gathers the necessary proofs, such as the PCR values and, if IMA is enabled, the IMA measurement list. It sends a quote containing all required information back to the verifier. This process is repeated for each node in the network.

While this architecture functions well, it does have a vulnerability: the presence of a centralized verification point creates a single point of failure. If the node containing the registrar or verifier is compromised or subjected to a denial-of-service attack, we lose visibility into the network's state. A possible approach to mitigate this is to adopt a decentralized architecture, and one of the possibilities that is being discussed is to use a blockchain.

In the event of a critical node being compromised, there would need to be a protocol to notify administrators to deploy a replacement node. This concept of a self-healing network is significant, as it allows the network to recover autonomously from attacks by excluding malicious nodes.

Regarding the chain of trust, you can choose to utilize an endorsement key or other keys as part of this framework. While the chain of trust is not strictly required, a minimum setup for a node with a TPM includes the endorsement key, which is always present. If privacy is a concern, the chain of trust can be established with identifiers like the IDevID

and LDevID.

At last, each time a quote is received, it will be verified against the golden measurements to ensure integrity.

4.7.3 Remote Attestation Procedures – RATS

The Remote Attestation Procedures (RATS) were proposed by the Internet Engineering Task Force (IETF) to enhance support for different platforms during the load time. RATS defines various actors involved in the remote attestation procedures, including the Attester, Relying Party, Verifier, Relying Party Owner, Verifier Owner, Endorser, and Reference Value Provider. The definition of all this roles sure helps in environments such as the cloud.

Additionally, RATS specifies several topological patterns, such as the **Background Check** and the **Passport** Model, to organize the interactions among these actors. The framework is documented in RFC-9334, titled "RATS Architecture," which lays the foundation for the remote attestation process. Numerous other RFC drafts are currently in progress, focusing on various aspects such as data formats, procedures, and attestation models to further develop the RATS framework.

Interaction schema

This diagram in figure 4.18 outlines the schema of interactions, with the verifier as the central component. The **verifier**'s primary function is to receive quotes and determine their validity—whether they are good or bad. To make this determination, the verifier collects evidence from the **attesters**, including quotes, mailing lists, signed PCRs, and other relevant data. It then provides attestation results to a relying party, which is an entity that trusts the verifier to conduct this verification.

However, this is just the basic setup. To enhance the process, we introduce the concept of the endorser. An endorser is an entity that designates the correct elements to be verified within the system. For example, if you've purchased a component from another company, you would need to measure it, but the measurements must be supplied by that vendor.

The verifier has an owner responsible for setting up the appraisal policy for the evidence. This is particularly important for software components, as you may have both current and legacy versions in your network. Questions arise regarding which versions to accept: Is it only the latest version, or are older versions accepted as well? You might categorize versions based on their status—some may lack functionality, while others could have critical bugs or security vulnerabilities. Policies should be established to define acceptable conditions for older versions, allowing acceptance of those that only have missing functionality but rejecting those with known issues.

In addition to the verifier owner, there is the relying party owner, who must decide the appropriate course of action based on the attestation results. They determine whether the results are acceptable and assess the severity of any issues identified.

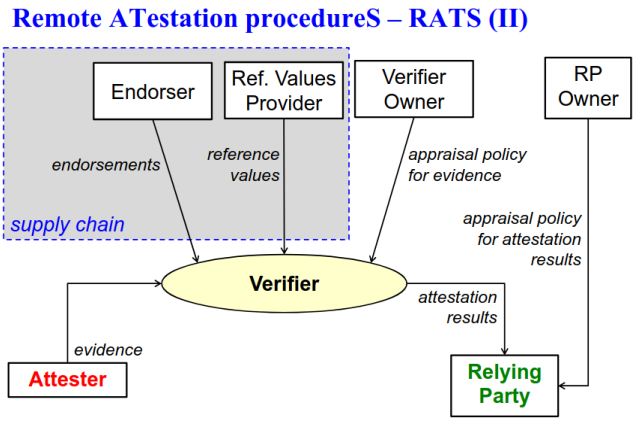


Figure 4.18: Remote Attestation Procedures (RATS)

RATS – Main Roles

Let's go now in details by providing a formal definition of the main actors.

The **Attester** is responsible for generating evidence when attestation is needed, typically at the request of the Relying Party. This evidence is essential for accessing services, such as those that utilize OAuth for authentication. This is important because even if a node has the correct token it may be infected.

The **Verifier** plays a critical role in comparing the evidence provided by the Attester against predefined reference values, applying an appraisal policy in the process. It also utilizes endorsements to identify valid attesters, ensuring that the attestation is credible.

Finally, the **Relying Party** evaluates the attestation results based on its specific policy, known as the Relying Party Policy. It is important to note that the Relying Party and the Verifier may be part of the same service, streamlining the attestation process.

Attestation Models

There are two main models for running attestation: the background check model and the passport model.

Background Check Model : In this model, the attester provides evidence, which is then passed on to the relying party. However, the relying party cannot evaluate the evidence on its own, so it sends the evidence to a verifier. The verifier reviews the evidence and sends back an attestation result to the relying party. This is similar to a background check, where an external authority is consulted to verify someone's status. For instance, when you go to buy a firearm, the shop (acting as the relying party) contacts law enforcement (the verifier) to confirm if you're eligible to make the purchase. Similarly, when a company hires an employee, it might conduct a background check, examining the candidate's online history to ensure they align with company values. Many companies now require applicants to share their social media profiles for such checks. In both cases, external evidence is checked by an independent verifier on behalf of the relying party.

Passport Model : In this model, the attester creates evidence and submits it directly to the verifier. The verifier then reviews this evidence and issues an attestation result back to

the attester, who can present it to the relying party. It's like presenting a passport when traveling: instead of the relying party (e.g., a border control officer) having to verify your identity independently, you present a pre-verified document. This model allows the attester to proactively acquire verification from the verifier, making it readily available for the relying party. Using the previous example, imagine obtaining a certificate of good standing from the police before entering a store. This way, you're pre-approved, and the store doesn't need to conduct further checks.

In both models, the final attestation result is provided to the relying party, which relies on this verification to establish security and trust.

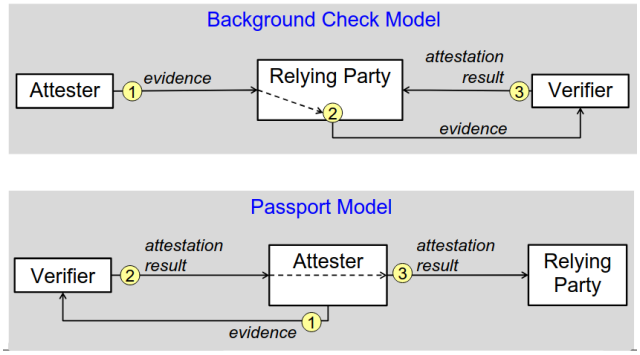


Figure 4.19: RATS attestation procedure

4.8 Veraison (VERificAtIon of atteStatiON)

Nowadays, one of the main problems in attestation is that every companby is developing its own solution, and this is a problem because there's no interoperability. We would like to have a generic verifier, which can be used with any kind of attestation produced by any kind of company.

Veraison is an open-source project designed to enhance consistency for Verification Services in the realm of remote attestation. It implements various standards and provides a set of libraries that allow for extensive customizations.

Originally generated by ARM's Advanced Technology Group (ATG), Veraison was later adopted by the Confidential Computing Consortium within the Linux Foundation. It supports a variety of architectures and RoT implementations, making it a versatile solution in the field of attestation.

While Veraison does not offer a standard agent implementation, it features a flexible structure for evidence provisioning. Its high customizability enables users to select only the necessary features for their applications, facilitating the easy development of custom functionalities tailored to specific requirements.

4.8.1 Veraison Architecture

The architecture of Veraison is shown in figure 4.20. The components highlighted in blue are managed by Verizon Trusted Services. Verizon handles several plugins to facilitate attestation, including a handler for endorsements that allows connection to any endorser of

your choice, as well as a handler for evidence. Within the provisioning process, there is a plugin manager for coordinating these plugins. Verification occurs outside of this managed area, and there is also a key-value store for maintaining relevant information.

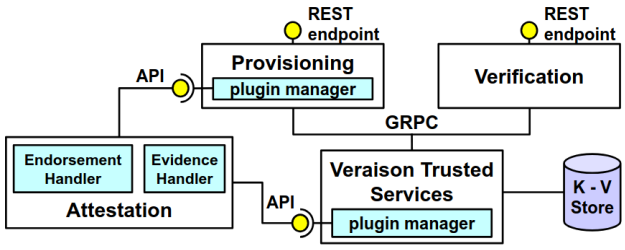


Figure 4.20: Veraison architecture

4.8.2 Verification

Verification is at the core of Verizon's functionality, and it involves a complex, multi-step process. Upon receiving a request, the handler performs the verification steps, all of which are integral to Verizon's operations. This includes support for different media types, which can each be processed in a customized way. The process also involves establishing trust anchors—root authorities that certify the TPM or your own root of trust. Key stages include handling claims, endorsements, validation, appraisal, and signatures, among other tasks.

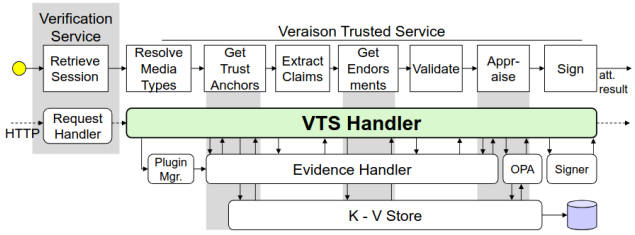


Figure 4.21: Veraison verification

4.8.3 Data formats

Data formats are essential for achieving interoperability in attestation. Verizon's architecture natively supports ingestion and generation of various attestation formats.

Entity Attestation Tokens (EAT) The attestation results are formatted as Entity Attestation Tokens, which can be encoded in CBOR or JSON. EAT is used as a standardized format for representing evidence, especially within the PSA (Platform Security Architecture) ecosystem—a security certification schema designed for IoT devices.

Evidence Formats Evidence can be presented in multiple formats, including:

- **PSA-EAT** — Developed under the PSA framework, primarily for IoT security.
- **CCA (Confidential Compute Architecture)** — The format used by ARM for secure compute environments.

- **TPM and DICE** — Trusted Computing Group (TCG) standards, where TPM is a longstanding standard, while DICE is designed for embedded and IoT systems.
- **Amazon Nitro** — Amazon's format for creating evidence within secure enclaves.

Endorsements and Reference Values Endorsement and reference values are represented using CORIM (Concise Reference Integrity Measurement). CORIM includes:

- **COMID** for hardware and firmware modules,
- **COSWID** for software components, and
- **COBOM** (Concise Bill of Materials) for hardware, software, and cryptographic inventories.

The bill of materials is essential in security frameworks, such as NIST's, where inventory management is a primary phase of security planning. Each element's hardware, software, and cryptographic components are listed to support system transparency.

Appraisal Policy for Evidence For appraising evidence against policies, Open Policy Agent (OPA) provides an open-source solution where policies can be defined and applied. This enables a comparison between attestation results and the established policy.

Attestation Result Formats Attestation results can be formatted in EAR for general attestation and R4C for secure interactions, offering flexibility in secure communication.

This broad range of formats ensures compatibility across platforms, though the variety also introduces significant complexity.

4.9 Device Identity Composition Engine (DICE)

So, insofar we discussed the attestation as was originally conceived, that is using a hardware root of trust implemented with the TPM. That is perfectly fine, but it's a rather complex solution, because TPM is a complex object that needs either to be implemented as a physical chip, or if it is in firmware, it's a large component. There are some alternatives being proposed for systems that cannot have additional hardware, and even running large portions of firmware is not suitable for them. Typically, they are internet of things or embedded systems with limited computational capacity.

The Trusted Computing Group developed the Device Identifier Composition Engine (DICE) to establish secure device identities, focusing on identifying each layer of the device through cryptographic keys. DICE's approach centers on the Compound Device Identifier (CDI), a unique secret value, usually a private key, generated by applying a one-way cryptographic hash to combine a secret for the current layer and a measurement of the next.

DICE's process involves layered identity verification. Each layer, beginning with firmware, is measured and linked to a unique key. When a new layer is initialized, DICE generates a new key based on the prior layer's key and the measurement of the upcoming layer. This ensures that any tampering at one layer disrupts the chain, as the altered measurement would produce a mismatched key.

At the base of this chain is the Unique Device Secret (UDS), a distinct and randomly generated secret that initializes the first CDI value. The UDS is statistically unique, ensuring it cannot correlate with UDS values on other devices. In practice, this may be created using a physically unclonable function (PUF) to generate a unique random value specific to the device.

Each CDI is recalculated upon reboot, depending on the device's static software state. If the software remains unchanged, the same keys are generated; otherwise, new keys indicate tampering. Verification against a public key certificate confirms the integrity of the system—any unauthorized software changes disrupt this chain of trust, invalidating the certification.

4.9.1 DICE Layered Architecture

The DICE architecture begins at the boot stage, forming the foundational layer for secure identity. The DICE layer alone has access to the UDS, a crucial element for establishing trust, and calculates the initial TCI for layer zero by hashing the software code of this layer. Using a one-way function (hash), the DICE layer then generates a unique key for the next layer in the boot sequence.

Each subsequent layer repeats this process: it measures the software of the following layer, applies a hash function, and combines it with the current CDI to generate a new CDI for the next layer. This recursive measurement and key generation process continues across layers, linking each layer's identity to its software integrity. Typically, only a few layers are involved, but the approach allows for as many layers as needed.

In essence, these keys are dynamically generated based on the software at each layer. Identical keys are recreated only if the software at each layer remains unchanged, thereby ensuring that any modification to a layer disrupts the integrity chain.

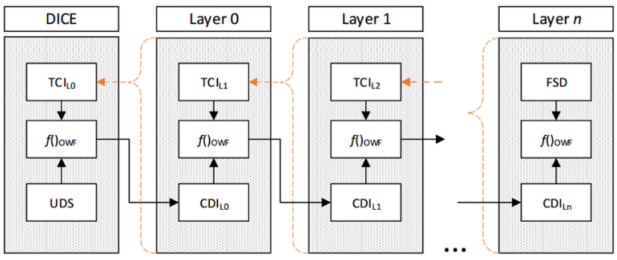


Figure 4.22: DICE layer architecture

4.9.2 Dice keys and certificates

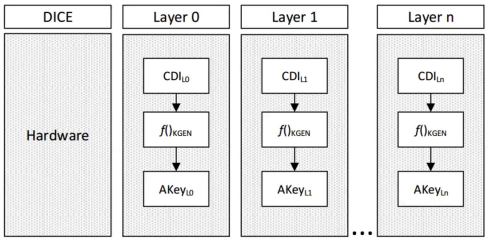
In the DICE architecture, each layer can have multiple keys and certificates, expanding beyond the initial key generated by the DICE layer for the first Compound Device Identifier (CDI). Starting from layer 0, a key derivation or key generation function produces additional keys for that layer, with each subsequent layer performing the same process. Rather than using the CDI directly, these derived keys serve in cryptographic operations.

Each layer may include an Embedded Certification Authority (ECA), which issues certificates for attestation or identity verification. This ECA is designed to sign only data it

generates, preventing potential risks associated with signing external data. There are three primary types of keys:

- ECA Key: Used to issue certificates for keys generated in the current or next layer, eliminating the need for an external certification authority—a practical approach for embedded or IoT devices.
- Attestation Key: Used to sign attestation evidence, ensuring integrity in reporting the device's state.
- Identity Key: Used exclusively for identity verification through challenge-response protocols, maintaining secure device authentication.

This layered approach allows for distinct roles for each key, although a single versatile certification authority could be employed for simplicity.



	CERTIFICATION	ATTESTATION	IDENTITY	NOTES
ECA Key	✓	✓	✓	Only sign known data
Attestation Key		✓	✓	Only sign known data
Identity Key			✓	May sign opaque challenge

Figure 4.23: DICE keys and certificates

4.9.3 Dice layered certification

> Each layer in the DICE framework can act as an ECA, establishing a certification hierarchy rooted in the manufacturer's root CA.

Each ECA can produce two different types of certificate. The device manufacturer provides this root CA, which issues the first certificate for the CDI (Compound Device Identifier) of the DICE layer. Since the DICE layer is meant to be immutable, except when a firmware update is performed, this initial certificate remains constant and can be externally verified as a marker of platform authenticity. The ECAs in subsequent layers issue certificates (the second kind) that represent the device's identity and attestation status.

Each certificate includes specific X.509 v3 extensions crucial to DICE architecture compliance, namely 'DICE PCB Info' and 'DiceTcbInfo'. These extensions contain vital identification and measurement data required for device authentication and security.

- **DiceTcbInfo Extension**: Each certificate must include the 'DiceTcbInfo' extension with OID:

  2.23.133.5.4.1 = joint-iso-itu(2).international-org(23).tcg(133).tcg-platformClass(5).dice(4).TcbInfo(1)

This extension provides a SEQUENCE of information specific to the device's target level, containing:

- **Names**: Identification fields such as *vendor, model, and layer*.
- **Measurements**: Versioning and security attributes, such as *version, security version number (svn)*, and *fwidlist* (a list of firmware IDs, each represented by a *hash algorithm and digest*).

The certificate, therefore, holds key information about the platform, without revealing its full details. Instead of explicitly naming firmware, it provides a digest. This digest allows anyone with the necessary knowledge to validate the platform's firmware identity, while keeping exact firmware information private.

4.9.4 Dice on RISC-V

The professor recenly completed another project on September 30 and are now preparing to launch the next phase. This project involved implementing DICE on a RISC-V platform equipped with a hardware cryptography accelerator. In this setup, layer 0 contains the Compound Device Identifier (CDI), also referred to as the device root key, which is accompanied by a root key certificate issued by the platform manufacturer.

For this solution, we integrated with Keystone, a system where the security monitor serves as the foundational layer. Here, the device root key signs the key of the Embedded Certification Authority (CA) specifically for the security monitor. The security monitor, in turn, generates an initial attestation key and issues the initial device ID certificate for the device, establishing a certified identity for the device.

In Keystone's architecture, trusted enclaves are established, isolated from the untrusted parts of the system. Each enclave can generate a local device identifier with its own certificate, signed by the previous layer, along with a local attestation key. Embedded CAs are not necessary within each enclave since they do not generate additional keys. Importantly, this entire chain of trust originates from the manufacturer's root certificate, which could pose privacy concerns if used directly in a user's system.

To address privacy, an option exists to associate a new certificate with the device identifier of each enclave. This approach allows users to use the same keys while issuing the public key to their own certification authority for signing. Consequently, the root of trust can shift from the manufacturer to the user's chosen authority. Once verified, this new certificate can be used exclusively.

If an attack or modification occurs in the platform, security monitor, or enclave, the system will automatically invalidate the certificate. Any changes in the trusted software or applications alter the generated keys, making the certificate invalid as it would now contain a different public key. Rather than storing keys persistently, keys are regenerated as needed and remain bound to the executing software, thereby reinforcing security.

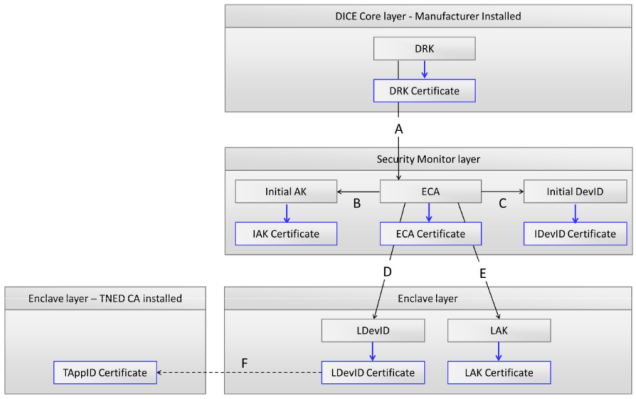


Figure 4.24: DICE on RISC-V

4.9.5 Open Profile for DICE

The Open Profile for DICE is a specification from Google for implementing DICE. In this model, each layer in the system is equipped with two types of Compound Device Identifiers (CDIs):

- **Attestation CDI** (mandatory)
- **Sealing CDI** (optional)

The CDI for each subsequent layer is derived using additional specific input values, depending on the CDI type. These inputs include:

- **Configuration Data:** Information regarding the integrity of the system.
- **Authority Data:** A hash representing information about the trusted authority used in verified boot.
- **Mode Decision:** The current operating mode of the device.
- **Hidden Inputs:** Values that are not disclosed in any certificate.

Furthermore, each layer has an Attestation keypair, which is derived from its corresponding Attestation CDI, enabling cryptographic attestation at each stage.

4.9.6 DICE and RIOT

RIOT (Robust Internet of Things) is Microsoft's specification for implementing DICE. Unlike the standard DICE implementation, RIOT utilizes a single Compound Device Identifier (CDI) for the entire device. Key aspects of RIOT's DICE model include:

- **Single CDI:** The CDI is generated by combining the platform's Unique Device Secret (UDS) with the measurement of the RIOT core, which is the only software component permitted to access the CDI.
- **Alias Keypairs for Attestation:** Each layer includes an Alias keypair for attestation.

- **Recursive Key Derivation:** Each layer N uses the measurement of the subsequent layer $N + 1$ to compute its Alias keypair. For the RIOT core, this Alias keypair is derived directly from the CDI.

4.9.7 Cloud providers and DICE

Let's wrap this chapter up by showing what the major cloud providers are doing with DICE.

Google Cloud Attestation

Google Cloud provides optional attestation for platform security, with key capabilities supported on services such as Google Kubernetes Engine (GKE) and Cloud Run. A central feature is binary authorization, which includes various scans and analyses—such as vulnerability scans, regression tests, and signing of the container image hash—to ensure container integrity.

At deploy-time, Google Cloud enforces security controls by blocking any container deployment lacking a valid signature. If the container meets the specified policies, deployment is allowed, ensuring policy-compliant security for containerized applications.

Amazon Web Services Attestation and Security

Amazon Web Services (AWS) provides several security features related to container integrity, though not traditional attestation. AWS Elastic Container Registry (ECR) stores container images that can be executed, and Amazon Inspector offers scanning capabilities for these images.

Amazon Inspector scans container images each time a new image is pushed or a new vulnerability definition is added. An additional feature, enhanced scanning, checks for vulnerabilities in the operating system and within specific programming languages. Users can specify the programming language used, allowing AWS to scan for known vulnerabilities relevant to that language. AWS also checks service vulnerabilities, particularly regarding network access, to enhance container security.

Amazon Web Services NITRO Attestation

AWS provides a form of attestation through NITRO, a specific type of enclave used for secure execution within a VM. NITRO enclaves are isolated and constrained, only able to communicate with their parent EC2 instance. The enclave can request attestation from the NITRO manager to prove its integrity and request services. For example, an enclave can ask the NITRO manager to attest its state, and the NITRO manager can request the parent EC2 instance to sign or decrypt data on behalf of the enclave.

The attestation process in NITRO occurs during load time, ensuring that the correct binary is loaded into the enclave. Once the correct binary is confirmed, runtime services are provided, and the attestation process is no longer necessary. While attestation takes place at runtime, it primarily verifies the initial loading of the enclave, rather than continuous runtime checks.

AWS Nitro's PCRs

AWS Nitro utilizes PCRs intead of TPMs for various attestation measurements. The following PCRs are defined for different components:

- **PCR0 (Enclave Image File):** Measures the enclave image file, excluding the section data.
- **PCR1 (Linux Kernel and Bootstrap):** Measures the Linux kernel and boot RAM filesystem (ramfs) data.
- **PCR2 (Application):** Measures the user applications, excluding the boot ramfs.
- **PCR3 (IAM Role Assigned to the Parent):** Attestation succeeds only if the parent EC2 instance has the correct IAM role assigned.
- **PCR4 (Instance ID of the Parent):** Attestation succeeds only when the parent EC2 instance has a specific instance ID.
- **PCR8 (Enclave Image File Signing Certificate):** Attestation succeeds only if the enclave was booted from an image file signed by a specific certificate.

Azure Confidential Containers

Azure Confidential Containers provide secure deployment and attestation of containers through the following process:

- Upon container deployment, a token is generated and signed by the cloud node.
- The token is verifiable by a remote entity.
- The token includes information about:
  - The correct deployment of the container in a Trusted Execution Environment (TEE).
  - The use of a VM-based TEE.
  - A hardware-based and attested TEE.
  - Full guest attestation.
- The solution provides additional data and code protection.
- There is no need for a specialized programming model or special management.

Chapter 5

Social Engineering

First of all, we should dead with what sociology really is, then we will master the vocabulary to be able to put a name on phenomenons, and then we will deal with social engineering. We will apply some sociological theories to think of a new way to perceive and represent the complex relationships among machines and human beings.

5.1 Sociology

Let's kick off with a definition of sociology.

> Sociology is the **science of social phenomena** subject to natural and invariable laws, with the goal of discovering these laws.
> - Auguste Comte, 1839

The important point is not the definition, but the fact the definition contains many information about the cultural environment from which it comes. If you think about it, the definition focuses on the scientific method, which tries to quantify something in a deterministic fashion, becase one believe that all the necessary informations are already there. But natural laws are more complicated than these and human laws are even more complicated.

To sum this up, first, one can quantify social interactions, today we do apply maths and statistics to social facts. Second there's a sort of cause and effect relationship between social phenomena: just as at that time they believed that nature was describable in this way, they came up with the idea that society could be as well.

Another definition of sociology is the following:

> Sociology is the study of human social life, groups, and societies. - Antony Giddens

As you can see, the scope of the definition was quite reduced.

There are also other figures that are important to keep in mind. The first one is German sociologist and economist Max Weber, because he guides us in understanding two key aspects of sociological inquiry. First, it is **evolutive**: We live in a context of digital and analogical interactions in which, if you don't take a stand for something or for somebody, it means you're out of the debate. If you think about it, social media algorithms have been coded in

order to emphasise this sort of polarisation effect, because the more polarised the debate, the more engaged the debate becomes, which makes sense for an economical standpoint, and this kind of interaction is becoming the natural of the way we interact. The second lesson is that the **uncertainty of data** is the only certainty we have. There might be systemic errors in your data sets, accidental errors, and they are due to the sometimes the impossibility to have direct access to some sort of phenomenon (think for instance about criminality). The fact that they are somehow at least partly accessible limits the scope of the conclusions you are reaching. Even biases and subjectivity are a problem, for instance, one reinvents the past when one tries to recall it, which is just the way our brain works.

The second relevant figure is Charles Wright Mills, a sociologist from the US, the 50s and the 60s, the one who studied the white collars. The key concept is the **sociological imagination**, which is the ability to see when you want to social phenomena through the eyes of scientific inquiry, not through common sense and stereotypes.

5.1.1 Basic Sociological Vocabulary

Sociology uses specific terms to describe the elements of society and social behavior. Below are key concepts:

Norms are the rules and expectations by which a society guides the behavior of its members. With members we refer to social actors, which can be individuals, groups, or institutions. After all, sociology uses theatrical metaphors to describe the interactions within a society. So, norms are sets of rules that you are given because you are recognised as a member, as an actor within a context, a society. There are mainly two types of rules. Explicit norms and implicit, or tacit, norms. Cyber criminals rely mostly on the latter ones, because they can be broken more easily. After all, they are invisible to newcomers, and they are not written down, and they have to be learned by experience.

Values refer to collective ideas about what is good, desirable, and proper. Those are tricky elements, because the more social they are, the less they seem to be, because, if a set of values is very well spread within a group, each member of the group feels like the values are obvious.

Role encompasses the set of norms, behaviors, and expectations associated with a particular social status or position within society. Roles guide how individuals are supposed to act and interact with others in specific contexts. As described before, they are strictly connected with the idea of playing the game or acting society. We are usually more than a role, but we decide to show only a part of ourselves according to the context we are in. Roles are not self-assigned, they are recognised by the others, and indeed, sometimes when you feel you are not recognized in one role, you feel uncomfortable.

Social Structure is the organized pattern of social relationships and social institutions that together constitute society.

Culture consists of shared beliefs, values, and practices. In the end, humankind was able to organize social interactions against chaos. That's why we can speak of societies, no matter how simple, small or very complex and big they are. But they are very well structured.

5.1.2 The social iceberg

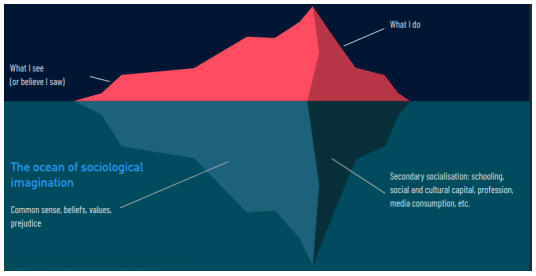
To conclude this sociological journey, consider the image of an iceberg—a fitting metaphor for social relations. Above the surface, we find two dimensions. The first relates to corporality: our body, nervous system, and senses help us perceive and interact with the world. However, our senses are not foolproof; they can mislead us. Hallucinations and memory distortions, for instance, demonstrate the limits of our perception. Witness accounts of even simple events, like a car accident, often differ significantly from each other and from camera footage. This highlights how imperfect and subjective our senses are—they are both gateways to the environment and sources of error.

The second dimension involves processing sensory input through our mental filters: past experiences, memories, learning, and adaptability. These filters shape how we interpret the world and decide how to act—or not to act. Sociologically, both action and inaction are forms of behavior, influenced by this interplay of perception and processing.

Below the surface of the iceberg lies a deeper realm of human experience. This includes the mental "autopilot" we rely on to navigate most of our daily activities. We don't constantly "meta-think" or reflect on every action, like sitting in a chair or walking. These routines are built on assumptions that rarely fail us—until they do. When something disrupts our expectations, it forces us to pause, question, and adjust. This reflects the philosopher Hume's insight: we assume the sun will rise tomorrow because it always has, but that certainty is not guaranteed.

Additionally, from birth, we begin absorbing knowledge through a process sociologists call primary socialization. Caregivers teach us how to express needs, interact with others, and adapt to our environment. Early lessons—both instinctive and taught—form the foundation of our behavior.

As we grow, secondary socialization takes over, enriching our "autopilot" with experiences from school, friendships, work, and culture. These interactions shape our social network, values, and identity. From the music we enjoy to the relationships we build, this ongoing process continually informs how we navigate and understand the world.



5.2 Vocabulary

In general it is important to provide a clear definition of complex phenomena to be able to apply more then a label to them. As such, we will try to analyze different attempts to give a name to cyber criminal activities. During the years, many definitions have been provided, meaning that we need a way to classify them too.

> One of the leading factors leading to difficulties in estimating cybercrime is exactly the fact that we don't have very well formed definitions nor a rigorous classification method.

The wider and richer the vocabulary, the more complex the thoughts you can make, you can formulate.

We begin with the period from 1995 to 2000, during which the term "cybercrime" was not even the dominant word used in discussions related to the field. This stands in stark contrast to the following window, 2001 to 2018, when "cybercrime" emerged as the most frequently used term in scientific literature. It's worth noting that 30 years ago, there wasn't even a widely agreed-upon term to describe cybercrime in a scientific context.

While 30 years might seem like a long time, it's relatively brief in the grand scheme of scientific progress. At the same time, it represents a significant period during which research and terminology have evolved. For instance, efforts to classify the various forms of cybercrime often result in taxonomies that, no matter how comprehensive, fall short of fully bridging the gap between theoretical frameworks and real-world occurrences.

From such classifications, it becomes evident that creating broader categories or groups can be helpful. These allow for more efficient communication and problem-solving without the need to enumerate every specific instance when referring to a broader class or subclass of cybercrime.

Year	Organization	Definition of Cybercrime
1994	The United Nations	"The United Nations manual [23] on the prevention and control of computer-related crime (1994) uses the terms, computer crime and computer-related crime interchangeably. This manual did not provide any definition" [18] (p. 116)
2000	The Tenth United Nations Congress on the Prevention of Crime and the Treatment of Offenders	1. "any illegal behaviour directed by means of electronic operations that target the security of computer systems and the data processed by them." 2. "any illegal behaviour committed by means of, or in relation to, a computer system or network, including such crimes as illegal possession and offering or distributing information by means of a computer system or network" [24] (p. 5)
2001	The Council of Europe Cybercrime Convention (also known as The Budapest Convention)	"action directed against the confidentiality, integrity and availability of computer systems, networks and computer data as well as the misuse of such systems, networks and data by providing for the criminalisation of such conduct" [25] (p. 2)
2007	The Commission of European Communities	"criminal acts committed using electronic communications networks and information systems or against such networks and systems" [26] (p. 2)
2013	Shanghai Cooperation Organization (SCO) Agreement	"the use of information resources and (or) the impact on them in the informational sphere for illegal purposes" (cited in Malby et al. [27] (p. 15))
2013	Cybersecurity Strategy of the European Union	"a broad range of different criminal activities where computers and information systems are involved either as a primary tool or as a primary target" [28] (p. 3)
2016	Commonwealth of Independent States Agreement	"a criminal act of which the target is computer information" (cited in Akhgar et al. [29] (p. 298))

Figure 5.1: Organizational definitions of cybercrime.

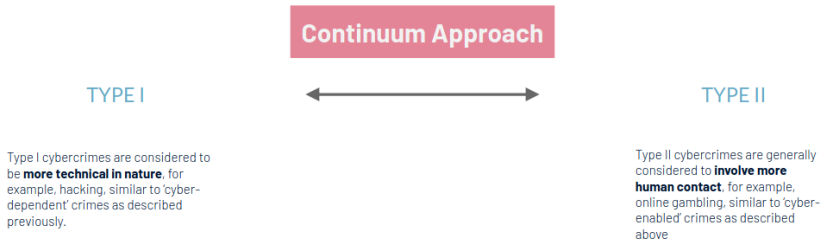
As you can see, the definitions are quite different from each other, some are broader, some are more specific.

There are several attempts to try and make it more clear and try to classify those definitions in order to make it simpler to understand for those maybe who lack some tech knowledge about cybersecurity. For example, one proposal is to adopt a sort of categorical approach to the systematization of definitions. A categorical approach is a systematic way to classify things, and with this approach we can discern between cyber-enabled crimes and cyber-dependent crimes. **Cyber-enabled** crimes are traditional crimes that predate the

advent of the technology, and are now facilitated or have been made easier (i.e., enabled) by cyber technology. Crimes range from white-collar crime to drug trafficking, to online harassment, terrorism and beyond.

Cyber-dependent crimes are crimes that arose with the advent of technology and cannot exist (i.e., dependent) outside of the digital world, e.g., hacking, such as ransomware attacks or hacktivism.

This classification is very rigid, in fact, if you belong to one of those category, you can't belong to the other one. We can try to fix this rigidity with another approach: if we consider the two classifications as the extremes of a continuous function, each criminal activity should be placed somewhere on this continuum.



A less rigid categorical approach is possible without resorting to the continuum, by discerning different properties of a crime and activities.

From figure 5.2, we can see 2 different classification, which provides different labels for the same definitions. This is just to say that even today, we don't have a clear classification.

Figure 5.2: Trichotomic definition of cybercrime.

As for what concerns the definition of what a cybercrime is, there's no a clear definition. Those can vary depending on perspective, sensitivity, and the traits emphasized in its definition. Different stakeholders, institutions, and interests shape how it is understood and defined. What one might highlight as key traits or concerns may not align with another's perspective, and all those factor may influence the definition of cybercrime.

Mr. Frederick Chung, a former research director at the NSA, describes cybersecurity as fundamentally rooted in **adversarial social interactions**. While it clearly involves technology and engineering—emphasized by the "cyber" prefix—what sets it apart is its focus on the science of society and the dynamics of conflict. Essentially, cybersecurity deals with humans defending systems and machines against other humans, which often use machines as tools for attack. This definition highlights the need for a multidisciplinary

approach that goes beyond traditional technical disciplines to fully grasp the complexities of cybersecurity.

Fom this definition, we can expand a bit more on the contect of *cyber*, which is usually a prefix that refers to the **cyberspace**, a term coined by William Gibson in his 1984 novel *Neuromancer*. This same term is used by Public Safety Canada to define "he electronic world created by interconnected networks of information technology and the information on those networks". The enphasis in this definition is on two things: the communication that is mediated by the network, that isn't necessarily between humans, but may be between machines, and the **information** that is conveyed by the network, and also the stimulus for interconnection.

5.2.1 Cybersecurity and it's many definitions

The discourse on security revolves around mitigating risks and controlling hazards, often represented by attackers—whether human or machine—targeting us or our properties. Security involves protecting these properties and addressing potential dangers that could harm them. An essential dimension of this discussion is the ability to ask the right questions, as highlighted by Buzz, Weaver, and Wild (1998). They propose a cognitive framework for analyzing security in cyberspace: understanding who is securing what, against which threats, for whom, and why, along with the conditions and outcomes. This serves as a conceptual checklist for planning or analyzing security strategies.

According to Craigen et al(2014), we may identify 9 different deinitions about cybersecurity.

1. "Cybersecurity consists largely of defensive methods used to detect and thwart would-be intruders." (Kemmerer, 2003)
2. "Cybersecurity entails the safeguarding of computer networks and the information they contain from penetration and from malicious damage or disruption." (Lewis, 2006)
3. "Cybersecurity is the collection of tools, policies, security concepts, security safeguards, guidelines, risk management approaches, actions, training, best practices, assurance and technologies that can be used to protect the cyber environment and organization and user's assets." (ITU, 2009)
4. "The ability to protect or defend the use of cyberspace from cyber-attacks." (CNSS, 2010)
5. "The state of being protected against the criminal or unauthorized use of electronic data, or the measures taken to achieve this." (Oxford University Press, 2014)
6. "The activity or process, ability or capability, or state whereby information and communications systems and the information contained therein are protected from and/or defended against damage, unauthorized use or modification, or exploitation." (DHS, 2014)
7. "The art of ensuring the existence and continuity of the information society of a nation, guaranteeing and protecting, in Cyberspace, its information, assets and critical infrastructure." (Canongia & Mandarino, 2014)

8. "The body of technologies, processes, practices and response and mitigation measures designed to protect networks, computers, programs and data from attack, damage or unauthorized access so as to ensure confidentiality, integrity and availability." (Public Safety Canada, 2014)

9. "Cyber Security involves reducing the risk of malicious attack to software, computers and networks. This includes tools used to detect break-ins, stop viruses, block malicious access, enforce authentication, enable encrypted communications, and on and on." (Amoroso, 2006)

Let's now talk a little bit about those definitions.

Cybersecurity is a big, complex topic, and different people think about it in different ways. Over time, researchers and experts have developed various approaches—or paradigms to help us understand and deal with cybersecurity challenges. Let's take a closer look at a few of these paradigms.

The first and most common is the **defense paradigm**. This approach is all about protecting systems as much as possible. It's focused on avoiding attacks, preventing harm, and keeping everything safe. The idea is straightforward: keep the bad stuff out and protect what matters. It's an idealistic view, assuming that if you work hard enough, you can secure systems completely. While this is great in theory, reality often throws curveballs.

That's where the **continuity paradigm** comes in. It's a more practical take, accepting that perfect protection isn't always possible. Instead of aiming for total safety, this approach focuses on surviving attacks and minimizing damage. Think of it like disaster management—when something goes wrong, you figure out what's most important to save and focus your energy there. Sure, you might lose some things, but the goal is to avoid a complete collapse. It's about keeping the core functions alive, even if other parts take a hit.

Next up is the **ecological paradigm**, which takes a broader view. This one treats cybersecurity like an ecosystem, where everything is interconnected and balanced. A system works best when all its parts are in harmony, and disruptions to one part can ripple through the rest. This perspective reminds us that systems are complex, and it's not just about protecting individual pieces—it's about keeping the whole thing working smoothly. Balance is key, and when things get out of whack, mitigation strategies help bring them back on track.

Finally, there's the **risk paradigm**. This approach gets straight to the numbers. It's all about assessing risks, measuring how likely something bad is to happen, and figuring out the potential impact. Once you know the risks, you can weigh the costs of protection against the potential losses. Sometimes, it might even make more sense to accept the risk rather than spend too much on safeguarding something that isn't worth it. This paradigm is especially useful when resources are limited, and you need to make smart decisions.

These paradigms don't compete with each other—they work together. The *defense paradigm* sets the ideal goal, the *continuity paradigm* helps when things don't go perfectly, the *ecological paradigm* looks at the bigger picture, and the *risk paradigm* makes sure decisions are grounded in reality.

Together, they show just how diverse and dynamic cybersecurity is. There's no one-size-fits-all solution, but by combining these perspectives, we can handle the challenges and keep adapting to new threats.

Now, do we need a 10th definition? According to the professor, yes we do. But to do so, we have to consider the similarities and differences between the definitions we have already seen. As for what concerns the similarities, they all seems to revolve around a few core ideas:

- **Technology**: At its core, cybersecurity depends on technology. Cyberspace itself wouldn't exist without it.
- **Events**: Cybersecurity focuses on events—incidents that need analyzing, mitigating, or learning from. Cybercrimes often aim to stay hidden, making evidence critical.
- **Methods and Strategies**: It's not just about reacting to threats but having processes and plans to protect systems.
- **Human Involvement**: Despite advancements in AI and automation, human input remains essential for oversight and adaptation.
- **Evolving Focus**: Over time, cybersecurity has shifted its focus—from protecting machines to data, privacy, and beyond. Legal definitions of cybercrimes have evolved too.

But there are also some distinguishing factors that set the definitions apart:

- **Sociotechnical Perspective**: Cybersecurity requires both technical skills and insights from social sciences and humanities. It's about understanding human and human-machine interactions, as highlighted by the need for a multidisciplinary approach.
- **Scaling the Problem**: Some definitions address cybersecurity on a broader scale, emphasizing its role in national security. This includes diplomacy, financial systems, espionage, and other political and economic concerns.
- **Warfare and Resilience**: Cyberattacks are now integral to modern warfare, affecting conflicts worldwide. Resilience, adaptability, and building systems that remain connected and reliable under attack are crucial.

Now, let's have our 10th definition:

> Cybersecurity is the organization and collection of resources, processes, and structures used to protect cyberspace and cyberspace-enabled systems from occurrences that misalign *de jure* from de facto property rights.

Disclamer: at this point, the professor just followed his stream of consciousness for 1 hour and i half, and even after reading it all, i still don't know what he was talking about. It was just like a bad sermon, so i won't write it down here.
Here's part of it just to understand what i'm talking about: Immanuel Kant, started to think that there was no abstract reason to passively have another entity decide on your life and your death, that you wanted to have your dignity, your personal dignity and the dignity of your group recognized so that you could pursue your happiness in life as far as other privileged groups like nobles, clergymen, and, of course, the monarchy did for centuries.

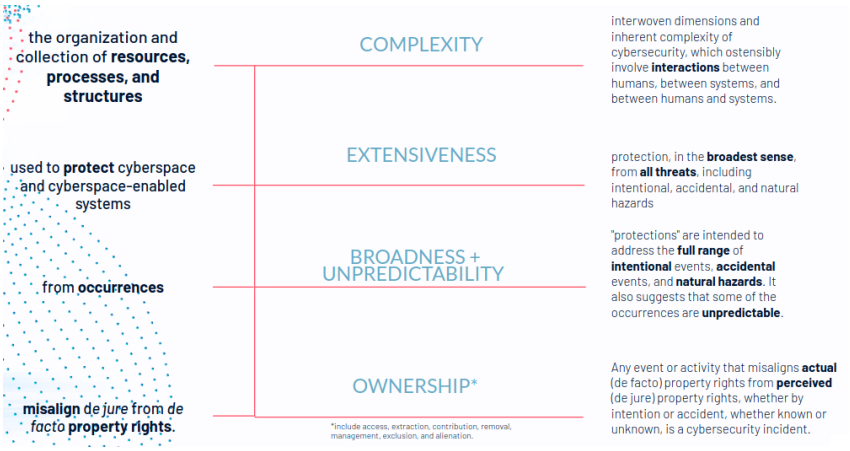


Figure 5.3: Insights on the 10th definition of cybersecurity.

Chapter 6

Electronic Identity

6.1 Introduction

When an information system becomes really large and complex, you need ways to manage authentication in a unified way, because having separate authentication systems or separate subsystem is creating opportunities for the attackers. In some instances, several **relaying parties**(RPs) may decide to delegate the authentication process to a separate entity, called **authentication server** (AS) to perform authN on their behalf, interaction with the authN client by executing an authentication protocol(there are several of them), and finally providing to the RP the authN result in the form of a **ticket** (or assertion).

Figure 6.1 illustrates the process of delegated authentication. In this scenario , a client seeks access to a service provided by a relying party. An authentication server serves as the central authority for verifying the client's identity.

The client begins by sending a request to the relying party. The relying party, requiring authentication, responds with a redirect to the authentication server, instructing the client to authenticate first.

The authentication server then performs an authentication protocol with the client, which could involve methods like challenge-response, one-time passwords, or Kerberos tickets. The outcome of this process is communicated to the relying party, allowing it to determine whether to grant access.

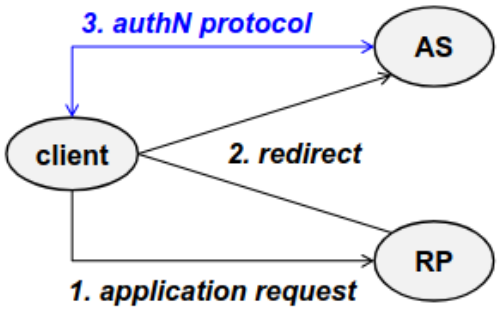


Figure 6.1: Delegated authentication schema

At the end of the authN process, the result is transmitted from the AS to the RP trough various possible ways, which can vary depending on various factors:

- speed, or reaction time
- security and trust
- on services, interfaces, network filters

In the end, there's no best solution, meaning that one have to evaluate the best solution for the specific case.

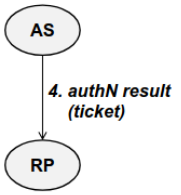
6.1.1 Possible ticket transmission methods

Let's now concentrate on the connection between the AS and the RP. The AS can transmit the ticket to the RP in different ways, which can be classified in three main categories.

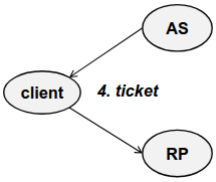
Push tickets(Figure 6.2a): the ticket is sent directly from the authentication server to the relying party. This seems easy but is not that straightforward(think about network filters, firewalls, etc).

Indirect push tickets(Figure 6.2b): Since a direct communication between the authentication server and the relying party may be impossible for various reasons, an alternative is that the ticket is generated by the authentication server, given to the client, and then the client will send it to the relying party. An example of this method is Kerberos.

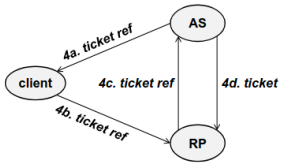
Push reference + pull tickets(Figure 6.2c): the ticket reference(not the ticket itself) is sent from the authentication server to the client, and then from the client to the relying party. Finally, the ticket will be pulled from the relying party.



(a) Push tickets



(b) Indirect push tickets



(c) Push reference + pull tickets

Figure 6.2: Ticket transmission methods

6.1.2 Problems with tickets

Each one of those solutions has its own problems:

- binding with client, or how to bind the ticket with a specific client and a specific request. If binding is not done correctly, replay attacks can be performed.
- ticket authentication, or authentication of the server from which is the ticket coming from

- ticket manipulation (at client). There are cases in which the ticket is passed through the client, so the client may alter that ticket, if not adequately protected.
- ticket manipulation (by MITM)
- ticket sniffing (in the network / at client), which exposes another concern: privacy!
- listening service at RP, which is necessary for the RP to listen for incoming tickets
- incoming firewall at RP, which can block incoming tickets
- ticket replay (by same client): not only when is bound to a client, but also if its coming from the same client(identity ticket shared among multi user client)
- ticket reuse (at different client): is it tied to the client or movable to another client?

6.1.3 Ticket protection

Ticket protection is essential for securing the transmission of tickets between the AS and the RP.

In the case of the direct transmission between the authentication server and the relying party, so the case of the push of the ticket, we have different solutions. One solution could be the creation a **digital signature** by the authentication server that will provide authentication and integrity, plus encryption of the data with a key possessed by the relying party. This means full protection of the ticket. Or if you prefer, one can create a **secure channel** where the requirement of the secure channel which security requirements are the authentication of the AS data integrity authentication, data encryption and protection against replay.

In case of indirect communication, digital signature provided by the AS and encryption for the RS is necessary and the only solution to protect the ticket.

More in general, protection for replay or reuse requires two things. First of all, a timestamp, with an associated validity window(as short as possible to still be usable). Secondly, binding the ticket with the ID of the user that authenticated, and eventually, its network address.

6.1.4 Federated authentication

Delegated authentication is typically performed within one single security environment, typically within a company (like PoliTo), since there is a very well-known set of users and RPs, and it is possible to setup one single authentication server and all the relying parties will delegate to that authentication server the authentication. It does not work well when we're in a public system like Internet, in which we don't know all the RP and there are several AS.

It is possible to federate the authentication among different security domains, by using a **federated authentication system**. The problem rely in the fact that there are various security domains, each one managed by a different authentication server so each security domain, internally, has got delegated authentication mechanism but the problem is how users from one domain are able to access another security domain. We need to create **trust**

relationship so that a relying party belonging to one domain will accept the authentication performed by the authentication server in another domain.

When we talk about federated authentication, unfortunately, we change the terms. The authentication server is typically named IDP (**Identity Provider**) while the relying party is named SP (**Service Provider**). They are basically the same thing; the names are related to the context.

6.2 XACML

XACML (eXtensible Access Control Markup Language) is a markup language derived from XML designed for describing authorization policies and managing access to protected resources. It defines policies in terms of:

- **Subject:** Entities such as users, computers, or services requesting access.
- **Resource:** Items like documents, files, or data that are protected and identified through URIs.

XACML also includes a language for managing access requests to these resources. It specifies:

- A structured data format to represent access request and response messages.
- Transmission over a client-server protocol of choice. After all, it's just a language.

XACML is an OASIS standard, with its syntax based on XML, making it widely compatible for authorization tasks in distributed environments.

6.2.1 Policy-based access control

So, the general concept of policy-based access control it's back to when the IETF (Internet Engineering Task Force) wanted to have a common way to describe admission control policies for Quality of Services on routers. Internet infrastructure is typically multidomain, since routers are managed by different corporations so it is needed a common way to describe if a specific request for QoS can be satisfied or not, after the requestor has been authenticated. So, there was an original RFC that specified the general framework for policy-based admission control, and then there was the COPS (Common Open Policy Service) protocol that tried to implement that concept.

COPS was not very successful but it leaded to the foundation of what came subsequently. After this initial attempt, there was a generalization, an extension, to the management of information systems (by DMTF: Distributed Management Task Force) and to the access control in distributed environments (work performed by OASIS to apply this concept to access control in multidomain environments).

6.2.2 Components policy-based access control

The general architecture of XACML is used for other applications, even though XACML is not used anymore, and it's made up of four components:

- **PEP = Policy Enforcement Point**: protects a resource and allows access only after verification of compatibility with the policy
- **PDP = Policy Decision Point**: receives all the data (policy, subject, resource, access type, context) and decides whether to permit or deny the access
- **PIP = Policy Information Point**: provides the info related to the access requested
- **PAP = Policy Access Point**: provides the policy applicable to the requested access

The general architecture of a policy-based access control, independent of XACML, is shown if figure 6.3, and be implemented in several ways.

Somewhere there is the *policy repository*, where all the access policies are stored, which have also been created by a policy administrator or security manager using the PAP, which is also in charge of retrieving the applicable ones. Then there is a subject (user/router/network service) that wants to access an protected object. In between the subject and the object to be accessed there is the PEP which must decide if this kind of access is permitted or denied. Typically, the subject is sending a request in the form of a triplet (S,O,T):" I am this subject S, I want to access this object O, and the kind of access I am requesting is of this type T". PEP does not know if this should be permitted or denied, so in any case it will block the initial request and it will talk with an appropriate identity: the PDP, the one that is in charge of taking a decision. The request MAY be (optionally) enriched with some context information (for example, let's imagine that we need to know what time of the day is it, or where the request is physically coming from, for example by using geolocalization, or I could see if there is a direct connection or if the user is using a proxy, etc.) which provides more information about the kind of request and are provided by the PIP. Everything is then packed together in the form of a XACML request. Now the PDP, in order to take a decision, must know which policy is applicable to this specific case. It will query the PAP and when it will have that policy, it will take the decision and will send back, in XACML format, the response. This is mediated by the context handler, which has the task of translating that to the language which is understood by the PEP. So, finally, the response is provided to the PEP that will implement the response: if authorized, it will allow the connection to take place.

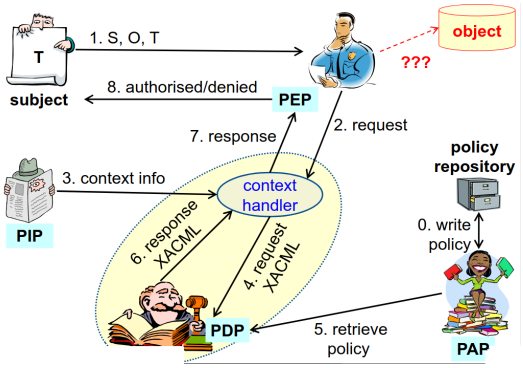


Figure 6.3: Policy-based access control architecture

Notice that XACML is limited to the part in yellow in the picture, the internal communication, because PEP is typically a kind of security control that already exist and was

considered many years ago before XACML. The typical example of a PEP could be a firewall (network or application firewall), or it can be any engine inside an application that must decide if a certain action is permitted or denied, or an Operating System in which you want to perform some operations on the files. Those already have their own way to be configured, take decisions, etc. That's why normally the context handler is in place, if not for this part (imagine we don't need/have that), at least for translating from a specific request format to the generic XACML. In the end, the scope of XACML is too small.

Context handler The PEP is tightly bound to the application or service (e.g., it can be a web server or a firewall) and it uses specific formats for requests/responses (few PEPs are capable of using directly XACML). The context handler converts access requests/responses from/to XACML and, if needed, it enhances the requests with the attribute values (obtained from PIP) often in the form of SAML assertions. When you put some information, you want to be sure that the information is correct, so an assertion is a strong statement: "Yes, in this moment it is 14:14:54" and it is possible to prove that, or it is possible to prove that the user has been authenticated with his username and password. Some strong foundation to take your decision are needed.

6.2.3 XACML policy format

A **PolicySet** serves as the primary container in an access control system, capable of holding individual policies or other nested PolicySets, allowing for a recursive structure. At its core, a PolicySet must contain at least **one policy**, which defines access control through specific components.

Rules within a policy establish **conditions** for granting or denying **access**. Each rule includes an **effect**, which determines whether access is permitted or denied, and an **optional condition** that specifies additional criteria to evaluate. The **target** of a policy defines its **scope** by identifying the circumstances under which it applies. This involves specifying the subject, the action, and the resource.

In a Target, **Subjects** are entities described by **attributes**, such as roles, IP addresses, or usernames. This approach supports Role-Based Access Control (RBAC), a model that focuses on roles within an organization rather than individual identities, making it more flexible and scalable. **Actions** outline the operations permitted under the policy, such as viewing, creating, or deleting resources. **Resources** refer to the protected entities, typically identified by URIs, that the policy governs.

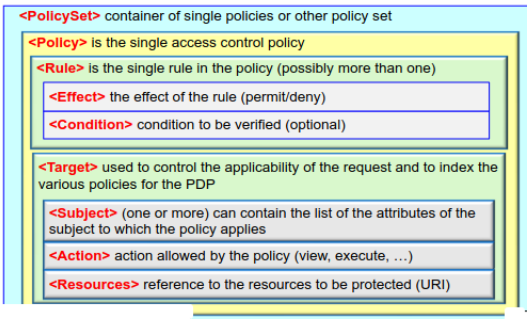


Figure 6.4: XACML policy format

6.2.4 XACML request format

A request in an access control system specifies the subject, resource, action, and environment, all derived from the request context. Each of these elements is described through attributes, which provide detailed information for evaluation.

The **resource** identifies the data or object to which access is requested. It is characterized by a set of attributes that define its nature and context. The **action** describes the operation to be performed on the resource, represented by its associated attributes. The **subject** refers to the entity requesting the action, detailed through its attributes, such as identity or role.

Attributes serve as the foundation for defining and evaluating access. Each attribute includes an **AttributeID**, which uniquely identifies it, and an **AttributeValue**, which specifies the required value for access to be granted or denied. Standard identifiers, such as usernames, certificate Distinguished Names (DNs), URIs, and action types, are commonly used to structure these attributes within the system.

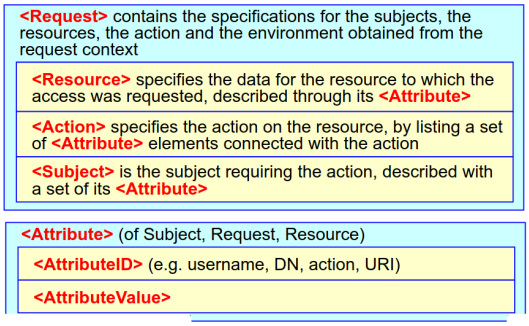


Figure 6.5: XACML request format

6.2.5 XACML response format

The response format in XACML conveys the decision made by the PDP regarding an access request. This decision is encapsulated in the result, which includes key elements for interpreting the outcome.

The decision reflects the outcome of applying the policy to the request. It can result in one of four possibilities: permit, deny, indeterminate, or not applicable. An indeterminate decision arises when inconsistencies or conflicts in the policy prevent a clear resolution, where access might be both permissible and deniable depending on the criteria. A not applicable decision indicates that no relevant policy exists for the specific request.

The response also includes the status, which provides additional context about the authorization process. This status contains a status code, a message detailing the decision, and any associated status details. Together, these components offer a comprehensive explanation of the PDP's decision-making process.

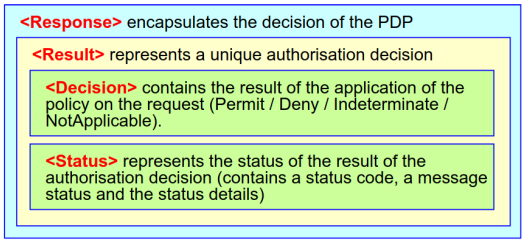


Figure 6.6: XACML response format

6.3 SAML

SAML (Security Assertion Markup Language) is a data format used for handling authorization and authentication assertions. It is designed to:

- Represent different types of assertions.
- Construct requests for assertions.
- Represent responses containing assertions.

Again, the transport protocol is not specified: there is the object, there is the format of the request, the format for the reply, but transportation is left to implementation.

The core concept in SAML is the **Assertion**, which serves as the base object to convey authentication and authorization decisions. SAML aims to standardize and simplify interactions required to establish permissions across a multi-domain distributed system, which requires trust relationships.

SAML is an OASIS standard with syntax based on XML. It also provides online tools to encode and decode various SAML formats and messages. A helpful tool for this purpose can be found at: `https://www.samltool.com/online_tools.php`.

There are various versions of SAML:

- SAML 1.0
  - november 2002
  - original version
- SAML 1.1
  - september 2003
  - can protect messages with XML-dsig
  - defines profiles for web browser SSO:
    - browser/artifact profile = token SAML by ref
    - browser/POST profile = token SAML by value

Nowadays most of the applications use SAML2.0, which is incompatible with the previous versions. It can still protect the messages with XML-dsig, but can additionally use XML-enc for identifiers, attributes and assertions (the reason is to protect the privacy of the requestor). In addition to browser/artifact and browser/POST it has defined new protocols, binding and profiles.

6.3.1 Use cases

Let's start to see the typical cases that have generated the various usage of SAML.

Web browser SSO use case

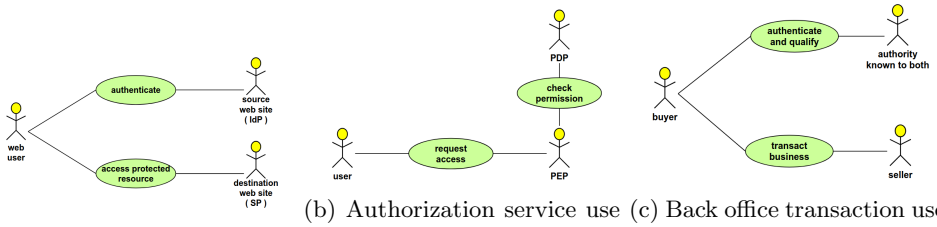
A web user would like to access a protected resource of the Service Provider (or relying party). The website does not want to implement by itself authentication and authorization, meaning that it relies on an Identity Provider to perform it. This is quite like the Delegated Authentication that we discussed. This is one of the ways (the most common) to implement it (e.g., when logging to `polito.it`, you are always redirected to `idp.polito.it` to prove the identity, and then, it will return an assertion about the authentication). SAML here is used from the IDP to the SP.

Authorization service use case

This use case is very similar to the XACML one: an user is requesting access to a resource, which access is mediated by a access control point, the PEP, that will use SAML to check permission with the PDP. The authorization can be returned in a SAML object (the SAML response). So, the SAML part is in the way from PDP to PEP.

Back office transaction use case

Imagine that a professor at PoliTo want to buy something on behalf of the university. In order to be able to do so, he must authenticate to a authority known by both parties and be qualified to perform that operation, receiving a SAML assertion that he can use to prove that he is allowed to do that.



(a) Web browser SSO use case

(b) Authorization service use case

(c) Back office transaction use case

Figure 6.7: SAML use cases

6.3.2 SAML Assertion

A SAML assertion is a **declaration of a fact** regarding a subject, such as the role of a user, made by a specific issuer. There are three primary types of assertions related to security:

- **Authentication**: confirms the identity of the subject. This is the one returned by the Identity Provider (IDF – SP)
- **Authorization Decision**: declares whether the subject is allowed to access a specific resource. This is for the PEP-PDP communication.

- **Attributes**: provides additional information about the subject, like roles or permissions.

SAML assertions are extensible, allowing for the addition of new assertion types as needed which can be used in a closed domain. Additionally, assertions can be digitally signed using XML signature to ensure authenticity and integrity. Why optionally? Because assertions can be carried in a secure channel, like an SSL one.

Information Common to All Assertions

All SAML assertions contain the following common information:

- **Issuer and Issuance Timestamp** – identifies the issuer and the exact time the assertion was issued.
- **Assertion ID** – a unique identifier for each assertion.
- **Subject** – includes the subject's name and the associated security domain.
- **Conditions** – specifies conditions under which the assertion is valid:
  - SAML clients must reject assertions with conditions they do not understand (just like critical extensions in x.509).
  - An essential condition is the *assertion validity period*, the lifetime of this assertion.
- **Other Useful Information** – may include an explanation or proof of the basis on which the assertion was constructed.

Authentication Assertion

An authentication assertion allows an issuer to declare the following information: **the Subject S, at time T, was authenticated with the mechanism M.**

That's important, because in the previous course we discussed that there are several authentication mechanisms, and they are not equivalent: some are strong, some are weak, maybe it is possible to combine them in a multi-factor authentication, and maybe there are some Service Providers that will permit access only if strong authentication has been performed, otherwise will not accept. Since authentication is outside the control of the SP, because it is performed by another entity (the IdP), it is needed a way to specify back to the requestor "Yes, you are authenticated and I tell you that I used these mechanism", it is possible to decide if it is strong enough or not.

> SAML itself **does not perform the authentication process** (such as password requests or challenge-response interactions). Instead, it provides a mechanism to link to the results of an authentication that has already been completed by an authentication agent.

Now, take a look at the figure 6.8 to see how an authentication assertion is structured. As we said, SAML is just XML, with an opening tag `<saml:Assertion>`. A version major and minor fields are present, as well as the AssertionID, which is often in the form of an

IP address followed by the date and time or the serial number. Then there is the Issuer, and the IssueInstant fields, which will specify the date and time of the creation (the Z at the end stands for Greenwhich Time). The second part contains Conditions, for instance, in this case the validity time frame. Then there is the AuthenticationStatement, which is the of the authentication. It is made by:

- AuthenticationMethod (e.g., password, reusable password, . . .).
- AuthenticationInstant: the date and time in which the user interacted with the IdP.
- Subject: NameIdentifier + SecurityDomain. Whitin the domain polito.it the user with username alioy has been identified.

If the issuer is trusted, then it is possible to accept this as a proof that the user was authenticated. Of course, there are problems of trust: if that has been manipulated, how it is transmitted, so the transmission of this SAML token goes back to the discussion of the previous pages about how to transfer the result of the delegated authentication.

```
<saml:Assertion
   MajorVersion="1" MinorVersion="0"
   AssertionID="192.168.1.1.12345678"
   Issuer="Politecnico di Torino"
   IssueInstant="2007-12-03T10:02:00Z">
   <saml:Conditions
      NotBefore="2007-12-03T10:00:00Z"
      NotAfter="2007-12-03T10:05:00Z" />
   <saml:AuthenticationStatement
      AuthenticationMethod="password"
      AuthenticationInstant="2007-12-03T10:02:00Z">
      <saml:Subject>
         <saml:NameIdentifier
            SecurityDomain="polito.it" Name="alioy" />
      </saml:Subject>
   </saml:AuthenticationStatement>
</saml:Assertion>
```

Figure 6.8: SAML authentication assertion

Attribute Assertion

An attribute assertion allows an issuer to declare the following information: **the subject S is associated with one or more attributes (attributes A, B, C, . . .) that currently (in this moment) have the values "a", "b", "c", . . .**
These attributes are typically obtained through an LDAP query.
Example: The subject "alioy" within the domain "polito.it" is associated with the attribute "Department," holding the value "DAUIN."

Take a look at the figure 6.9 to see how an attribute assertion is structured. The Initial tag and Conditions are the same as in the previous example. Then, it is specified that the example is an AttributeStatement, in which there will be a different content. The NameIdentifier is the same as before (security domain + name). Then in the Attribute part there will be the AttributeName "Dipartimento", again in the specified namespace (because each namespace may have different attributes) and the value will be DAUIN. So, if the system manager is performing Role Based Access Control, it is possible to say something like "Everybody which belongs to a certain department can access, for example, the portion

of the information system related to that department". This is the way in which is possible to know that a user belongs to that department.

```
<saml:Assertion ...>
   <saml:Conditions .../>
   <saml:AttributeStatement>
      <saml:Subject>
         <saml:NameIdentifier
            SecurityDomain="polito.it"
            Name="alioy" />
      </saml:Subject>
      <saml:Attribute
         AttributeName="Dipartimento"
         AttributeNamespace="http://polito.it">
         <saml:AttributeValue>
            DAUIN
         </saml:AttributeValue>
      </saml:Attribute>
   </saml:AttributeStatement>
</saml:Assertion>
```

Figure 6.9: SAML attribute assertion

Authorization decision assertion

Finally, to implement the PIP-PDP model, there is the authorization decision. An issuer declares that **it has taken a decision regarding an access request made by a subject S for an access of type T to the resource R based on the evidence E**.

The S, T, R elements are the access request, while the E is important for taking care of why the permission has been given/denied. For example, it could as a minimum say "this is based on polito.it.policy.2.5", because maybe the policy will change in the future and by specifying it there will be an evidence that in that version there was the allowed access to anybody of the DAUIN department for the resource. The subject can be a person, a program and a resource can be everything (e.g. a web page, a file, a web service etc.).

Take a look at the figure 6.10 to see how an authorization decision assertion is structured. The initial tags are the same as before, but in this case there will be the Authorization-Statement, in which there will be the decision field to specify if it is permitted to access the specified resource (`http://did.polito.it/m2170.php`) from the person specified in the Subject (the one corresponding to name alioy in the domain polito.it). In this case the evidence part is not specified. It means that the decision has been taken and it is not conveying in the assertion why the decision has been taken in that way.

```
<saml:Assertion ...>
   <saml:Conditions .../>
   <saml:AuthorizationStatement
      Decision="Permit"
      Resource="http://did.polito.it/m2170.php">
      <saml:Subject>
         <saml:NameIdentifier
            SecurityDomain="polito.it" Name="alioy" />
      </saml:Subject>
  </saml:AuthorizationStatement>
</saml:Assertion>
```

Figure 6.10: SAML authorization decision assertion

6.3.3 SAML producer-consumer model

These different kinds of assertions are related. By looking at the last example (authorization decision assertion) it is possible to notice that the subject is specified, but has the subject authenticated? If authentication is needed, it means another assertion is needed.

The assertions can be used individually, but they are quite often used together, which leads to the general schema presented in figure 6.11, in which there are several entities that at the same time are producing an assertion but also consuming (receiving) an assertion.

The schema contains three types of available assertion in the SAML part. The authentication assertion is from an authentication authority, the attribute assertion is from an attribute authority but notice that before deciding which is the attribute it is needed to know who is the requestor, and the Policy Decision Point is creating an authorization decision, but this could be based on the identity of who is requesting and maybe on the attributes (maybe "Lioy" is not enough, which Lioy? From which department?) and the authorization decision is used by the **Policy Enforcement Point**. Each of them (the authorities) has got a specific policy, for example the authentication method, and then there is some system entity (which can be a user, or a program) that performs an application request, the PEP will block (control) the access to the SP and the system entity in order to get access must perform authentication (maybe trough a Credentials Collector or an Authentication Protocol) and then the path showed in the picture will start.

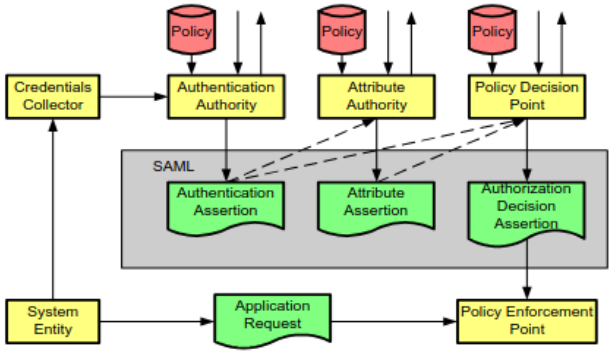


Figure 6.11: SAML producer-consumer model

6.3.4 SAML: protocol for the assertion

SAML is not only describing the format of the assertion itself but is also specifying how the request and the response are created (but not how they are transported).

The assertion will be contained in a SAML container and will be put in a Response. The request must be created for that specific type of assertion, and that is created between the relying party and the asserting party. It is used "relying party" because it relies upon the asserting party being a trusted source of information, so it relies on the assertion created by the asserting party to implement its own functionality.

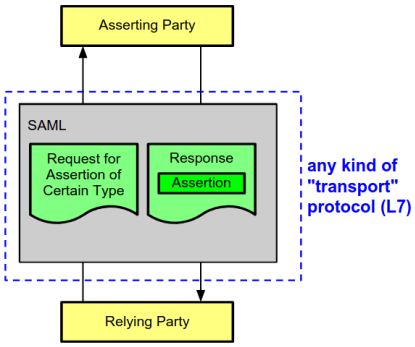


Figure 6.12: SAML assertion protocol

Request of authentication assertion

The request of authentication assertion is conceptually something of the kind "please, give me authentication information, regarding this subject, if you have any". It is assumed that requestor and the responder have a trust relation, because they speak about the same subject and the response is a sort of recommendation letter: "yes, it is possible to trust this user because I have authenticated it".

An example of request of authentication assertion is shown in figure 6.13. In the picture the example shows now `<samlp>` where p stands for protocol. The content will now be an AuthenticationQuery which asks for information about the specified user. It is something like: "please tell me if you have authenticated this guy that pretends to be alioy in the domain polito.it".

```
<samlp:Request
  MajorVersion="1" MinorVersion="0"
  RequestID="128.14.234.20.12345678" >
  <samlp:AuthenticationQuery>
    <saml:Subject>
      <saml:NameIdentifier
        SecurityDomain="polito.it" Name="alioy" />
    </saml:Subject>
  </samlp:AuthenticationQuery>
</samlp:Request>
```

Figure 6.13: An example of request of authentication assertion

6.3.5 Trust relationship

The assertion is part of a triangle between to the parties: user, service provider and the identity provider.

The one who accepts the assertion must trust the entity that generates an assertion. The trust relation is established by pushing or direct pull on a secure channel (e.g., TLS), which may be established with mutual authentication or at least the asserting party must authenticate itself. If a TLS channel is not used or if it is wanted to maintain a proof of the assertion, it is possible to use XMLsignature over the SAML object using a shared (MAC) or public key (real digital signature). The last case is a long-term solution while the TLS channel gives a temporary trust.

6.3.6 Binding SAML

SAML defines **"what"** to transport, while the binding defines **"how"** to transport it, i.e., a network protocol for SAML requests and responses.

- SAML/SOAP(Service oriented architecture protocol)-over-HTTP is the original binding in version 1.0.
- SAML 2.0 defines other bindings:
  - SAML SOAP binding (based on SOAP 1.1)
  - Reverse SOAP (PAOS) binding
  - HTTP Redirect (GET) binding
  - HTTP POST binding
  - HTTP Artifact binding
  - SAML URI binding

6.3.7 SAML Profiles

A SAML profile is a concrete manifestation of a defined use case using a particular combination of assertions, protocols, and bindings.

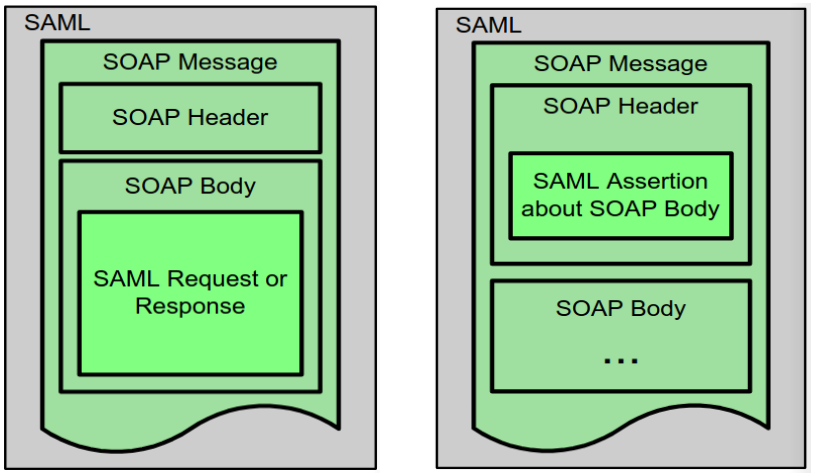
In practice a profile is like a software pattern (design pattern) which is a standard way to implement something relative to important information for a specific use case:

- Web browser profile is needed to implement Single Sign On (SSO) web
- SOAP profile is used for assertion about the SOAP payload – in case it is used

6.3.8 SAML and SOAP

Typically, we have SAML that contains a SOAP message with the SOAP header, SOAP body, and the request and response are carried inside this protocol. So it actually is layered in three protocols: SAML inside SOAP inside HTTP.

Figure 6.14b shows how the SOAP profile is structured.



(a) SAML over SOAP over HTTP (b) SOAP profile

Figure 6.14: SAML and SOAP

6.3.9 Web Browser Profiles

Web browser profiles, which are surely more relevant nowadays, operate under the following assumptions:

- A standard commercial browser and HTTP(S) are used.
- The user has authenticated with a local source site.
- The assertion's subject implicitly refers to the user.

When a user tries to access a target site:

- A small authentication assertion reference is included with the request, allowing the real assertion to be dereferenced.
- Alternatively, the real assertion is directly POSTed.

The choice between these methods is determined by the agreement between the Service Provider (SP) and Identity Provider (IdP).

6.3.10 SSO use case

SSO push use case

The procedure is shown in figure 6.15. The the client (browser) is trying to connect to a specific page of a web server (which is the SP). The resource is protected, so a redirect is received (HTTP codes that start with 300). The SP has a specific agreement with an Identity Provider like the case of PoliTo and is redirecting there with an authentication request, which is hidden inside the redirect. It means that when the client goes to the Identity Provider it is automatically transmitting this authentication request (number 3 in the picture) which is not created by the client, but it is a consequence of the redirect.

The Identity Provider is implementing some kind of authentication protocol (username and password, OTP, challenge response etc.) and will finally create the answer and it must return the answer to the Service Provider.

Assuming that it is successful, then will create for you a form that, when you hit the Submit button, will send you to the service provider. Inside this form, as hidden data, there will be the authentication result. When the client wants to go back to the Service Provider, it will automatically (involuntarily) transmit the authentication response (number 6, again it is not created by the client). This is the push use case (with reference to the delegated authentication), because the IdP is pushing the token (which is the SAML assertion) to the SP. Finally, the SP (if authN was successful) will provide the requested service to the client.

It is also named front-channel exchange because it directly uses the channel towards the SP.

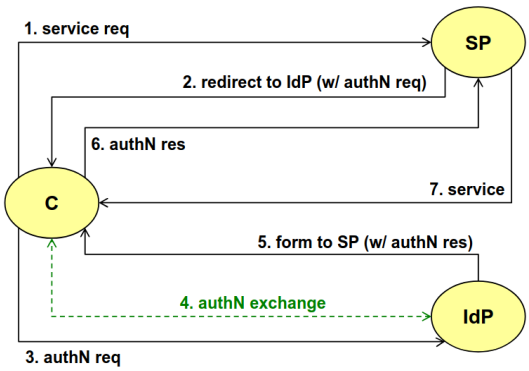


Figure 6.15: SSO push use case

SSO pull use case

In the SSO push use case all data are transmitted using the same port (there will not be an alternate port).

In this case, the first stages are the same as before: service request, redirect to IdP, authentication request (passed as a GET parameter), authentication protocol and then the difference. The step number 5 is changed: here there is now a redirect (a GET) with an artifact, which is a pointer to the result. The client will pass the artifact to the SP, which will need to open a direct channel to the IdP and to perform an authentication result request. Then the IdP will sent the authentication result and if it is positive the SP will provide to the client the requested service.

This case is named pull because in the response it is passed the pointer, and the SP needs to go and pull (take) the response from the IdP. This case could be better than the previous one because, for example, it does not require any signature. Assuming that the communication channel between SP and IdP is based on TLS, with TLS authentication for the IdP, then the SP can be sure of the result even without a signature of the signature. Of course, if the SP will need in the future to demonstrate the assertion, that could not be possible since it is not signed.

This use case is simpler (keys or certificates are not needed) but it takes a bit more time because it is needed to open a separate network channel. If a lot of authentications is performed, it is possible to keep the TLS channel always open, so it is just needed a RTT

to perform the request and get response, but there will not be the overhead of opening the channel. Finally, there is a problem on the IdP if there is an incoming firewall.

It is also named artifact binding or back-channel exchange because another channel is needed.

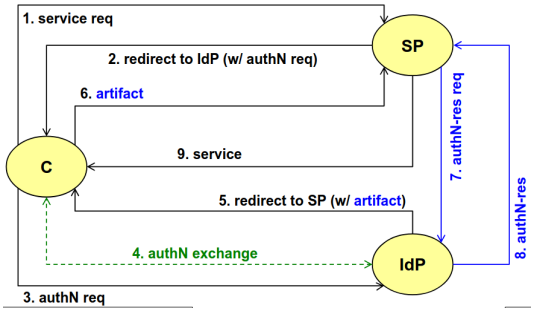


Figure 6.16: SSO pull use case

6.3.11 SAML SSO for Google Apps

This kind of architecture is being used by large providers such as Goggle to implement SSO for Google Apps.

Google Apps are applications that are hosted by Google, but the authentication and the authorization are managed by the company that developed the app. The company asks to Google to run applications on Google but with the possibility to manage the authentication (so the company does not want to use Google authentication) in order to keep control on who is accessing". This can be performed using SAML.

A company, basically a Google Partner, installs its own applications on Google (which will be just a service provider). The Partner wants to maintain control of the authentication and authorization part (basically the company wants to be an Identity Provider). The exchange is based on SAML-2.0 with XML signature: this is important because here there are two difference companies and Google wants to be sure that any mistake about the authentication cannot be on charge of Google. This is the typical case in which a signature is needed, typically a digital signature with X.509 certificate, because in case of any commercial discussion or legal discussion between Google and the company, Google wants to have a proof of why they permitted access to the application.

Basically, there is Google with the Application. All the users that want to use the application are redirected to the company. Then there will be the SAML assertion sent to Google which must be digitally signed, since that is the message that gives access to the application from Google to the user. If an assertion is accepted without any signature (or with a symmetric signature), then there is no way to prove that.

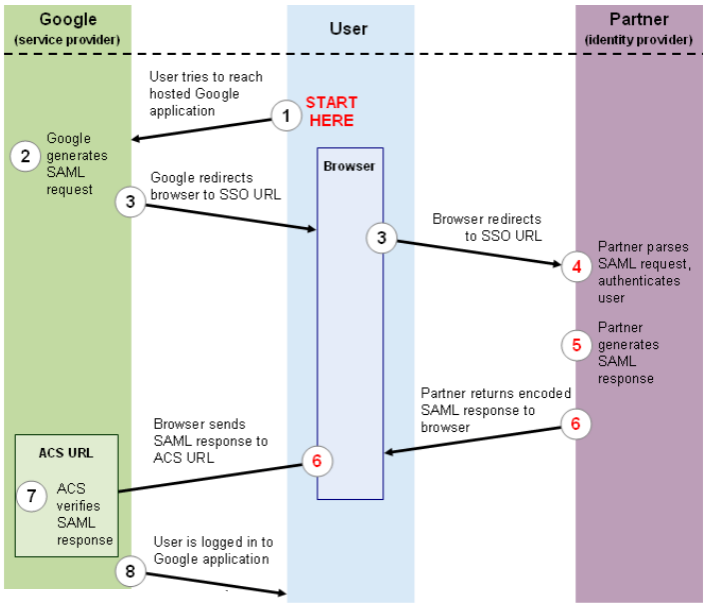


Figure 6.17: SAML SSO for Google Apps

The partner must provide to Google the URL of the single-sign-on service that can be also named the IDP, or the Authentication Server, as you prefer, and the X.509 certificate to verify the signature. Because the answer is signed with a real public key. Step three contains, in opaque mode, the URL of the Google service requested by the user. So the company knows not only, I have to authenticate, they can also perform authorization. Because this user is trying to use this service, the SML authentication request. And the URL where to send the response. And step six, in opaque mode, contains still the URL of the Google service to avoid a reply.

6.4 Federated Identity

So far, we have seen **Delegated Identity**, which operates within the same security domain. For example, in the case of PoliTo (where 100 servers delegate authentication to `idp.polito.it`) or Google's delegated authentication model, the authentication is handled by a specific Identity Provider (IdP) for a specific application. This setup always involves a one-to-one mapping: one application corresponds to one IdP. While the same IdP can serve multiple applications, there is still a 1:1 mapping.

On the other hand, **Federated Identity** enables multiple IdPs to interact with the same application service. This is a common scenario, such as when creating an account on a website where users can either create a new username and password or authenticate using Google or Facebook credentials. This federation allows authentication performed by an external service (outside the security domain) to be automatically recognized. A key advantage is avoiding *duplicate authentication*, which can be inconvenient for users.

Federated Identity extends federated authentication by incorporating **identity-related attributes**. SAML is frequently used for building federated identity systems because it supports both authentication and attribute assertion. Identity involves more than just

authentication; it includes *authentication + attributes* (e.g., name, surname, student ID, residence, etc.).

SAML is XML-based. While XML is *simple but heavy*, SAML is typically suited for PC or server-based environments. However, its complexity makes it challenging to use in lightweight or mobile contexts.

There is a juxtaposition between **SAML** and **OpenID Connect**. While SAML works well for server environments, it is less mobile-friendly. OpenID Connect, on the other hand, provides a similar architecture involving the client, Service Provider (SP), and IdP, but it uses **JSON** instead of XML and relies on the **REST protocol**. This makes OpenID Connect more suitable for mobile and web applications.

> **Beware:** *OpenID 1.0* and *OpenID 2.0* are **not** the same as OpenID Connect. The latter is based on **OAuth 2.0**, an IETF authorization framework. This distinction is critical, as the terminology surrounding OpenID can sometimes be confusing.

6.4.1 OpenID Connect

OpenID Connect(OIDC) is a delegated authentication system (the federated aspect rely in the fact that it support many IdPs).

It uses JSON data and REST protocol(which are native in smartphone environments), and it is not correlated to OpenID-2.0 but this is an identity layer put on top of Oauth-2.0 (IETF authorization framework). It should be the reverse: first authentication then authorization, but not in this case.

The user agent can be a normal browser o can be a mobile app, beware of the terms because the client is not the user agent, the client is the relying party (application server) that wishes to use OpenID-Connect for authentication.

In this schema the user has its mobile phone, and it is named User- Agent, which is connecting to an application server, which is the client, because it is the client of OpenID-connect, connecting to a server that will provide authentication, authorization, and attributes. That's why it is named client because it is the client for OpenID-connect.

The server S is not a single server, but it is a collection of servers, or as a minimum a collection of endpoints. It means that when there is a server with a certain network address, there may be several different ports or even if everything is on port 80 or port 443, when performing a GET or POST a path is specified, and that is an endpoint. There may be `/authenticate`, `/authorize`, `/attributes`: these are different entry points in a REST application.

Notice that the server, which is the OpenID-Provider (OP), which is conceptually similar to the IdP, has various endpoints:

- Authorization endpoint (AuthZ_{EP}: it is called authorization, but it performs authentication (confusing).
- Token endpoint (Token_{EP}), something that verifies if a certain token generated during the protocol is valid or not.
- UserInfo endpoint (UserInfo_{EP}), if the user has given the consent, then the client can retrieve information about the user.

User authentication

The login workflow is shown in figure 6.18 and is as follows.

We have the user interacting through a **user agent**, like a mobile app or a browser. The **client** here is the relying party (RP)—for instance, a web application or a service accessed via the app. The user begins by choosing to log in using an external IdP, such as Facebook or Google.

The client generates an **authentication request** to the IdP. This request is sent to the IdP's **authorization endpoint**, even though the goal is authentication (this dual-purpose endpoint can cause some confusion). The request includes details like the client ID, redirect URI, and requested scopes.

Upon receiving the authentication request, the IdP validates it. This validation ensures that:

- The client (RP) is registered with the IdP (a trusted relationship exists).
- The request is properly signed and formatted, verifying it genuinely comes from the registered client.

If the request is valid, the IdP generates an **authentication page**. Here, the user enters their credentials (e.g., username and password). The credentials are submitted back to the IdP's `/authenticate` endpoint, where they are verified.

Once authenticated, the IdP often presents the user with an **authorization page**. This page asks the user to consent to the RP accessing specific data (e.g., profile, email). For example, when using Google or Facebook login, you might see a message like, "This app will receive your name, email, and profile picture."

If the user grants consent, the request is finalized at the `/authorize` endpoint. The IdP issues an **authentication response**, typically as an ID token (a JSON Web Token, or JWT), which contains the authenticated user's information. This token is sent back to the client via the redirect URI specified earlier.

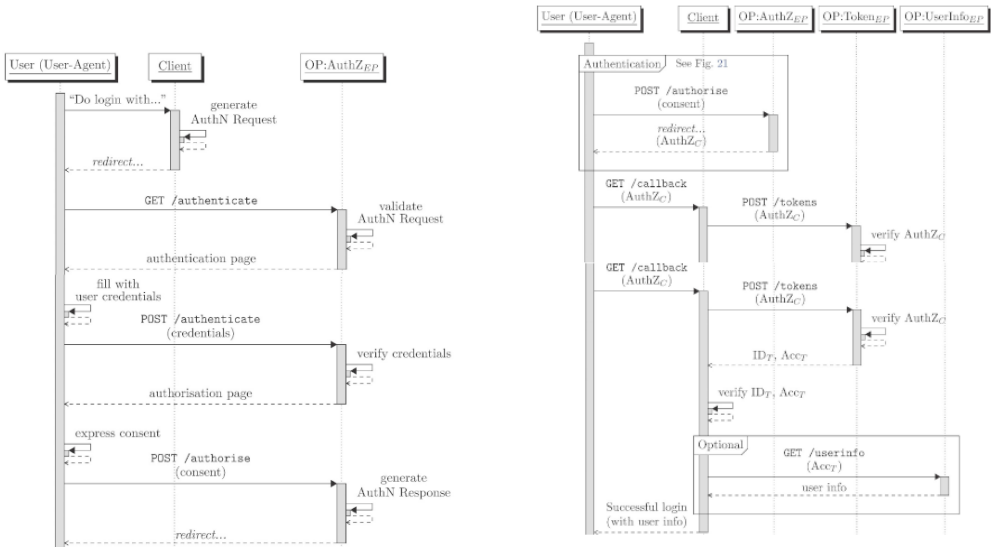
Login with token

The login workflow is shown in figure 6.18 and is as follows.

It's assumed that the user has already authenticated with the IdP and has an active session. After the redirect from the Authorization Endpoint (AuthZEP), a `GET` request is sent to a `/callback` endpoint, passing the received information. The Authorization Endpoint provided a token in response. This token is then handed to the client, which can use it at the Token Endpoint (TokenEP) for validation. This demonstrates that the user consented to transfer their data to the client.

The Token Endpoint verifies the token, and if valid, it returns **IDT** (Identity Token) and **ACCT** (Accessory Information). The client verifies the IDT, and if accessory information is present, the **UserInfo Endpoint** (UserInfoEP) may optionally be accessed to retrieve additional user details.

Finally, a successful login occurs, and the client is provided with the user's identity and any additional information.



(a) User authentication

(b) Login with token

Figure 6.18: OpenID Connect

Trust, security and discovery

All the messages are authenticated with digital signatures, that requires registration of the public keys among the various actors. All the messages are protected via secure channel (TLS) but this is not a real federation, it is just the fact that it is possible to use more than one. There is a proposed service, WebFinger, to discover the OpenID Providers but that works only if the provider registered itself with WebFinger, so it is not much used. But in any case, OpenID Connect is much used, OIDC providers: Google, Facebook, Salesforce.

OIDC standard claims

When additional information about user is requested, it is possible to ask for:

- profile category: subject (ID at the issuer), name (full), given_name, family_name, middle_name, nickname, preferred_username, gender, birthdate, zoneinfo, locale, profile (URI), picture (URI), website (URI), updated_at (last update at the issuer)
- email category: email, email_verified (Boolean)
- phone category: phone_number, phone_number_verified (Boolean)
- address category: address

When a client is requesting a claim, it may request a single item or a whole category (e.g., profile category or only family_name). Apart from the ones with _verified=TRUE all the others are self-asserted by the user and hence unreliable. This is the bad part: the user can be a fake user. Custom claims can be created, but they have a restricted audience.

JWT for claims – example of ID token

There is: the subject, the issuer (the OpenID connect at PoliTo), the audience (which client, it can be also www.unito.it because that server can be a client to accept the user

of PoliTo rather than forcing them to have an account at that website), the nonce, the authentication time, the authentication context class reference (it is a way to explain which kind of authentication was performed, of course some values must be agreed which could be PoliTo, loa stands for level of assurance and hisec could be high security but there should be a meaning, a definition behind that), the issued at (at what time the token was created), the expiration (when it will expire).

```
{
  "sub"       : "alice",
  "iss"       : "https://openid.polito.it",
  "aud"       : "didattica-polito-it",
  "nonce"     : "n-0S6_WzA2Mj",
  "auth_time" : 1604841420,    # 11/08/2020 13:17:00pm (UTC)
  "acr"       : "polito.loa.hisec",
  "iat"       : 1604841421,    # 11/08/2020 13:17:01pm (UTC)
  "exp"       : 1604842021     # 11/08/2020 13:27:00pm (UTC)
}
```

Figure 6.19: Example of ID token

6.5 eIDAS

eIDAS, stands for *electronic IDentification, Authentication, and trust Services for electronic transactions in the internal market*, and it's based on the EU regulation 910/2014.

The key point is that in each European country there is a different identity system (in Italy SPID) but in the same way in which the identity card, the driving license, the passport is valid also on abroad, the European Union wanted to allow the use of the electronic identity when accessing services in a different European country. It took a lot of years (more then 10 years) and it works and it is used all over Europe.

Initially, the eIDAS electronic identification (eID) infrastructure was voluntary. However, starting from September 2018, it became compulsory for EU public services. Adoption by the private sector remains optional but is strongly encouraged.

The purpose is to boost confidence and trust towards digital world by adopting the following principles among others:

- Mutual acceptance of national e-ID: all the members of EU are trusted
- Common framework for secure interaction between citizens, companies, and public administration
- Technological neutrality of requirements Required to not restrict to specific solutions (some countries use smartcards, others OTP, etc.). All the solutions are mutually recognized, but it is not possible to say all the methods are equivalent.
- Level of trust in national electronic identity can be defined by a certain e-ID quality level
- Country-specific supervision organizations to verify the Regulation adoption and interact with the European Commission (e.g., for data privacy)

There were various implementing acts made by Commission Implementing

- 2015/296 (24 February 2015): eID procedural arrangement for MS(Member state) cooperation
- 2015/1501 (8 September 2015): interoperability framework
- 2015/1502 (8 September 2015): technical specifications for assurance levels for electronic identification means
- 2015/1984 (3 November 2015): formats and procedures for notification

6.5.1 Pan-European eID

Notice once again that identity is not just authentication, it is **authentication plus attributes**. The very important point is the fact that with eIDAS there are certified attributes. In OpenID Connect there are a lot of information, but only phone number and email can be trusted, since all the others are self-declared by user. On the contrary, in eIDAS, since the information is provided by the government it is trusted.

So, again in eIDAS there will be e-identity with authentication + certified attributes

- Set of certified European attributes
- Lexicon (multilanguage attribute names): it is a problem, since all the countries in the EU must agree (due to different alphabets and characters).
- Syntax (possible values)
- Semantics e.g., surname in Italy is just one given by the father but in Island there are two surnames, since each person keeps both surname of father and mother and they can decide to use one or the other. So, it is needed to understand the different meaning that even simple things may have in different countries.

It is possible to use various authentication credentials such as reusable password, one-time-password, cellphone, software certificate, smart-card since the system is technology neutral. The point is that it is used in a transparent way and with legal value (according to the citizen's country).

6.5.2 Adaptive security and privacy protection

Everything is accepted but each authentication method it is assigned various authentication levels and the authentication level is attributed through the LOA (Level of Assurance) which means how much the result (on the authentication method) can be trusted. LOA consists in three different levels: substantial, medium, high. This is not only related to the cryptographic strength of the authentication technique, but also on the strength of the identification process, because e.g., a country is using a smart-card with a PKC asymmetric challenge-response which is the strongest authentication, but how does the smart-card is given to the citizen? If there is no identification of the citizen, then it is bad. In the end, the LOA tests both things in order to give a level to the whole procedure.

When accessing a service there may be a mismatch (e.g., a service is for transferring money so an High LOA is wanted) if the procedure provides a Medium LOA but the service requires an High LOA, so authentication may fail.

For privacy protection and localization, the user talks with its own country and before transferring attributes abroad, it must provide explicit consent for the required attributes. The attributes are managed end-to-end, which means that attributes are transferred from the government of a country directly to the Service Provider in the other country: eIDAS infrastructure itself does not store any personal data. There is minimal disclosure, in the sense that on the contrary to OpenID Connect in which everything can be requested, in this case it is applied the Need-To-Know principle, e.g., some services are reserved to people with age major than 18: in this case the birth date is not requested, since the service is not interested to that information but the service will ask to the system if the user is above 18 or not.

6.5.3 eIDAS terminology

The MS is the Member State. It can be:

- Sending MS: it is the MS whose eID scheme is used in the authentication process, and sending authenticated ID
- Receiving MS: it is the MS where the RP requesting an authentication (service that citizen is trying to access) is established

Then there is an eIDAS-Connector, which is a node requesting a cross-border authentication and the eIDAS-Service which is a node providing cross-border authentication. Unfortunately, there are two implementations.

One big country in Europe did not want to agree with the others, so 99% of countries implements the eIDAS- Proxy-Service (an eIDAS-Service operated by the Sending MS and providing personal identification data) schema, while one country is implementing it using eIDAS-Middleware-Service (an eIDAS-Service running Middleware provided by the Sending MS, operated by the Receiving MS, and providing personal identification data) schema.

6.5.4 eIDAS infrastructure

Imagine to be an Italian citizen with SPID identity that needs to access a SP in Sweden. The SP will ask to the citizen to choose between the Swedish identifier or eIDAS for authentication. The citizen will of course choose eIDAS, so it will be redirected to the Swedish eIDAS Connector (which connects that country to the rest of the infrastructure). It will ask to select the country of the citizen and will then redirect to the Italian eIDAS Service which is implemented as a proxy. At this point the proxy will provide to the citizen all the information that Sweden requested (name, surname, date of birth) and will ask for a preliminary consent (otherwise the process will stop). Then, since in Italy there are different systems for identity (SPID, but with many providers such as Poste Italiane and it could be possible to use the electronic identity card) the citizen must choose which electronic identifier (e-ID) to use. In this way the citizen will be redirected to that specific provider, which has got the identity and the attributes (IDP + AP). At this point, the authentication will be performed according to the system selected and there will be a final consent to transfer abroad the specified information (again name="Antonio", surname="Lioy", etc.).

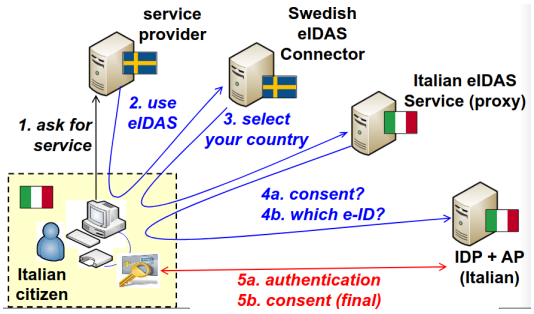


Figure 6.20: eIDAS infrastructure

6.5.5 eIDAS technical specifications

Technical specifications are evolving over the time: Version 1.0 (26/01/2016), 1.1 (16/12/2016), Version 1.2 (27/09/2019). They are all publicly available on the European website. The protocol is based on STORK1 and then STORK2, that are developed and tested infrastructure and finally after STARK2 everything was adopted as the official eIDAS Infrastructure. It is similar, but not compatible, e.g., in the real implementation it has been added the encryption of authentication response to guarantee privacy against network sniffing. It covers, but only for the international part, i.e., MS-to-MS. Looking the previous picture, eIDAS in the part 3 (Connector exiting from one country and the proxy accepting request from the other country). Specification is only for this part (how SPID works in Italy it is decided by Italian Government, ...). eIDAS is end-to-end conceptually, in the sense that the response will directly transfer to the SP but as a protocol, it is only for connecting the end-points at the borders of the various countries. It covers, only for the international part, the:

- Interoperability architecture
- SAML message format
- SAML attribute profiles
- Cryptographic requirements

eIDAS Minimum Data-Set

The eIDAS regulation specifies a minimum data-set that must be supported by any eIDAS node to enable cross-border authentication. This data-set includes attributes for both natural and legal persons.

For **natural persons**, the data-set includes:

- **Mandatory Attributes:**
  - `PersonIdentifier`, `FirstName`, `FamilyName`, `DateOfBirth`
- **Optional Attributes:**
  - `BirthName`, `PlaceOfBirth`, `CurrentAddress`, `Gender`

For **legal persons**, the data-set includes:

- **Mandatory Attributes:**
  - `LegalName`, `LegalPersonIdentifier`
- **Optional Attributes:**
  - `LegalAddress` , `VATRegistration` , `TaxReference` , `BusinessCodes` , `LEI` (Legal Entity Identifier) , `EORI` (Economic Operators Registration and Identification) , `SEED` (System for Exchange of Excise Data) , `SIC` (Standard Industrial Classification)

Security Requirements

- **SAML request** (no personal data) **MUST** be signed with a digital signature, but no encryption is required.
- The request may be transmitted via:
  - **HTTP Redirect**: For example, a 302 GET with the request in a parameter. This is preferred if the size of the request does not exceed the maximum URI length (256 characters).
  - **HTTP POST**: For example, a form with POST and the request in a hidden field. This is preferred if the request is much larger.
  - Note: The size of the request can change based on the data requested. For instance, authentication requests with attributes will make the request larger.
- **SAML response** (containing personal data) **MUST** be signed with a digital signature and must include an `EncryptedAssertion` with:
  - One `AuthenticationStatement`
  - One `AttributeStatement`

  The response itself is not encrypted, but portions of it may be encrypted as allowed in SAML.
- The response is transmitted via **POST binding** to the ACS (*Assertion Consumer Service*) of the connector.
- The connector metadata must contain:
  - The **encryption certificate** and **ACS URI**.
  - For example, when Italy sends back encrypted data, it needs to know the public certificate and the ACS URI to perform a redirect after authentication. These details are included in the metadata.

All communication channels must use TLS version $\geq$ 1.2, with qualified web certificates. The supported TLS ciphersuites are:

- ECDHE+ECDSA or ECDHE+RSA or DHE+DSA (accepted for backward compatibility)

- AES_128_CBC/GCM with SHA256 for encryption
- AES_256_CBC/GCM with SHA384 for encryption
- SHA1 MAY be accepted but only due to restrictions of the client browser

Other Requirements for TLS:

- For **ECDH keys**, the minimum key size is 256 bits, while for **DH keys** the minimum is 2048 bits.
- The **TLS compression SHOULD NOT** be used.
- **TLS heartbeat** and **Session Renegotiation MUST NOT** be used.
- If a **CBC-based cipher suite** is used, first encrypt and then authenticate the data (e.g., use the Enc-then-MAC extension).
- **DON'T** use a truncated HMAC (e.g., unsupported that extension).

Requirements for SAML:

- **Data encryption** only via `AES-128/256-GCM`, but `AES-192-GCM` may also be used.
- **Key encryption** with `RSA-OAEP-MGF1P` or `RSA-OAEP` (3072 bits minimum key length).
- **Key agreement** via `ECDH-ES`:
  - `ES` is *Ephemeral-Static mode* (i.e., the recipient has static DH parameters while the sender creates ephemeral ones). The destination publishes in the metadata what its own DH parameters are, while the sender will create the ephemerals.
- **Key wrapping** via `KW-AES-128/256`.
- **Digital Signature** via `RSAPSS` (3072 bits minimum key length) or `ECDSA` (256 bits minimum key length) with `SHA-256/384/512`.
- The trusted EC are: `BrainPoolP256r1, BrainpoolP384r1, BrainpoolP521r1, NIST Curve P-256, NIST Curve P-384, NIST Curve P-521`.

6.5.6 eIDAS 2.0

eIDAS 2.0 introduces a new schema based on the **EUDI** (EU Digital Identity) wallet. It implements the concept of **SSI** (Self-Sovereign Identity). The EUDI wallet will provide identification and authentication, as well as the verifiability of the validity of the evidence by third parties throughout Europe. It ensures secure storage and presentation of verified identities and their data, and enables the generation of qualified electronic signatures.

6.6 SPID

SPID is the Italian Public System for Digital Identity. It's architecture is based on three main components:

- **Identity Provider (IdP)**: the entity that provides digital identity to the user. It is responsible for the authentication process and the enrollment of the user. It also returns the user identity plus some basic attributes(not all of them).
- **Service Provider (SP)**: the entity that performs access control based on the identity and attributes.
- **Attribute Authority (AA)**: the entity that provides the user's attributes based on the identity. This is necessary there are some attributes that are not controlled by the government, for instance the enrollment to the *Ordine degli Ingegneri*.

6.6.1 Protocols and data

SPID uses SAML version 2 with the web browser SSO profile for data exchange. The caveat is that each entity prodives its own metadata, containing:

- its X.509 signature certificate, which may be self-signed! This is possible because the metadata is public and can be accessed from a goverment controlled website.
- its protocol endpoints
- various other data (e.g. supported attributes)

The metadata are signed by the Agency for Italian Digitalization in detached form. This means that they will be the target of electronic signatures. The official metadata registry containing all the service providers and identity providers can be found at `https://registry.spid.gov.it/metadata`. The specification in case of spid are a little bit behind the eIDAS, because the basic signature algorithm is RSA-2048 with sha-256, while eIDAS requires RSA-3072 with sha-384.

In case of an HTTP redirect, the SAML request is passed in the query string of a GET. But this has a limit of 256B for the basic URI, but typically length of 2kB up to 8kB are supported. Since POST requests are not to be used, the SAML request is compressed with DEFLATE algorithm and not signed with SMLDSIG, but the SigAlg parameter is used to specify the signature algorithm, in a Base64 url encoded form of the whole query string. When using HTTP POST, then we permit normal SAML with normal XML signature.

There are many basic attributes, and each SP may require a specific attribute set, which groups them. A common example of an attribute set could be (name, familyName, fiscalNumber, email).

There are also different authentication levels:

- SPID Level 1: the user is authenticated with a password
- SPID Level 2: the user is authenticated with a password and OTP or password and an app
- SPID Level 3: this level requires 2FA with asymmetric authN via a secure device, for example a password with an app for remote digital signature and a Layer 3 pin

6.6.2 SPID evolution

The Attribute Authority is not yet implemented, but it's not for a technical issue, but an organizational one: the government has not decided who is authorized to create some attributes.

There's also a requesto make SPID available not only with SAML, but also with OIDC, to make it more compatible with the rest of the world. This is compulsory for the IDPs since May 1, 2022. For the service provider, it is optional to support that.

6.7 Q&A

Question 1

what role does the Authentication Server (AS) play?

- × provides authorization decisions for the RP
- ✓ performs authentication on behalf of the Relying Party (RP)
- × verifies the identity of the RP
- × issues a session timeout for the RP

Question 2

- × relaying party is A. the entity that directly authenticates the user
- × the server that stores user credentials
- ✓ the entity that trusts another server to handle authentication
- × the client application used by the user

Question 3

the Authentication Server (AS) interacts with the client using specific protocols to perform authentication. What amongst the following can be used for the authentication process

- × HTTP
- ✓ OAuth
- × FTP
- × SMTP
- ✓ SAML
- × SOAP

Question 4

how does the Authentication Server (AS) typically communicate the authentication result to the Relying Party (RP)?

- × by issuing an authorization code
- × by sending a cookie
- ✓ by providing a ticket or assertion
- × by creating a session ID

Question 5

Which of the following scenarios best represents a delegated authentication setup in a Single Sign-On (SSO) environment

- × the user logs in separately to each application with the same credentials
- ✓ the user logs in once, and a central AS authenticates them for multiple RPs
- × each application has its own AS that verifies the user separately
- × the user must verify their identity with each RP individually

Question 6

Which format is commonly used to provide assertions in delegated authentication systems like SAML?

- × JSON Web Tokens (JWT)
- × Hypertext Markup Language (HTML)
- ✓ Extensible Markup Language (XML)
- × Simple Mail Transfer Protocol (SMTP)

Question 7

Which of the following is a key benefit of using delegated authentication for a Relying Party (RP)?

- × reduced dependency on external servers
- ✓ enhanced data privacy for user credentials
- × increased complexity in user management
- × independent authentication for each RP

Question 8

what does a SAML assertion represent?

- ✓ declaration of a fact about a subject, like user information
- × a statement about of how often a user logs in
- × a protocol for secure email communication
- × a form of encryption used for securing passwords
- × a method of encoding data in XML format
- ✓ declaration made by an issuer about a different subject

Question 9

Which of the following is not a standard type of SAML assertion?

- × authentication assertion
- ✓ Service type assertion
- × authorization decision assertion
- ✓ public key certificate assertion
- × attribute assertion
- ✓ token assertion

Question 10

When an Identity Provider (IdP) issues a SAML assertion, who is typically the recipient that relies on this assertion?

- × user's web browser
- ✓ Relying Party (RP)
- ✓ Service Provider (SP)
- × DNS server
- × certificate authority
- × password manager

Question 11

Why does SAML use XML as the standard format for its assertions?

- ✓ XML provides a flexible, readable format for exchanging information across systems
- × XML is universally adored by data specialists
- × XML is the default because JSON was too complicated
- ✓ XML allows for custom-defined tags, which means SAML can extend its structure to include additional fields or attributes as needed without breaking compatibility with existing systems
- × XML prevents other formats from being easily implemented
- × XML encodes information only in binary for extra security
- ✓ XML allows SAML to be platform-independent and easily parsed by different systems

Question 12

In policy-based access control (PBAC), what is the primary function of the Policy Enforcement Point (PEP)?

- ✓ enforces access to a resource based on the policy decision
- × collects user attributes for policy decision-making
- × decides whether a policy is applicable to a specific access request
- × acts as a backup in case the Policy Decision Point (PDP) fails
- × protects resources by ensuring they're only accessed when policies align with user attributes

Question 13

Which of the following best describes the role of the Policy Decision Point (PDP) in PBAC?

- × enforces access controls based on predefined policies
- × collects and provides the necessary policies for the reguested access
- ✓ evaluates all information (policy, subject, resource, and access type) to determine if access should be granted
- × monitors user behavior and access patterns for anomaly detection
- × checks user permissions, but only on days when its not overloaded
- × none of the above

Question 14

What is the primary purpose of a SAML binding?

- ✓ to specify the underlying protocol used to transport SAML reguests and responses
- × to define how SAML assertions are signed for security purposes
- × to describe the format of SAML messages exchanged between systems
- × to establish a fallback protocol for non-compliant systems
- × to ensure SAML messages are transported using the fastest possible route

Question 15

Which of the following is not a valid SAML binding?

- × SAML SOAP binding
- × Reverse SOAP (PAOS) binding
- × HTTP POST binding
- ✓ FTP-over-SAML binding
- × HTTP redirect binding
- ✓ SAML-over-TLS binding

Question 16

In the SAML SOAP binding, what role does SOAP play?

- × It formats the SAML reguest/response messages in XML for transmission
- × It determines the security of the SAML assertion being transported
- ✓ It wraps SAML messages inside a SOAP envelope over HTTP
- × It acts as an intermediary to deliver SAML assertions across different servers
- × It ensures messages are "clean and shiny" before they'Tre sent

Chapter 7

Secure (signed) Electronic Documents

Let's start, as always with some definitions.

> An **electronic document** is any **electronic representation** of a **document**, while a **digital document** is a **digital representation** of a document.

Secure electronic documents must satisfy the following properties:

- **Security:** Ensure data integrity and authenticity using a digital signature.
- **Readability:** Use an open format for content and signature interpretation. This is crucial for long-term archival: for example, we are having issue reading the data of the first Apollo mission.
- **Long-term Archival:** Ensure readability and verifiability over decades.

7.1 Formats of Signed Documents

Signed documents can take different formats:

- **Enveloped Signature:** The signature is embedded within the document in a specific location (e.g., PDF).
- **Enveloping Signature(another case):** The document is embedded within the signature structure (e.g., PKCS#7).
- **Detached Signature:** The signature is separate from the document (e.g., PKCS#7). In this case the correlation between the signature of the document(for instance, using a pointer).

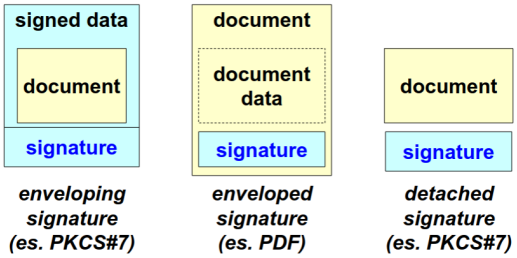
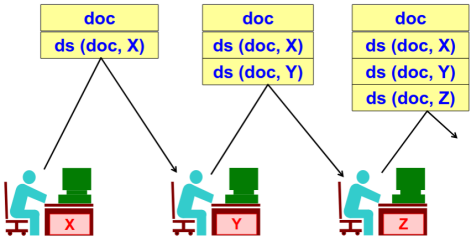


Figure 7.1: Different formats of signed documents.

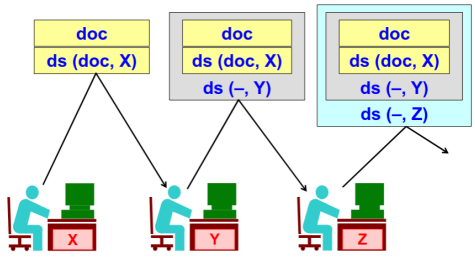
7.2 Multiple signatures

In most case, multiple signatures are needed, for example in an office there is typically a specific document workflow: an purchase order may need to be validated and approved by different people, each of which will sign the document in sequence(see figure 7.2a).

In those scenarios, the signatures can be either parallel or sequential. In case of independent signature, each signature is applied to the original document, meaning that the order of the signature is irrelevant since each signature is indipendend from one another, while in case of sequential signature, each signature is applied to the previous signed document.



(a) Parallel signatures in a single document.



(b) Sequential signatures in a single document.

7.3 Digital Signatures in PDF Files

When a PDF needs to be signed, it is at first converted in a byte stream, while also reserving a specific place for the signature. This allows to embed any kind of file format (even a movie) in a pdf file.

The data to be signed is identified by two intervals:

```
/ByteRange[ offset1, length1, offset2, length2 ]
```

This divides the document in two parts, both of which are for the data to allow the document to seem *contiguous*.

Then, a digest of the data(signature excluded) is compute, for example using sha-256, which is afterward encrypted with the signed private key, via RSA-2048 for instance.

The signature value is encoded as a PKCS#7 detached signature, and the hex encoded signature value is inserted in the reserved space, along with the necessary padding(made up

of zeroes). This means that even if technically the signature is detached from the document, in reality it is enveloped in the document.

If there's any leftover space in the reserved space, it is filled with zeroes.

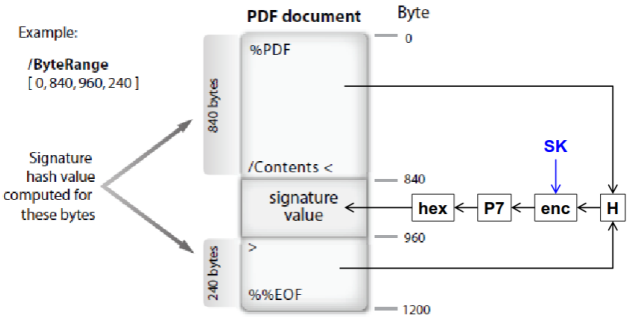


Figure 7.3: Digital signature in a pdf file

7.4 Adobe Acrobat Signature Formats

Adobe Acrobat supports the following signature formats:

- `adbe.pkcs7.detached` (default).
- `adbe.pkcs7.sha1`.
- `adbe.x509.rsa.sha1`.
- `ETSI.CAdES.detached` (since Acrobat 10.x).

but also supports custom signature handlers, while retaining the same structural format.

7.4.1 Adobe Acrobat signature algorithms

Adobe Acrobat provides support for various signature algorithms, which evolve based on the Acrobat and PDF version. The details of supported message digest algorithms and encryption or signature methods are as follows:

- **4.x–5.x** / PDF 1.3: MD5, SHA1
- **6.x** / PDF 1.3: MD5, SHA1
- **7.x** / PDF 1.6: MD5, SHA1, SHA256
- **8.x–9.0** / PDF 1.7: MD5, SHA1, SHA256, SHA384, SHA512, RIPEMD160
- **9.1–9.x** / PDF 1.7: MD5, SHA1, SHA256, SHA384, SHA512, RIPEMD160
- **10.x and later** / PDF 1.7: MD5, SHA1, SHA256, SHA384, SHA512, RIPEMD160

Encryption or Signature Algorithms

- **4.x–5.x** / PDF 1.3: RSA up to 1024-bit
- **6.x** / PDF 1.5: RSA up to 4096-bit
- **7.x** / PDF 1.6: RSA and DSA up to 4096-bit
- **8.x–10.x** / PDF 1.7: RSA and DSA up to 4096-bit
- **11.x and later** / PDF 1.7:
  - RSA and DSA up to 4096-bit
  - ECDSA P256-SHA256, P384-SHA384, or P512-SHA512 (available only with `adbe.pkcs7.detached` and `ETSI.CAdES.detached`)

DSA supports only SHA1 (by design) and can only be used in `adbe.pkcs7.detached`.

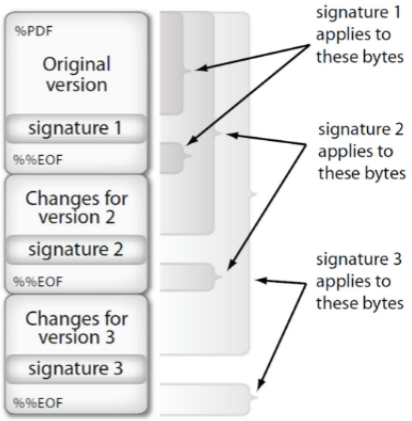
7.4.2 Adobe Acrobat multiple signatures

Since embedding another signature in the same document would require overwriting the existing signature, Adobe Acrobat supports multiple signature by associating them with incremental updates. This is intewresting because a PDF usually keeps a history of the changes made to the document.

7.5 Electronic Signature (ES) and Advanced Electronic Signature (AES)

Electronic signatures are so important nowadays that the EU has decided to standardize their format(not how they are created). An ES is defined as:

> Data in electronic form, attached to or logically associated with other electronic data, serving as a method of authentication.



This definition is quite broad to allow for different implementations and technologies.

On the other hand, an AES satisfies additional requirements:

- Uniquely linked to the signatory, typically achieved by a asymmetric key pair.
- Capable of identifying the signatory, by using a public key certificate.
- Created under the signatory's sole control, by using a device in possession of the signatory, like a smart card.

- Detects any subsequent changes to the signed data, achieved by hashing the data.

7.6 Qualified Certificate (QC)

A Qualified Certificate (QC) is a PKC that certifies the identity of a person. It includes specific attributes and information to ensure its applicability for intended purposes. The main components of a QC are:

- An explicit indication that it was issued as a Qualified Certificate.
- The name of the signatory or a pseudonym, clearly identified as such.
- Provision for specific attributes of the signatory, relevant to the certificate's intended purpose.
- Any limitations on the scope of the certificate, if applicable.
- Limits on the value of transactions, if applicable.

The QC is defined and profiled in `RFC-3739` under the IETF-PKIX framework.

7.6.1 Qualified Electronic Signature (QES)

A QES is:

- An AES based on a Qualified Certificate (QC).
- Created by a secure signature creation device.
- Legally equivalent to a handwritten signature.

Furthermore, notice that a QES is surely an AES, but the opposite is not necessarily true.

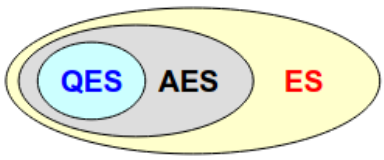


Figure 7.4: Electronic Signature (ES), Advanced Electronic Signature (AES), and Qualified Electronic Signature (QES).

7.7 Legal effects

Member States shall ensure that an electronic signature is not denied legal effectiveness and admissibility as evidence in legal proceedings solely on the grounds that it is:

- in electronic form, or

- not based upon a qualified certificate, or
- not based upon a qualified certificate issued by an accredited certification-service-provider, or
- not created by a secure signature-creation device

7.8 ETSI Standards for electronic signature

The European Union created the definitions for what is legally valid, and then entrusted the ETSI to create the technical standards for the implementation of those definitions.

ETSI standards for electronic signatures include:

- **CAdES:** CMS Advanced Electronic Signatures(originally CMS was for PKCS#7).
- **XAdES:** XML Advanced Electronic Signatures.
- **PAdES:** PDF Advanced Electronic Signatures.
- **ASiC:** Associated Signature Containers for detached signatures.

7.8.1 CAdES Formats

CAdES is a raw signature format, being also a enveloping signature one. The basic CAdES format is usually for electronic signature, even though, as you can see for figure 7.5, in the example is actually shown a digital one, because the signature envelops the policy identifier, used to specify the purpose of the signature.

Other elements that are also present are:

- a **secure timestamp**, generated by timestamping authority, over the digital signature. This is important, because in case the certificate is revoked, then we know that this signature was created before revocation
- a **certificate chain**, which is the chain of the certificates as well as the fill revocation reference(ie: a simple pointer), needed for long term archival.

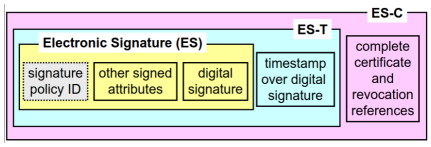


Figure 7.5: CAdES Formats.

7.8.2 Extended ES(ES-X)

The Extended Electronic Signature (ES-X) formats are recommended in scenarios where CA certificates may be compromised. These formats provide additional timestamping mechanisms to enhance trust:

- **ES-X-Timestamp (type 1)**: Extends ES-C by adding a timestamp (TS) over the entire ES-C structure. This format is particularly useful when the OCSP is employed. This is quite useful because if the reference for revocation is an OCSP signature, one also need verification of the OCSP signature.
- **ES-X-Timestamp (type 2)**: Extends ES-C by adding a timestamp over only the references to the certificates and revocation information. This format is better suited for cases where CRLs are used.

7.8.3 TSL (Trust Service Status List)

The Trust Service Status List (TSL) provides a structured and signed list of Trust-Service Providers (TSPs) and their associated services. Key aspects of the TSL include:

- **Content:**
  - References to national TSLs.
  - A list of TSPs and their services, such as certification, revocation, and time-stamping.
- **State of TSPs:**
  - Information on the status of each TSP, such as supervised, suspended, or revoked.
  - A history of the state of each TSP.
- **Additional Information:**
  - Details about the schema and the schema operator.
  - Distinction between accredited and non-accredited TSPs.

Relevant tools include the TSL browser available at `http://tlbrowser.tsl.website/tools/index.jsp` and the EU's signed TSL at `https://ec.europa.eu/tools/lotl/eu-lotl.xml`.

7.8.4 Other ETSI ES Formats

The ETSI defines several advanced electronic signature formats tailored to specific use cases and technologies. These formats include:

XML Advanced Electronic Signatures (XAdES):

- Based on `XML-dsig`.
- Standardized under ETSI TS 101 903.
- Profiles for XAdES are detailed in ETSI TS 102 904.

PDF Advanced Electronic Signature Profiles (PAdES):

- Specifically designed for the ISO-32000 format (PDF).
- ETSI TS 102 778 series provides comprehensive specifications:

- **TS 102 778-1:** Overview of PAdES.
- **TS 102 778-2:** Basic PAdES features.
- **TS 102 778-3:** Enhanced PAdES (BES, EPES).
- **TS 102 778-4:** Long-Term Validation (LTV) support.
- **TS 102 778-5:** XML content integration.

7.8.5 ETSI ES: detached formats

The ETSI defines the **Associated Signature Containers (ASiC)** format for managing detached electronic signatures. This format enables the association of e-documents with their corresponding detached signatures or timestamps in a single container.

ASiC Overview:

- Based on the `ZIP` format.
- Designed to associate electronic documents with detached signatures (e.g., CAdES or XAdES) and/or timestamps.

Standards:

- **ETSI TS 119 162-1:** Defines the building blocks and baseline containers for ASiC.
- **ETSI TS 119 162-2:** Specifies additional ASiC containers with extended functionalities.

7.9 The "macro" problem

As previously explained, the content of the PDF to be signed is converted in a bytestream, and then signed. So if the bytecode is some javascript, the code itself will be signed, and not the result of the execution. This means that the signature will be valid even if the code is malicious. An this is a problem, because in some case the macro will be evaluated and the macro code will be removed.

For exaple the Politecnico thesis archive requires a PDF/A format, which is vulnerable to this kind of attack.

7.10 WYSIWYS

What You See Is What You Sign (WYSIWYS) ensures that the displayed document content is what is actually signed. Think about very small clauses, or even the ones written with the same color of the background. In this situations, this is an highly desirable property, and it's not intrinsic to the digital signature itself.

Since this is not technically achievable by a digital signature alone, from a legal standpoint, the signature may be considered invalid.

7.11 The magic of Poste Italiane

In the end, we use software to create a digital signatures, which is usually buggy as a result of the complexity of the task.

For example, in 2003, Postecom, which is the branch of Poste Italiane that is managing certificates and the signature tool, which at the time was *Firma&Cifra 1.0*, had a bug which accepted a root certificate that was inside the PKCS#7 document, with no check about the root CA trustworthiness. This behaviour allowed for the creation of a valid certificate for *Arsene Lupin*, by even using a fake root certificate for Poste Italiane, but at the time that didn't have a CRL available so the signature was valid but the verification was not.

Chapter 8

Raw and XML digital signature and encryption

On the previous chapter we have seen digital documents, which biggest security part is the signature. For that, PDFs are using the PKCS#7 standard, and XMLs are using the XML-DSig standard. In this chapter we will see how these standards work.

8.1 PKCS#7 and CMS

PKCS#7 is the original standard from RSA, since RSA was the owner of the RSA algorithm, they wanted also to define standards about the usage of the algorithm, and they had a series of standards named PKCS.

PKCS#7 was a standard for secure enveloping (secure is a generic term, it means authentication, integrity, confidentiality and so on), at some point in v1.5, the work was shared with IETF and later RSA stopped the development of PKCS#7 and the rest was developed directly by IETF which renamed it in CMS (Cryptographic Message Syntax).

> CMS is a **secure envelop** that allows data authentication, integrity and, optionally, privacy, with symmetric or asymmetric algorithms, depending on the kind of application.

It also supports multiple signatures (hierarchical or parallel) on the same object, and it can include the certificates (and can include also revocation info) to verify the signature to be a self contained object, where the signature envelops the data and the certificate needed to verify the signature.

PKCS#7 and CMS are raw format, in the sense that they consider generic data that are just binary blob/stream (sequence of bit), and this sequence of bits is encapsulated in a secure container. It is very important because it is the only standard that permits to sign and encrypt any kind of data, and it is a recursive format, which is important because there is not one format which is providing signature and encryption, but it is possible to achieve that by making first encryption, then that object becomes a generic data which is inserted in another container which is providing the signature.

PKCS#7 and CMS are defined using ASN.1 (Abstract Syntax Notation 1) with different encoding rules (Basic Encoding Rule, BER, for most of the standard and the Distinguished Encoding Rule, DER, only for the signed attributes and authenticated attributes).

Initially, it was defined by RSA and then it evolved over the time:

- RFC-2630 (Jun '99): fully compatible PKCS#7 1.5 but with key-agreement (DH algorithm) and pre- shared keys
- RFC-3369 (Aug '02): adds password-based keys and an extension schema for generic key management and it splits the document into two RFCs (one for the basic structure of secure envelop and the other for the algorithms, so that they can evolve independently for the secure container)
- RFC-3852 (Jul '04): it is just a generalization, an extension that supports generic certificates (not very frequent to use something different from X.509 is used)
- RFC-5652 (Sep '09): clarifications about multiple signatures

8.1.1 Algorithms for CMS

The allowed algorithms are documented in RFC-3370:

- Digest MD5, SHA-1: quite old
- Signature RSA, DSA (insecure without elliptic curve)
- Key management:
  - DH for key agreement
  - RSA for transport
  - For symmetric wrapping 3DES and RC2
  - Derivation PBKDF2
- For encryption: 3DES-CBC, RC2-CBC
- For MAC: HMAC-SHA1

These algorithms were very basic and quite old, but with new RFCs the situation improved:

- For encryption: CAST-128, IDEA, AES, Camellia, SEED
- For digital signature RSASSA-PSS, so a better schema is used (more resistant to attacks)
- For encryption and digest: GOST (the one used in Russia)
- AES-CCM and AES-GCM for authenticated encryption (be aware: it does not provide nonrepudiation since it is not digital signature)
- Boneh-Franklin and Boneh-Boyen for Identity-Based Encryption, it is a new kind of encryption in which the key is based on the identity of the actors (e.g., IP address)
- ECC and SHA-2 family is supported
- RSA-KEM and RSAES-OAEP for key transport

8.1.2 CMS structure

The CMS structure is shown in figure 8.1, and basically one CMS component, the **contentInfo**, which contains inside the contentType and the content itself.

This allows for encapsulation, because the content can be another CMS object(contentInfo) allowing for a recursive structure.

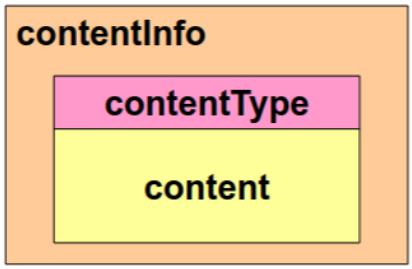


Figure 8.1: CMS structure

8.1.3 CMS content type

There are different contentType available:

- **data**: A generic sequence of bytes, serving as the base content.
- **signedData**: The data accompanied by one or more digital signatures (1..N), allowing parallel signing. In this case, all the signatures are computed over the same data, without any particular order.
- **envelopedData**: The data encrypted using a symmetric algorithm, with the symmetric key encrypted for the intended recipient(s). This means that the symmetric key is present in the data, but only the intended recipient(s) can decrypt it, because it is encrypted with their public key.
- **authenticatedData**: The data, a Message Authentication Code (MAC), and an encrypted key for recipient(s). Same drill as with envelopedData, the key is present and encrypted with the public key of the intended recipient(s).
- **digestedData**: The data along with its cryptographic digest for integrity verification.
- **encryptedData**: The data encrypted using a symmetric algorithm, providing confidentiality.

8.1.4 CMS signed data

It is used to implement digital signature, so the content type is signedData, while the actual content has:

- **version**: the version of the object, not the version of the CMS standard.
- **digestAlgorithms**: this is a list: since the signed data can be signed by different people, they may be using different algorithms to sign the data. The algorithms must

be listed before the signature to make them known to the verifier before the data is actually read. This allows to compute the digest in parallel while the data is being read.

- **encapContentInfo**: the informations about the content that is being encapsulated.
- **certificates**: this is optional, these are all the certificates of the signers, and typically all the certificate chain for each signer.
- **crls**: this is optional, it is used to verify that each certificate was valid at the moment of the signature.
- **signerInfo**: a block for each signer and each block is composed by a version number, the distinguish name of the issuer of the certificate (so the CA) + the Serial Number of the certificate related to this specific signature (because a generic signer could have more than one certificate), finally you have the digest encrypted with the private key corresponding with this public key.

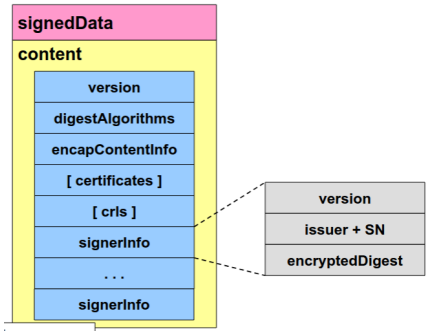


Figure 8.2: CMS signed data

8.1.5 CMS enveloped data

The enveloped data, on the contrary, provides encryption.

- **version**: to know which kind of object we are considering.
- **encryptedContentInfo**: which is a block composed of a contentType (contains the type of content you will find after the decryption), the description of the algorithm used for the encryption, and finally the encrypted content. Notice that only one algorithms can be specified, meaning that one can sign with different algorithms but can encrypt with only one.
- **recipientInfo**: this is the list of possible recipients (or people that are authorized to perform the decryption), because for each possible recipient there is one block that contains the version, the issuer + the serial number of the certificate (as in the previous case), the encryption algorithm used to encrypt the symmetric key (e.g. RSA), and finally the encrypted key.

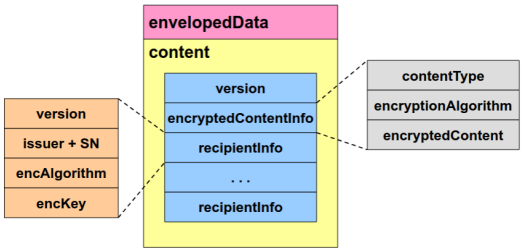


Figure 8.3: CMS enveloped data

8.2 XML digital signature

XML signature is important because there are several practical systems that make use of XML to represent generic data.

XML signature is a standard promoted not by IETF but the W3C with the cooperation by IETF but since most of the XML standardization is done in W3C they were the initiator. Even if the standard is named XML signature note that it covers not only digital signature but also a MAC. One standard here covers both things. With respect to the signed data, it provides authentication, integrity, and non-repudiation (optional, only if the signature is performed with asymmetric crypto and associated to an X.509 certificate).

Another desirable feature that the XML signature provides is the independence from the transport layer or any storage technique.

8.2.1 XML Signature – Standard

1st Edition (Recommendation):

- Initial version: `www.w3.org/TR/2002/REC-xmldsig-core-20020212/`
- RFC-3275 specifies its use.
- Latest version of v1 available at: `http://www.w3.org/TR/xmldsig-core/` (currently v1.1, dated 11-Apr-2013).

2nd Edition:

- Available at: `https://www.w3.org/TR/xmldsig-core2/`.
- Latest version published on 23-Jul-2015.

8.2.2 XML Digital Signature – Namespace

To identify the elements that are present in the standard, there is the need of specific identifiers. In XML terminology it is namespace. Specific identifiers are can be found at `http://www.w3.org/2000/09/xmldsig#`. An example of the namespace is shown in figure 8.4. It is named `dsig`, which is providing entries to the defined url.

It is common, sing XML is very verbose, to define an abbreviation, by using the ENTITY command. It means that after that is possible to use the abbreviation with the `&` in the front, so that is not necessary to repeat that long string.

```
<?xml version='1.0'>
<!DOCTYPE Signature SYSTEM "xmldsig-core-schema.dtd"
  [ ENTITY dsig "http://www.w3.org/2000/09/xmldsig#" ] >
<Signature xmlns="&dsig;" id="mySignature">
...
```

Figure 8.4: XML Signature Namespace

8.2.3 General Characteristics of XMLdsig

The XML Signature framework offers versatile signing capabilities for XML and non-XML objects.

The signed object may be:

- Internal (contained within the signature) or external (only referenced via a URI).
- XML (native object) or non-XML (encapsulated object with an XML signature).

It supports different signature levels for the same content, with flexible semantics: The main difference with reference to a CMS signature is the flexibility of the semantics, in fact the XML signature can be:

- Signed.
- Co-signed.
- Witnessed.
- Notarized.

However, XMLdsig does not address key creation or certification, focusing solely on signing and integrity verification.

8.2.4 Signature types

The different formas have been already briefly discussed in section 7.1, and are basically the same in XML. With an enveloping signature, the XML object in stored inside a XML signature object, while with an enveloped signature the XML object reserve some space to store the signature. A detached signature in XML is basically and URI which points to the signature.

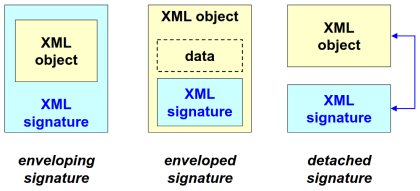


Figure 8.5: XML Signature Types

8.2.5 Canonicalization

When dealing with XML, there's a problem that have to be dealt with: there are many equivalent forms for the same data. Take a look at this example:

```
<person id="12345" name="Antonio"/>
<person name="Antonio" id="12345"/>
<person name="Antonio" id="12345"></person>
```

Listing 3: Equivalent XML

All the three forms are equivalent, but if you compute the hash of each of them, you will get different results, due to the different format. Another example could be empty spaces, which are not relevant for the XML parser, but they are relevant for the hash computation, and so on.

In those case logical equivalence should be the same as actual one in terms of signature.

Canonicalization algorithms

Canonicalization algorithms are used to transform the XML document in a canonical form, so that the signature is always the same, no matter the format of the XML document. This is done by creating a physical representation of the XML document, by representing the XPath node set as an octet sequence.

The algorithms changed over the time, the default being canonical XML 1.0 which follows that specification, eventually with comments (if comments are appropriate or not inside the XML document) but the preferred version is the most recent one called canonical XML 1.1: the comments should be deleted when computing the signature or should be included in the computation of the signature? There are two different canonicalization ways according to the fact that comments are included or not.

Canonicalization process

The canonicalization process is specified at `http://www.w3.org/TR/xml-c14n.html` and is as follows:

1. encode the document in UTF-8
2. normalize line breaks to #xA(10), before parsing, after all some use line feed, some use carriage return, some use both
3. normalize attribute values, as if by a validating processor, for example remove the leading zeroes in the numbers
4. replace character and parsed entity references(replace references with the actual content)
5. replace CDATA sections with their character content
6. remove XML declaration and DTD
7. convert empty elements to start-end tag pairs

8. normalize whitespace outside of the document element and within start and end tags
9. retain all whitespace in character content (but characters removed during line feed normalization), meaning that spaces in strings are retained
10. set attribute value delimiters to quotation marks(single quotes became double quotes)
11. replace special characters in attribute values and character content by character references
12. remove superfluous namespace declarations from each element
13. add default attributes to each element
14. impose lexicographic order on the namespace declarations and attributes of each element

8.2.6 Structure of XML signature

The general structure of an XML signature is shown in figure 8.6, and as you can see can have an optional identifier. Keep in mind that SignatureValue is computed over the SignedInfo.

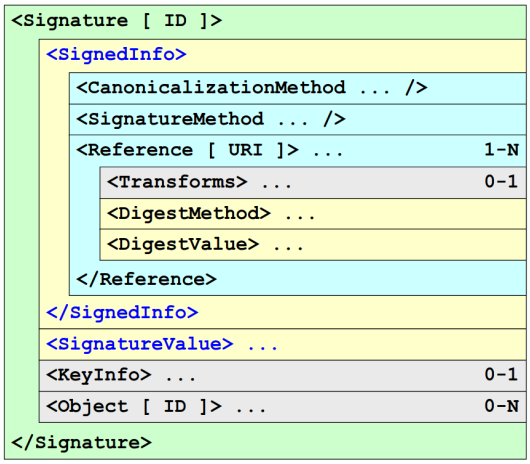


Figure 8.6: XML Signature Structure

SignedInfo

The `SignedInfo` element is a crucial component of an XML signature and always appears as the first element. Its main components include:

- **CanonicalizationMethod:** A data normalization technique applied before generating the signature.
- **SignatureMethod:** Specifies the algorithm used for signature generation, such as RSA+SHA1.
- **Reference [URI]:** A pointer to the actual data that has been signed. The 1:N cardinality means that one can have just one signature for n objects or n references when you want to sign a group of documents.

- **Transforms (optional):** Data transformations applied before signing, such as selecting elements via XPath.
- **DigestMethod:** The hash algorithm employed, e.g., SHA1.
- **DigestValue:** The computed hash value of the referenced data.

Reference URI

The `Reference URI` is a pointer to an element that has been signed. Its usage depends on the context and signature type:

- **Omission:** The `URI` may be omitted for at most one `Reference` or `Manifest` component if the signed element is unambiguously determined by the context (e.g., an application-level message signature).
- **Null URI (URI=""):** Used to generate an enveloped signature.
- **Self URI (URI="#id"):** Used to generate an enveloping signature.
- **External URI (e.g., URI="http://..."):** Used to generate a detached signature.
  - When an external URI is specified, follow any redirects (status 3xx) until reaching the body of a positive response (status 200).

Reference Type

A reference can point to various element types, each serving a specific purpose:

- **Object:** Refers to a generic object.
- **SignatureProperty:** Points to a signature attribute, such as date and time or the HSM identifier. The type is specified as `Type="&dsig;SignatureProperties"`.
- **Manifest:** Represents a set of references, useful for grouping related elements. The type is specified as `Type="&dsig;Manifest"`.

Manifest

It is a list of references external to SignedInfo but pointed to by the same. Basically the manifest is somewhere in the XML tree and is pointed to by the SignedInfo, as you can also see from figure 8.7.

There is a conceptual difference to understand why there is a manifest rather than having the pointer to the object directly inside signedInfo. Each reference in SignedInfo is subject to the validation procedure; it means that if anyone is invalid (e.g., inaccessible URI, wrong DigestValue) then the whole signature is invalid. If the object is put inside SignedInfo and validation fails, the whole signature must be rejected. On the other hand, any reference in the manifest is not individually validated since the procedure considers only the Manifest as a whole. This means that it is possible to check that the signature is the correct digest of the manifest but not where the manifest is pointing because the manifest is here, it is signed but then the manifest is pointing, for example, to three external documents, beacuse it out of

scope checking the integrity of the linked documents, as it's the application's responsibility to do so.

It is possible to notice that in the SignedInfo there is a reference to a URI which is towards an object, and the type is "signature of a manifest". Then, at some point there is an object which has the same identifier (**myManifest**) as the reference, saying "this is the thing that was signed", and inside the manifest there are several references. So the digestValue in the Reference is the digest of the list contained in the manifest. So, we are only checking that the SHA1 is correct, but then each of these references contains a method and a value. So, if you want you can verify also if those objects have changed or not. But this is not what is done by XML signature.

This approach provides flexibility but also introduces risks. If you rely on XML signatures without careful implementation, you might assume that having a digital signature guarantees protection. However, if your implementation references objects indirectly through a manifest instead of directly, someone could potentially alter values within the references without detection. The automatic XML signature process only verifies that the SHA1 hash in the `DigestValue` field matches the overall manifest. It does not follow the links to validate individual elements.

```
<Signature>
  <SignedInfo>
    <Reference URI="#MyManifest" Type="&dsig;Manifest">
      <Transforms> ... </Transforms>
      <DigestMethod Algorithm="&dsig;sha1"/>
      <DigestValue> dGhpcyBpcyBub3Q... </DigestValue>
    </Reference>
  </SignedInfo>
  <Object>
    <Manifest id="myManifest">
      <Reference ...> ... </Reference>
      <Reference ...> ... </Reference>
    </Manifest>
  </Object>
</Signature>
```

Figure 8.7: XML Signature Manifest

Transforms

The transforms may contain one or more `<Transform>` tags to be applied sequentially before computing the digest. That's important otherwise you risk computing the wrong digest. Possible transformations are:

- EnvelopedSignature: the one which is compulsory and may be implemented via Xpath
- XPath transform: compulsory
- XSTL (X Stylesheet) transform optional
- Other transformations may be used but they are quite deprecated because they are not native in XML

XPath plays a crucial role by allowing you to specify which parts of an XML object should be included in the signature. This flexibility is important depending on what you are signing—whether every part is relevant or if certain elements are variable and not meaningful.

For example, imagine signing a file that includes an access control section recording when the file was last accessed. This timestamp is likely to change over time and isn't relevant to verifying the file's integrity. What matters is ensuring that the file's content remains unchanged. Including such auxiliary information in the signature computation would require re-signing the file every time it's accessed.

Using XPath, you can define precisely which parts of the object should be included in the digest computation, enabling efficient and meaningful signatures.

DigestMethod and DigestValue

The `DigestMethod` specifies the cryptographic hash algorithm used to generate the input for the signature process. One compulsory algorithm is `SHA1`, which is identified by the URI `http://www.w3.org/2000/09/xmldsig#sha1`. This algorithm operates on a byte stream as input.

The handling of input depends on the output of the URI dereference and any `Transforms` applied:

- If the result is a set of XPath nodes, the byte stream is generated using Canonical XML (`XML-C14N`).
- If the result is already a byte stream, the digest is computed directly.

The `DigestValue` represents the value obtained after applying the `DigestMethod`. This value is encoded using Base64 for inclusion in the XML signature structure.

SignatureMethod

The `SignatureMethod` specifies the algorithm used to generate the digital signature. A complete list of supported algorithms is available at `https://www.w3.org/TR/xmldsig-core2/#sec-SignatureMethod`.

Mandatory algorithms include:

- For digest operations: `SHA-256` (recommended) and `SHA1` (discouraged).
- For MAC operations: `HMAC-SHA-256` (recommended) and `HMAC-SHA1` (discouraged), with an optional parameter to specify the truncation length.
- For digital signatures: `RSAwithSHA256`, `ECDSAwithSHA256`, and `DSAwithSHA1` (supported for verification only).

The signature value is the value base64 encoded. This is not computed directly over the referenced data, but on their digest (because we compute the signature of SignedInfo).

```
<SignatureMethod
  Algorithm="http://www.w3.org/2001/04/xmldsig-more#hmac-sha256">
  <HMACOutputLength>128</HMACOutputLength>
</SignatureMethod>
```

Figure 8.8: XML Signature Method

8.2.7 Security requirements

XML security mechanisms address several key requirements:

Confidentiality is provided by TLS and XML Encryption but not directly by XML Signature.

Message Authentication ensures message integrity and verifies the creator using MAC or digital signatures. However, it does not confirm the sender's identity.

Sender/Receiver Authentication verifies the identities of both parties, achieved through TLS client authentication. This does not ensure the authenticity of the message creator.

To guarantee that the signer is also the sender, we may use the same X.509 certificate as in SOAP-DSIG or SOAP-TLS (in this case, the `<ds:KeyInfo>` element may be omitted)

8.2.8 Weaknesses

XML Signature implementations have some potential vulnerabilities:

A **malicious receiver** could falsely claim to have received a message multiple times, even if it was sent only once. To counter this, a nonce, such as a timestamp, should be added to the message.

A **malicious receiver** might forward the signed message to a third party, who could then falsely claim to have received it directly from the sender. Including the intended receiver's identity within the signed message can mitigate this risk.

Possible countermeasures involve adding both a timestamp and receiver identity to the signed body of the message.

8.2.9 Example of a (detached) XML signature

An example of a detached XML signature is shown in listing 4. The signature algorithm is DSA with SHA-1, and the canonicalization algorithm is XML-C14N, as specified by the transform algorithm. The signature is deatached because the URI points to an external document.

```
<Signature Id=\Lioy_dsig_081026173907" xmlns=\http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm= "http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
    <SignatureMethod Algorithm= "http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
    <Reference URI="http://www.sito.it/doc.xml"/>
      <Transforms>
        <Transform Algorithm= "http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
      </Transforms>
      <DigestMethod Algorithm= "http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>j61wx3rvEPO...</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>JK34s...</SignatureValue>
  <KeyInfo>
    <X509Data>
      <X509Certificate>
        MII...AABvi
      </X509Certificate>
    </X509Data>
  </KeyInfo>
</Signature>
```

Listing 4: XML Signature Example

8.3 XML Encryption

XML encryption is a standard to represent encrypted data while also specifying the encryption and decryption process. It is a W3C standard with different versions, the first one being the 1.0, and the latest, and recommended, one being the 1.1.

There are 4 different possible encryption scopes:

- a XML element
- the content of an XML element
- a whole XML document
- just a portion: an EncryptedData or EncryptedKey element (this is named super-encryption)

The result of any of these operations is always an XML encryption element, which may contain the encrypted data. So, in this case, there is a detached encryption. It means that the encrypted content is in some place, and in XML encryption there is everything needed to decrypt that content.

8.3.1 Syntax

The syntax of XML Encryption (`XMLenc`) includes the following components:

- **EncryptedType**: This is the abstract type used for both `EncryptedData` and `EncryptedKey`.
- **EncryptionMethod**: Specifies the algorithm employed for encryption.
- `ds:KeyInfo`: Contains details about the key used in the encryption process.
- **CipherData**: This includes either:
  - `CipherValue`: Base64-encoded encrypted data.
  - `CipherReference`: A pointer to the encrypted data.
- **EncryptionProperties**: Provides supplementary information about the generation of the encrypted data.
- `Id` (optional): A unique document identifier.
- `Type` (optional): Specifies the type of document, such as an element or content.
- `MimeType` (optional): Indicates the MIME type of the encrypted data.

8.3.2 XML Encryption: Required Algorithms

In version 1.0 of XML Encryption, the following algorithms were used:

- **Encryption**: 3DES-CBC (for backward compatibility), but also AES-128/256-CBC.
- **Key Transport**: RSA-v1.5, RSA-OAEP (RSA Optimal Encapsulation, preferred). Used to give the encrypted content to the recipient.
- **Symmetric Key Wrap**: 3DES (for backward compatibility, should never be used), AES-128, or AES-256 KeyWrap.
- **Digest**: SHA1, SHA256 (recommended). The digest is not a signature but is useful.
- **Authentication**: XMLdsig.

In version 1.1, additional algorithms were introduced, including support for authenticated encryption:

- **Encryption**: 3DES-CBC, AES-128/256-CBC, AES128-GCM.
- **Key Derivation**: ConcatKDF. In the previous versions, PKDF2 or HKDF were used, but XMLenc uses ConcatKDF.
- **Key Transport/Key Agreement**: RSA-OAEP, ECDH.
- **Symmetric Key Wrap**: 3DES, AES-128, or AES-256 KeyWrap.
- **Digest**: SHA1 (discouraged), SHA256.

8.3.3 Examples of XML Encryption

Let's now look at some examples of XML encryption. We will take as reference listing 5, which is an XML document containing payment information of a user.

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://cards.org/payment'>
  <Name>Gianni Pautasso</Name>
  <CreditCard Limit='1,500' Currency='EUR'>
    <Number>4091 2450 0288 5657</Number>
    <Issuer>MyBank/Issuer>
    <Expiration>06/10</Expiration>
  </CreditCard>
</PaymentInfo>
```

Listing 5: XML Payment Info

One possibility, if we want to protect sensitive data, is to encrypt the whole `CreditCard` XML tag, as well as its content, as shown in figure 8.9a. In this case, the encryption algorithm is not specified, its not compulsory after all, which means that the knowledge of which algorithm to use is delegated to the application that will decrypt the data, as well as the key agreement.

Another possibility is to leave most of the `CreditCard` data in clear, and encrypt only the credit card number, as shown in figure 8.9b.

A third possibility is to encrypt the whole XML document, as shown in figure 8.9c. In this case, it is necessary to specify the MIME type of the encrypted data explicitly, unlike the previous example where the encrypted part was basically just data.

Until now, the encryption algorithms were not specified, but it is possible to do so, as shown in figure 8.9d, which is basically the previous example with the addition of the encryption algorithm. A key must also be specified. In this case, the standard from XMLd-sig is used to give a name to this key, "Alice-Bob" in this case, which is just something meaningful to the two parties involved.

(a) XML Encryption Example 1

(b) XML Encryption Example 2

(c) XML Encryption Example 3

(d) XML Encryption Example 4

Figure 8.9: XML Encryption Examples

Chapter 9

JOSE

XML objects are cool and all, but are very verbose and, in some instances, require many resources to be parsed, which is an undesirable feature when the main focus in on IoT and embedded devices. This is where JSON comes in. JSON is a lightweight data interchange format, which is also becoming the de-facto standard for data exchange on the web. After all, XML isn't even natively supported in mobile devices, while JSON is.

This leads to the need for industry-standard JSON-based formats for: Security Token, Signature, Encryption and Public Key. The solutions are: JSON Web Token (JWT), JSON Web Signature (JWS), JSON Web Encryption (JWE) and JSON Web Key (JWK).

The design philosophy is to try to keep things as simple as possible and to make complex things simpler without losing too many features. The design goal was to make these things easy to use in all systems that are already web native. For this reason, it will use compact, URL-safe representation. URL-safe means that if an identifier of any kind is used inside the URL, then the URL is still usable. It is known that there are various ways to represent special character (for example the space is substituted with %20) so the requirement is that any identifier used in JSON must be URL-safe because it will be probably exchanged through a URL. The success of this is due to the fact that these standards have been promoted and rapidly adopted by the major internet companies (Google, Facebook, AOL, NRI, Microsoft, ...)

Standardization

The following RFCs define the standards and best practices for JSON Web technologies:

- **RFC-7515**: "JSON Web Signature (JWS)".
- **RFC-7516**: "JSON Web Encryption (JWE)".
- **RFC-7517**: "JSON Web Key (JWK)".
- **RFC-7518**: "JSON Web Algorithms (JWA)".
- **RFC-7519**: "JSON Web Token (JWT)".
- **RFC-7797**: "JSON Web Signature (JWS) Unencoded Payload Option".

- **RFC-8725**: "JSON Web Token Best Current Practices".
- **RFC-7520**: "Examples of Protecting Content Using JSON Object Signing and Encryption (JOSE)".

9.1 JSON Web Signature (JWS)

JWS was designed to sign arbitrary content using a compact, JSON-based representation. However, as with XML Signature, the term "signature" in JWS is sometimes used loosely and may lead to confusion. JWS allows for both true digital signatures and MACs. While MACs ensure authentication and integrity, they differ from true signatures because they rely on symmetric keys, making the roles of signer and verifier interchangeable. In contrast, a proper digital signature clearly distinguishes between the signer and the verifier.

JWS representation consists of three parts:

- **Header**: Metadata about the signature and algorithm.
- **Payload**: The content being signed.
- **Signature**: Cryptographic signature or HMAC.

In JWS compact serialization, parts are base64url encoded and concatenated, separated by period characters:

```
base64url(header) || '.' || base64url(payload) || '.' || base64url(signature)
```

9.1.1 B64 and B64url

The specification for B64 are defined in in RFC-4648, as well as B16 and B32. Base64 encodes 6 bits at a time using an 8 bits ASCII character, leading to a 33% size increase (since 6 bits become 8). Most of the characters are allowed (A-Z, a-z, 0-9, +, /, =) except the padding character '=', used to make the size a multiple of 6 if needed. Basic B64 is not URL safe because some characters(/+=) are used in URLs. To make it URL safe, the characters are replaced with URL safe characters (+ becomes -, / becomes _) and the padding character is removed(not added at all).

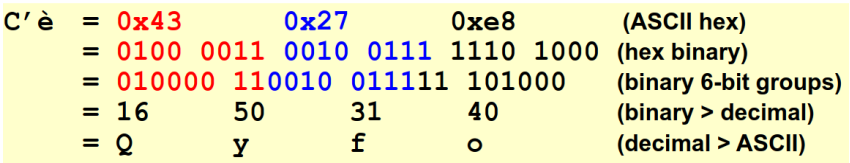


Figure 9.1: An example of B64 encoding.

9.1.2 JWS Header Parameters

The JWS header contains various parameters to provide metadata about the signature and associated keys. These parameters include:

- **"alg"**: Specifies the signature algorithm used (required).

- **"jku"**: Refers to the JSON Web Key URL.
- **"x5t"**: A base64url-encoded SHA-1 thumbprint of the DER encoding of the X.509 signature certificate.
- **"x5t#S256"**: Similar to `"x5t"` but uses a SHA-256 thumbprint.
- **"x5c"**: Contains the base64-encoded DER encoding of the signature certificate. If a certificate chain is provided, it appears as a JSON array.
- **"x5u"**: Specifies the X.509 URL, which can be used to retrieve the PEM-encoded signature certificate or its chain (starting with the end-entity certificate and proceeding up to the root).
- **"kid"**: Represents the key identifier.
- **"typ"**: Indicates the type of the signed content.

Note: URLs referenced in the header must use an integrity-protected channel with server authentication, such as TLS, to ensure security.

9.1.3 JSON Web Algorithm (JWA)

JSON Web Algorithm (JWA) defines compact identifiers for algorithms used in JWS. These include:

- **"none"**: No signature (required).
- **"HS256"**: HMAC using SHA-256 (required).
- **"RS256"**: RSASSA-PKCS1-v1_5 with SHA-256 (recommended).
- **"ES256"**: ECDSA with curve P-256 and SHA-256 (recommended+).

Other algorithms may be used but the ones above are the required or the recommended ones. The recommended one are not compulsory due to an interoperability problem: a JSON client that interacts with a JSON server it may not have stronger implementation than the required ones. In any case, for a complete list of JOSE parameters and supported algorithms, refer to `https://www.iana.org/assignments/jose/jose.xml`.

9.1.4 JWS header and payload example

Two examples of JWS header and payload are shown in Figures 9.2a and 9.2b. It's important to keep in mind that a Carriage Return Line Feed (CRLF) character must be present at the end of each line, except for the last one after the closing bracket.

The specified header is a JWT signed with HMAC SHA-256, which will be base64url encoded.

```
# plain header
# (beware! CR-LF at each EOL but the last one)

{
"typ":"JWT",
"alg":"HS256"
}
```

```
# base64url encoded header
ew0KInR5cCI6IkpXVCIsDQoiYWxnIjoiSFMyNTYiDQp9
```

(a) JWS header.

```
# plain payload (JWT)
# (beware! CR-LF at each EOL but the last one)
{
"iss":"polito.it",
"exp":"1609459199",
"http://polito.it/is_professor":true
}
```

```
# base64url encoded payload (w/ line break for readability)
ew0KImlzcyI6InBvbGl0by5pdCIsDQoiZXhwIjoiMTYwOTQ1OTE5OSIsDQo
iaHR0cDovL3BvbGl0by5pdC9pc19wcm9mZXNzb3IiOnRydWUNCn0
```

(b) JWS payload.

Figure 9.2: JWS header and payload example.

9.1.5 JWS signature

The signature in JWS covers both the header and the payload. This is crucial because if the signature were to cover only the payload, the header could be modified by an attacker. For instance, the attacker might alter the signature algorithm or content type, potentially causing issues with signature verification or content interpretation.

To create the signing input, the encoded header and payload are concatenated, separated by a period (.). The resulting signature is then generated based on this input, base64url-encoded, and appended to the signing input after another period. This ensures the integrity of both the header and payload.

9.2 JSON Web Key (JWK)

To perform a signature, keys are required. While the previous examples assumed that both parties already knew which key was being used, JWK (JSON Web Key) is used to explicitly represent the keys. JWK provides a standardized way to describe both asymmetric and symmetric keys.

The `kty` field specifies the key type and can have values such as EC (Elliptic Curve, strongly recommended), RSA (required), oct (octet sequence, also required), or OKP (Octet Key Pair). Additionally, the kid field can be used as a key identifier, allowing a specific key to be referenced by name.

JWK parameters vary depending on the key type and algorithm being used. For example:

- **For RSA keys:** `n` represents the modulus, and `e` represents the public exponent.
- **For AES keys:** `k` holds the key value.

```
# 256-bit symmetric key
{
"kty":"oct",
"k":"AyM1SysPpbyDfgZld3umj1qzKObwVMkoqQ-EstJQLr_T-1qS0gZH75
aKtMN3Yj0iPS4hcgUuTwjAzZr1Z9CAow"
}

# 2048-bit RSA public key
{
"kty":"RSA",
"n":"ofgWCuLjySXndS5z5rexMdbBYUsLA9e-KXBdQO...",
"e":"AQAB",
"kid":"Antonio.Lioy.2020"
}
```

Figure 9.3: Example of a JWK.

9.3 JSON Web Encryption (JWE)

JOSE also supports encryption through JWE, which allows arbitrary content to be encrypted using a JSON representation. A JWE consists of three parts: the header, the encrypted key, and the ciphertext. It can be represented in two formats: the native JWE compact serialization and the JWE JSON serialization. Importantly, the key is also encrypted, so the header must specify two algorithms: one for encrypting the key and another for encrypting the ciphertext.

9.3.1 JWE Header Parameters

The JWE (JSON Web Encryption) header contains several important parameters:

- `alg` = The key encryption algorithm, used to protect the Content Encryption Key (CEK).
- `enc` = The content encryption algorithm, used to encrypt the actual content.
- Key identification parameters (similar to those in JWS):
  - `jku` = The URL of the JSON Web Key.
  - `x5t` = The SHA-1 thumbprint of the X.509 certificate.
  - `x5t#S256` = The SHA-256 thumbprint of the X.509 certificate.
  - `x5c` = The X.509 certificate.
  - `x5u` = The URL of the X.509 certificate.
  - `kid` = The key identifier.

9.3.2 JWE Compact Serialization

As previously mentioned, JWE can be represented in two formats. The compact serialization format is similar to JWS, with the header, encrypted key, and ciphertext concatenated and separated by periods, but it also has more parts:

```
base64url(header) || '.' || base64url(encrypted CEK) || '.' ||
base64url(iv) || '.' || base64url(ciphertext) || '.' || base64url(auth
tag)
```

There's only one recipient (one of the limits) which means that it is not something that can be decrypted by many recipients (in PKCS#7 there was the possibility for both signatures to have various signers and encryption to have keys encrypted for several recipients), because JSON is thought to be exchanged between one client and one server. There are also no unprotected headers and no associated data (so the tag is computed on everything).

9.3.3 JWE JSON Serialization

This is the second format for JWE. JWE JSON serialization represents encrypted content as a JSON object with the following top-level members (all optional except for the ciphertext):

- `"protected"`: Integrity-protected headers.
- `"unprotected"`: Other (unprotected) headers.
- `"iv"`: Initialization vector.
- `"aad"`: Additional authenticated data.
- `"ciphertext"`: The actual encrypted content.
- `"tag"`: Authentication tag.
- `"recipients"`: An array of JSON objects identifying each recipient and providing the CEK encrypted with their public key.
- `header` {`alg, kid`}: Algorithm and key identifier for encryption.
- `encrypted_key`: The encrypted content encryption key.

An example of JWE JSON serialization is shown in Figure 9.4. The example demonstrates the structure of the protected and unprotected headers, with the unprotected header containing the key identifier. The "recipients" field includes the header specifying the algorithm and the key identifier, followed by the encrypted key, which is encrypted using the public key indicated by the identifier. In the case of the second recipient, the algorithm "a128kw" refers to AES-128-KW (Key Wrap), where the symmetric key is encrypted using another symmetric key. A previous agreement has been made about the key identified by number 7, and then the key is encrypted.

Next, the structure includes the initialization vector, the actual ciphertext, and the authentication tag. Notably, the algorithm used is not explicitly specified, as it is optional. In this example, it is assumed that the encryption algorithm declared externally was "A128CBC-HS256" (AES 128 CBC with HMAC SHA256 for the authentication tag). While the name implies encryption, the format supports both encryption and message authentication. It's possible to use the same format by specifying null for encryption and using only the authentication portion, though JWS would be preferred for that scenario. The standard format avoids using encryption, followed by re-encapsulation in a signature, in cases where both encryption and authentication are required.

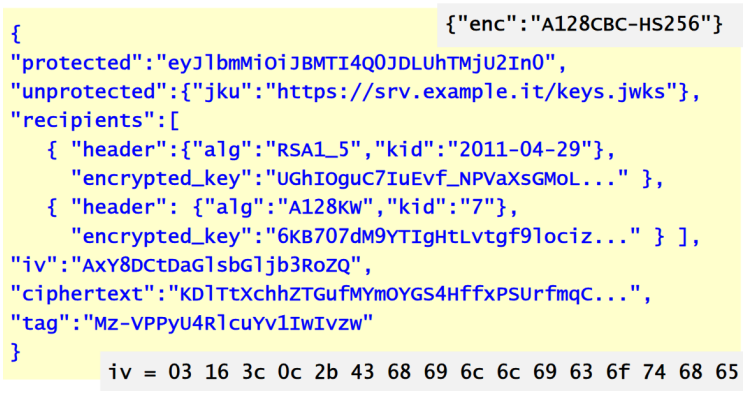


Figure 9.4: Example of a JWE JSON serialization.