POLITECNICO DI TORINO

Cybersecurity

Full notes from the course 01UDROV of Prof. Antonio Lioy

A.A. 2021/22

Authors: Marco Smorti, Oscar Piccirillo, Dario Lanfranco,
Andrea Scoppetta, Alberto Colletto, Francesco Rametta

A special thank to all the mentioned beautiful guys which helped so much during the course to make this hard job easier and to make these notes available.

DISCLAIMER: These are unofficial notes, it means that professors of Politecnico are not involved in this work in any way and notes may contains errors. Use it at your own risk!! To report errors in these notes, write to @GrayNeel on Telegram.

TLS (Transport Layer Security)

TLS is currently the most widely adopted network security protocol. In the origin it was named SSL (Secure Socket Layer): this term should not be used anymore because the term refers to a protocol that has been discontinued (SSL 2, 3 no more used) because it was weak and had vulnerabilities. The version that has been later standardized is TLS. SSL was originally proposed by Netscape Communications in 1995 and the main security properties and features remain the same, only the implementation changed. TLS creates a **secure transport channel** on top of the layer 4 (also called *session level* security, but it does not implement full session management) and it is also named a layer 4.5 protocol. It provides features equivalent to a normal streaming protocol such as TCP but with security features. Security features are:

- **Peer authentication** when channel is being opened (beware it is peer authentication, several meanings of authentication). It is compulsory for the server and optional for the client. If there is single peer authentication it is only the server, otherwise there is a mutual peer authentication for both server and client. It is based on *asymmetric challenge-response* authentication: a proof of possession of private key is needed. For the server that is **implicit** because the server is using its private key in creating all the other keys (indirect proof of the server having its private key otherwise it can't create the secure channel). On the contrary, for the client there is an **explicit** signature performed. This decision is a matter of the *system/security manager* (who implements TLS). User has no control over this. If peer authentication fails, channel is not opened. This is an important thing because if an attacker is stopped even from creating a channel to us, all the other problems are no more problems. That's why it is so important that authentication is performed at network level rather than application level (but most developers do not understand this and keep doing authentication at application level, which is too late). The sooner the authentication is performed in the network stack, the better is the rejection of the attackers.
- **Message confidentiality** means that the content of each record transferred over TLS is **optionally** encrypted. It depends on the version of the protocol, which in some cases is optionally and in other cases is mandatory. Remember that encrypting all the data, especially in massive downloads, it takes lot of time and in some cases data is not confidential, so message confidentiality is not needed. Again, it is a matter of the system manager.
- **Message authentication and integrity** for each message sent over the TLS channel. Here there is another meaning of authentication (data authentication), which is a proof that data exchanged on the channel is really coming from the counterpart. It also get a proof that data have not been changed: beware that integrity gives only the information if data have been changed or not (but do not prevent changes).
- **Protection against replay and filtering attacks**, in which a valid message is taken and injected again. TLS provide replay protection, because if an old message is replayed another time it is detected. In the same way it provides protection for message cancellation, which could change the meaning of what is exchanged (e.g. think to two messages: *move to /tmp, remove all files*. If the first one is cancelled, then all the files of the whole machine are deleted). This kind of protection is obtained thanks to an **implicit record numbering,** which means that each message that is sent across TLS is a numbered record, but there's no place for that number inside the packet. This is because TLS is layered on top of TCP, which has the property of no data loss and data are received in the same order as they are sent. So the first record received is number 1, the second is number 2 and so on.. Thanks to this it is possible to detect replay attacks. Since record number is used in message authentication, if there is a message received two times, then authentication fails. The same is for the cancellation of a message. Numbering works implicitly together with TCP and authentication.

From an architectural point of view there is the basic *network protocol*, then a *realiable transport protocol* is needed (e.g. TCP but anything equivalent works with TLS). On top of that there is the **TLS record protocol**, which is the protocol that encapsulates the data being transmitted. The basic data transmitted is the **record** (do not talk about TLS packet or TLS stream, because the unit of transmission is the TLS record). The TLS record is used for many purposes:

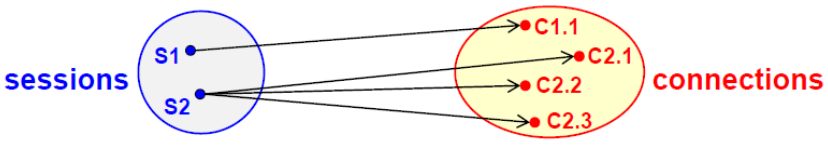
TLS handshake protocol	TLS change cipher spec protocol	TLS alert protocol	*application protocol* (e.g. HTTP)
TLS record protocol			
reliable transport protocol (e.g. TCP)			
network protocol (e.g. IP)			

- Transmitting **application protocols** like HTTP. If there's HTTPS it means that there is HTTP on top of TLS and files downloaded via HTTP are split in records and each record is put inside a TLS record.
- **Handshake Protocol** is the protocol being executed at the beginning when the TLS channel is created. It is the most *vulnerable part*, in which an unprotected TCP channel is transformed in a secured TCP channel.
- **Change cipher spec protocol** is the protocol that can be used for changing keys or algorithms without closing the channel. It is important because the handshake phase is a sensitive and complex one, but after transferring a lot of data using the same algorithm and key (e.g. for encryption) this gives the room to the attacker to perform cryptoanalysis. The more data are encrypted with the same pair cipher-key the more is the probability to understand the key. With this protocol the *system manager* can decide that after some time or some data exchanged the channel is not closed but the keys are changed. That is used even at the beginning to transform the channel from unprotected to protected.
- **Alert protocol** is the one being used every time there is a problem. For example, if a change in a packet is noticed (because the computed MAC is wrong) then a **last message** is sent: the alert "MAC is wrong" and then the **channel is closed**. It is not possible to recover from the error. An ACK is not needed (because there could be a MITM and maybe the alert will never reach the destination). The error means that there was an attack in between so an alarm for the security manager could be set to perform investigations.

All these protocols mentioned use the TLS record protocol.

Since the handshake phase is a very heavy one, it is possible to reuse the same keys and algorithms several times. That is named a TLS session. It is important to understand the relation between sessions and connections and the keys.

A **TLS session** is a logical association between client and server, which is created with the Handshake Protocol and since it is heavy we try to have as few sessions as possible. During the session, it creates a set of cryptographic protocols (one for encryption, one for authentication and integrity) and the corresponding keys. The same session can be shared by multiple **TLS connections**. The first time that a client connects to a server a session is created, then it is closed. If another one is needed the handshake is not performed again but a previous session is reused to try to skip the handshake part and have a faster creation of the channel. One session **can** be shared by several connections, because it's up to the system manager. If speed is wanted, session is enabled (faster setup) but if security is wanted the session is refused and every time a new handshake will be performed (which is more secure). Typically, a balance between security and speed is wanted: sessions can be reused but for a limited time. On the contrary, **TLS connection** is a transient association between client and server and it uses the parameters set during the handshake for a specific session. So, there is a correspondence 1 to 1.

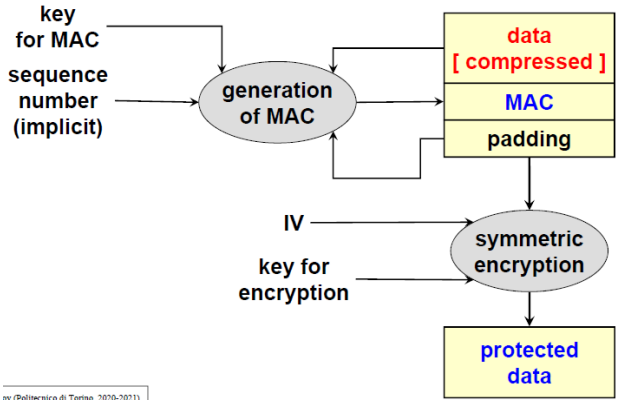


Session S1 is created and used for one connection C1.1. Then when user tries to reconnect, the session is expired, so another session is created and used to access (maybe 3 pages) with three connections.

TLS handshake protocol

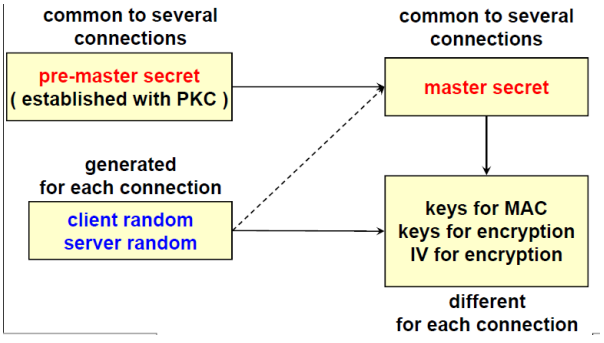
Here is where the core, the TLS security, is created. The handshake protocol is performed for several reasons:

- Agree on which algorithms we should use in this specific session for confidentiality and integrity
- Exchange **two random numbers** (one created by the client, one by the server) that will be needed for subsequent generation of the keys. It is important that these numbers are *truly random* because otherwise they can be understood in advance and so also the keys will be understandable.
- Then these numbers are used to establish a **symmetric key** by means of public key operations (RSA, DH, ECDH, …).
- Once keys and algorithms have been defined, a **session identifier** is established. If a session-id will be reused in the future, keys and algorithms will be remembered.
- Since there are public key operations for authentication, certificates are needed and exchanged to trust the public keys.



There are two ways of protecting data in TLS, depending on the TLS version. Until version 1.2 the prevalent way of protecting records was *authenticate-then-encrypt*. It means that in the beginning the data (which may be compressed, but it is discouraged) are in input to the MAC algorithm together with the key that has been negotiated during handshake, the sequence number (which is implicit) and the *padding* since we know how long the data are, how long will be the MAC and a multiple of the block of the encryption algorithm is needed. Padding enters the computation of the MAC, if we compute the MAC without padding someone could change the padding and give bad results. This is the basic record which is now encrypted (only if we negotiated confidentiality) with a symmetric encryption over everything using an IV (Initialization Vector) and the key. Looking the picture, the first upper part is compulsory, while the second one on the bottom right is optional.

We could notice that if the record is very big an attacker could try to perform some DoS because when we receive the record, we must first decrypt (which takes time), then the MAC is computed and then we discover that the record has been manipulated and so discarded. We wasted time performing decryption only to discard the packet in the end. So, for DoS protection we would prefer *encrypt-then-authenticate*.



The relationship among keys and session is the one represented in the picture on the right. When the handshake is performed, there is the negotiation of a **pre-master secret**, which is a secret established using public key cryptography (RSA, DH, …) so a secret key is established, which is typical of the session and common to several connections. Then, for each connection including the first one in which the handshake is performed, two random numbers are generated: **client random, server random**. These two numbers are used together with the pre-master secret to create the **master secret**, which is another key. While the pre-master secret is the same for the client and the server, the master secret **is different** (server will use server random to create master secret, client will do so with the client random). In this way we have 2 different master secrets common to several connections. (**BEWARE**: *This is what Lioy said, but on the RFC it says that the master secret is created using together client random + server random, so it will be the same for both*). Recap: pre-master secret and master secret are typical of a session. Then, when a new connection is opened by using a previous session the master secret is taken together with the random numbers and a specific KDF (key derivation function) to **create keys needed for MAC, keys needed for encryption and the IV for**

encryption. These will be different for each connection and *different for each direction* (keys client to server are different from keys server to client). Otherwise, an attacker could take a packet going in a direction and send it back to the other direction (copy attack). Using different keys for the direction will make this kind of attack impossible.

Perfect forward secrecy

Imagine that server has got an RSA certificate, which is good for both encryption and authentication. The problem is that if an attacker is copying all the traffic which is encrypted with the key that has been exchanged with that RSA certificate and then it makes a very long computation, the attacker may discover the private key and then decrypt *all the past traffic* sent before.

Server has got the pair (SK, PK). For each connection there will be new keys: $Enc(PK, K1)$, $Enc(PK, K2)$, … And then the keys are used to encrypt data: $Enc(K1, Data1)$, $Enc(K2, Data2)$.

The attacker may copy all these data without knowing the keys and start factorization taking the PK (public key) which is well known. If after some time the attacker success in factorization, then the SK (Secret Key) will be discovered and can be used to decrypt all the handshake packets to get all the keys and then decrypt all the traffic. On the contrary, if perfect forward secrecy is achieved, the compromise of one private key will compromise only the **current traffic** (eventually the future traffic) but **not** the past one.

The problem is that the server has a fixed certificated pair of keys (SK, PK) and generating these keys take time and must also be certified (X.509 needed). To achieve perfect forward secrecy is to use **ephemeral mechanisms**.

The way these mechanisms are performed in TLS is that rather than having a fixed key pair of keys there is the generation of key pairs. The server, every time there is a new session (new handshake) it will **create a new key pair** for itself, but it is not certified. The server one long term key pair which is the one associated with the X.509 certificate, but then it will generate new key pairs on the fly as needed. The solution is to take the generated public key to create a digital signature using the long-term secret key. In this way, digital signature is associated to the X.509 certificate and the server will create the signature "this is my temporary key for this handshake". It's like the server will be a CA for itself.

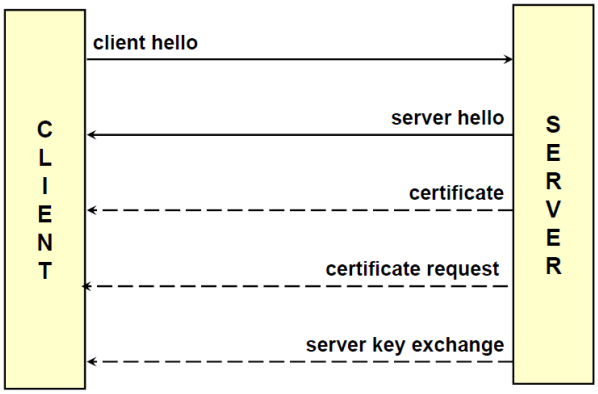
Another problem is that generating the key pair is a slow operation, especially if using RSA (DH is more suitable). If ephemeral keys are RSA this will take time, so the key pairs will not be created for each handshake, but one single pair will be used for some time or for a limited number of handshakes. If this is unwanted, Diffie Hellman is preferred since it is faster.

Using this method, if the private key of the server is discovered it is meaningless since all the other keys will not be discovered. On the contrary, if the temporary ephemeral private key is compromised, then the attacker can decrypt only that traffic and nothing more. The compromise of the long-term key is a problem only for server authentication and not for confidentiality. The ephemeral keys will use the well-known algorithms but with the "E" at the end, e.g., ECDHE (Elliptic Curve Diffie Hellman Ephemeral) which means that the key exchange happens with ECDH but the parameters are not long term but created on the fly and signed with the private key of the server.

Client hello

There are several messages being exchanged among the different parties. The message with a continuous line are *always present*, while the dashed one means that sometimes those message may not be present, since handshake can be performed in different ways.

The first message sent from client to server is the **client hello**, in which the client tells the server various things.



- The **version of the protocol preferred by the client**, that should be the most recent one
- **28 pseudo-random bytes** (*Client Random*)
- A **session identifier** (session-id)
    - If a new handshake (no previous session) is performed, the value is 0.
    - If it is different from 0 if the client is asking to resume the previous session.
- **List of "Cypher suite"** which is the set of algorithms that client is able to use for encryption, key exchange and integrity. It contains **all** the algorithms that client is able to perform.

Client is not deciding anything, just **proposing and listing** its capabilities. All decisions are taken by the server.

Server Hello

The **server hello** is the response of the server, which contains:

- **SSL version selected by the server**, which can be the one proposed by the client or a lower one in case server is not able to use client version.
- **28 pseudo-random bytes** (*Server Random*)
- **A session identifier** (session-id)
    - It is a new session-id if there was a session-id=0 in the client-hello or the server reject the session-id proposed by the client
    - Session-id proposed by the client if the server accepts to resume the session
- **"cipher suite" chosen by the server**, which should be the strongest one in common with the client
- **Compression method** chosen by the server if compression is enabled

At the end of this phase there are: the algorithms, random numbers and the identifier of the session, eventually the method of compression.

Cipher suite

Cipher suite is typically a complex string which summarizes all the algorithms, it includes *key exchange algorithm, symmetric encryption algorithm, hash algorithm (to compute MAC)*. Some examples are:

A. $SSL\_NULL\_WITH\_NULL\_NULL$
    a. This is the same cipher suite in action at the beginning: no protection.
B. $SSL\_RSA\_WITH\_NULL\_SHA$
    a. Keys will be exchanged with the RSA, no ephemeral keys. There will not be encryption but there will be the MAC based on SHA1 (SHA == SHA1 in this case).
C. $SSL\_RSA\_EXPORT\_WITH\_RC2\_CBC\_40\_MD5$
    a. SSL exported from the United States should obey some rules. RSA_EXPORT means that RSA is possible, but the key is long at most 512 bits. In the same way for encryption is possible to use RC2 but only with keys of 40 bits.
D. $SSL\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA$
    a. In this case RSA is used with 3-DES with mode Encrypt-Decrypt-Encrypt, then CBC and SHA.

The full list of all cipher suites is available on the IANA website. It could happen that connection is not established because client and server have no cipher suite in common (rare but possible).

Certificate (server)

Once the initial parameters are established, since server authentication is compulsory, an important message is the certificate message sent by the server. This is the certificate for server authentication: it contains the long-term public key of the server with a certificate, and the certificate must contain in a specific field the name of the server itself which can be in the subject field of the X.509 certificate or in the SubjectAltName but it must be the *same* name of the identity of the server. If we're connecting to "$https://www.polito.it$" and an X.509 certificate is received which contains the subject "$polito.it$" that will not work. In the same way if the IP address of website is 130.172.15.82 and we access using it, there will be a mismatch. The certificate

can be used only for digital signature or also for encryption, this is described in a specific field *KeyUsage*. If the certificate submitted in this phase contains only flag for signature, then the phase named *Server Key Exchange* is needed.

Certificate request

If the server is requesting client authentication, then it will send an explicit message to the client which is the *Certificate Request*. In this message the server is not only requesting the certificate of the client but also tells from which CA the certificate must be issued, because server trusts only some CAs.

Server key exchange

It is present only if the certificate of the server is valid only for signature. It carries the server public key for key exchange. It is needed only in the following cases:

- The RSA server certificate is usable only for signature
- Anonymous or ephemeral DH is used to establish the pre-master secret
- There are export problems that force the use of ephemeral RSA/DH keys
- Fortezza (old key agreement protocol)

This is the **only** message explicitly **signed by** the server. If we have this message, it **doesn't mean** that we have only ephemeral keys, but maybe RSA for signature and authentication is wanted and then also DH, but since DH keys are not certified the server needs to explicitly sign them.

Certificate (client)

In case the server requested the client authentication, then there is the certificate client message which contains the certificate for client authentication (typically browser or user using browser). It must be issued by one of the CAs requested in the certificate request.

Client key exchange

When client has got all the information to create keys, the client generates symmetric keys and send them to the server in various ways:

1. It creates the pre-master secret encrypted with the server RSA public key (ephemeral or key from its X.509 certificate) if it is the first connection of the session (handshake).
2. It can simply send the public parameters of DH keys
3. Fortezza parameters

When server receives this message also the server can generate the pre-master secret.

Certificate verify

The client is now authenticated. It is an explicit signature done by the client with the hash computed over all the handshake messages before this one and encrypted with the client private key. The signature is created using the client private key. Since the public key was already received, server can verify that the client possesses the corresponding private key and will be the right client because the signature will contain all the previous messages and in some sense is also authenticating the handshake. Used only with client authentication (to identify and reject fake clients) but this is **optional**.

Change cipher spec

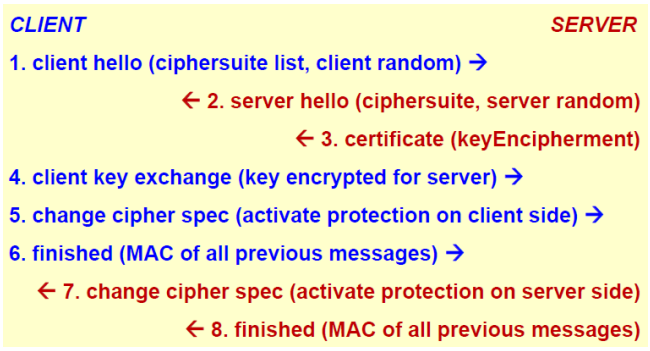
If there is no client authentication, how can we verify that nobody has altered the handshake? Because if there is a MITM it may have changed the algorithms (e.g., RC2 instead of AES) to easily decrypt the traffic. To avoid this, *change cipher spec* is used. It triggers the change of the algorithms to be used for message protection and allows to pass from the previous unprotected messages to the protection of the next messages with algorithms and keys just negotiated. Theoretically is a **protocol on its own** and some analysis suggest that it could be eliminated since it just signals that from the next messages algorithms and keys will be used (could be automatically understood to make connection faster).

Finished

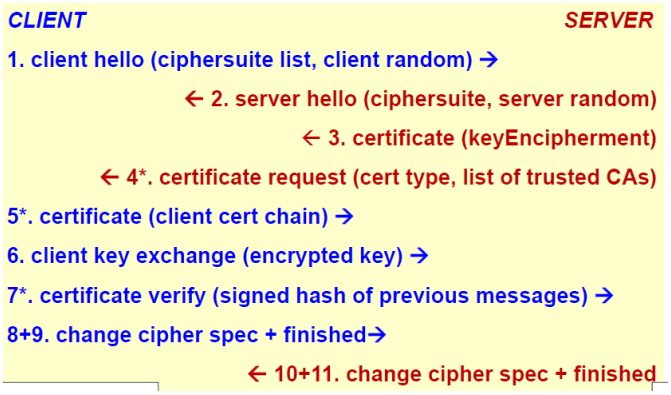
This message declares that handshake has completed, and it is the **first message protected** with the negotiated algorithms. All previous messages were not protected in any way. This is an important message because it contains a MAC (Message Authentication Code) computed over all the previous handshake messages with the exclusion of the change cipher spec (since it has no information). The hash is computed and protected with symmetric key, using as a key the master secret. This prevents *rollback MITM* attacks which is *version downgrade* or *ciphersuite downgrade*. The content of the finished message is **different** for client and server because they are computing the hash of the messages that they sent (in one case messages from client to server, in the other case from server to client) (**BEWARE**: *this is what Lioy said, but the RFC says that the Finished message's content is the same but the second one also contains the hash of the first one*). If these finished messages are correct, we know that we have negotiated the correct keys and authenticated server (possibly also client) and no one manipulated in any way the handshake protocol.

TLS Handshake examples

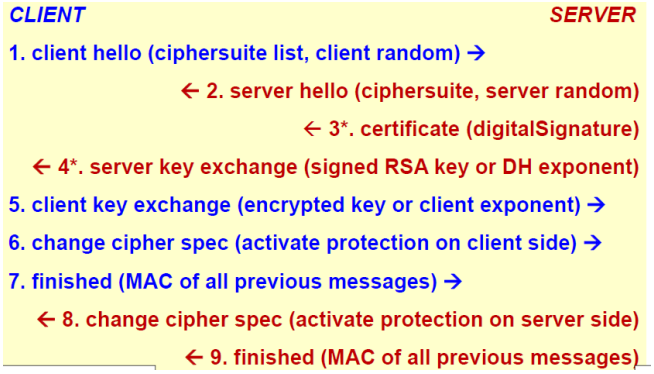
In the first example the TLS channel is created without ephemeral keys and without client authentication. It is the simplest possible connection. The client sends the *client hello* which contains the list of acceptable cipher suites and the client random. The server selects one cipher suite and sends also the server random. The server sends also its own certificate which contains the flag also for encryption and not only for signature (which is implicit and must always be present). In this way, the client can send the message *client key exchange* in which the client has generated the pre-master secret and it's sending it encrypted with the public key of the server. At this point client and server got the pre-master secret and random numbers, so the keys are computed with KDF defined in the protocol. The client announces that from the next message it will use the keys and the algorithms negotiated and sends also the *finished* message which is the MAC computed using the MAC algorithm in the cipher suite and the computed keys. Server replies with *change cipher spec*, which means that also the server will start using keys and algorithm and sends again the *finished* and so the MAC computed over all previous messages sent from the server to the client.



In the next example the TLS channel is created without ephemeral keys but with client authentication. Client hello, server hello and server certificate are the same as before. The new message is the *certificate request* sent from server to client, which contains the kind of certificate requested and the list of trusted CAs. Client replies with the *certificate* message which contains not only the single certificate, but the *whole chain* of intermediate CAs up to the root CA excluded, root CA is never sent since it must be trusted. Then there is the *client key exchange* in which there is the pre-master secret encrypted with the public key of the server and then the *certificate verify*. Why the 7 is not in place of the 6? We wait and first perform client key exchange and then we perform signature for additional protection, since signature at 7 computes the hash over all previous messages, more messages are computed the better is the protection. Then there are *change cipher spec, finished* from bot client and server.

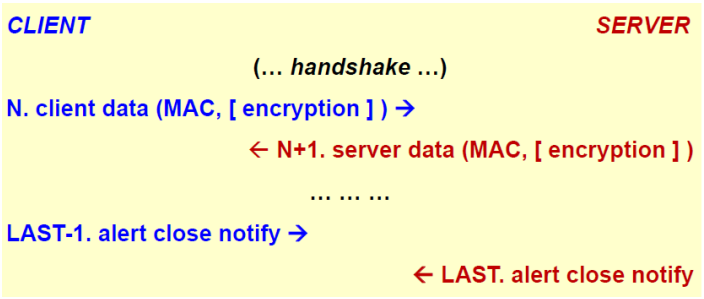


In the next example the TLS channel is created using ephemeral keys and no client authentication. If ephemeral keys are present, the message number 3 changes, because the certificate sent by the server is good **only for digital signature** so server cannot create pre-master secret on the client and send it encrypted with the public key of the server. Now, server sends ephemeral key (RSA, DH) signed with the public key which is certified. Client knows which is the public key to use to create/exchange the public pre-master secret, so the client key exchange is performed but the public key used in message number 5 is not the public key of message 3 but the public key of message 4, since is the ephemeral key. At this point, there is again *change cipher spec* and *finished*. The difference is only in the certificate valid just for signature and the ephemeral key signed by the server for authenticity.



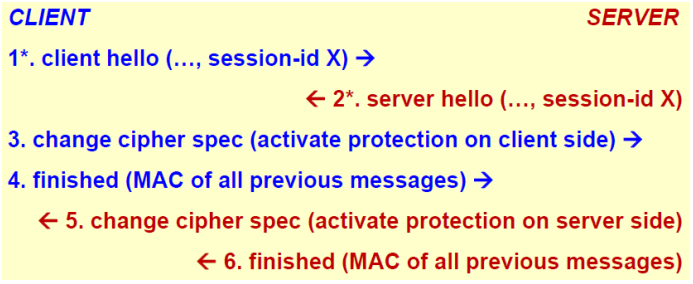
Data exchange and link teardown

After handshake is performed at some point there is the Nth record in which the client sends data which contains MAC and maybe are encrypted. Server replies with another record containing server data containing MAC and optionally encrypted. At some point the client wants to close the channel and sends the message corresponding to the **alert protocol**, which is used not only for attacks but also for signaling. The alert is the **alert close notify**. The last message from the server is the **alert close notify**. We don't wait for an ACK, since there could be a MITM which may stop our messages.



Resumed session

If we want to resume the session in the *client hello* rather than sending session-id 0 we send the number X of the session that we want to resume. If server accepts, the server replies with the same number X in the *server hello*. Then everything is skipped, since keys and algorithms have already been negotiated. There's only the change cipher spec and finished messages. It is faster but still has the MAC computation needed.



TLS setup time

Security has got a price which is typically high to pay. The problem with TLS secure networking is the **setup time**. First the TCP handshake is needed since we need a TCP channel. Then TLS handshake is needed, which is composed by various messages that can fit in a single TCP segment. Typically, this requires 1-RTT for TCP and 2-RTT as following:

- (C>S) SYN
  - (S>C) SYN-ACK
- (C>S) ACK + ClientHello (since ACK is just a flag, we can piggyback also clientHello)
  - (S>C) ServerHello + Certificate
- (C>S) ClientKeyExchange + ChangeCipherSpec + Finished
  - (S>C) ChangeCipherSpec + Finished

After 180 ms client and server are ready to send protected data (assuming 30ms delay one-way). For some applications this is too much, so we need to gain less latency.

TLS 1.0 (SSL 3.1)

The first version of Transport Layer Security that has been standardized was TLS 1.0 by IETF and it is 99% coincident with SSL-3 but with emphasis on standards (i.e. not proprietary, in the past it used proprietary algorithms) digest and asymmetric crypto algorithms.

- Key exchange with DH + Server authentication with DSA (digital signature algorithm) + 3DES
- Rather than using custom keyed digest, TLS always uses HMAC-SHA1

It corresponds to the cipher suite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA.

- DHE = Diffie Hellman Ephemeral
- DSS = Digital Signature Standard for signing DH parameters and authenticating the server
- 3DES using Encrypt-Decrypt-Encrypt (EDE) CBC
- SHA1 to compute MAC

TLS 1.1

This version was mainly developed to **protect against CBC attacks**. CBC can be attacked if some parts of the message are guessed. In order to prevent those attacks, the implicit IV is replaced with an **explicit IV** and errors in padding now use a generic message *bad_record_MAC* rather than *decryption_failed*. This is important because if there is an error, we don't have to give to the attacker too much information.

In addition, TLS 1.1 introduced also the **IANA registry** defined for protocol parameters**,** and if there is a premature close due to a network problem this does not prevent resuming a session. An attack that an attacker can do is that while using a TLS channel the packets are killed (e.g. using a RESET in TCP) so that channel needed to be opened again. In the past, if channel was broken and not regularly closed that implied that session cannot be resumed. On the contrary, with TLS 1.1 the specification is that the channel was closed due to a network problem the **session can be resumed**. Only if the channel was closed due to an attack (e.g. alert bad MAC) session *can't be resumed* since the attacker may have manipulated the session.

TLS 1.2

This version is currently the most used version even if it is not the most recent one.

- The cipher suite now specifies also the PRF (pseudo-random function) needed to client server and client random. It is no more left to the vendor.
- It is possible to negotiate the hash algorithm used in the MAC but by default SHA-256 is used (e.g. in the finished message) since SHA1 has been attacked.
- Optionally, it is possible to have *authenticated encryption* (rather than authenticate-then-encrypt) which means that it is possible to use AES in GCM or CCM mode.
- TLS 1.2 incorporates the protocol extensions and the AES ciphersuite.
- The default cipher suite is now TLS_RSA_WITH_AES_128_CBC_SHA. The IDEA and DES based cipher suites have been deprecated.

TLS evolution

TLS evolves not only through the definition of new versions but also through auxiliary RFCs. There are various documents to introduce new cipher suites for encryption such as AES, ECC, Camellia, SEED and ARIA and for authentication such as Kerberos, **pre-shared key** (secret, DH, RSA) which possibly avoid Public Key Cryptography, RSP (Secure Remote Password) and OpenPGP which is a variant of the DS often used for electronic mail security. For compression there are 2 RFCs: compression methods + Deflate and protocol compression using LZS (Lempel-Ziv-Stac). Other RFCs are:

- Extensions (specific and generic)

- User mapping extensions
- Renegotiation indication extensions
- Authorization extensions
- Prohibiting SSL-2
- **Session resumption without server state**: if client wants to resume a session, it means that server must have in memory all the parameters to resume that specific session (which is a huge quantity of memory if there are hundreds of clients). This RFC talks about how to enable session without storing them at the server.
- Handshake with supplement data

Be aware that these documents are **extensions** so no guarantee that it is possible to find then in all clients and servers, but only some of them implement those features.

Heartbleed

One of the most famous attacks to TLS is Heartbleed. It is based on the **TLS/DTLS heartbeat extension**. Since handshake takes a lot of time (even if session is reused) since we close and re-open the channel multiple times. For this reason, for optimal performance, this extension keeps the channel alive even if there's nothing to exchange. It is just a message sent over the time to avoid the end of the channel. These messages are also useful in Path-MTU discovery important in IPv6.

There is a problem named **CVE-2014-0160** which is an OpenSSL bug named **buffer over-read**. It works in this way:

- $C \rightarrow S$ "Server, are you still there? If so, reply "POTATO" (6 letters)
- $S \rightarrow C$ "POTATO"
- $C \rightarrow S$ "Server, are you still there? If so, reply "BIRD" (4 letters)
- $S \rightarrow C$ "BIRD"
- $C \rightarrow S$ "Server, are you still there? If so, reply "HAT" (500 letters)
- $S \rightarrow C$ "HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants…"

Using this vulnerability, TLS server sends back more data (up to 64KB) than in the heartbeat request. Attacker can get sensitive data stored in RAM such as user+password and/or server private key (if not using HSM).

This doesn't mean that TLS is a bad protocol, but it is an implementation problem.

Bleichenbacher attack (and ROBOT)

In 1998 Daniel Bleichenbacher performed the "million-message attack". He discovered a vulnerability in the way RSA encryption was done in TLS (while taking pre-master secret and encrypting it with public key of the server). The attacker can 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. **This means that performing a private key operation without knowing the private key means the ability to decrypt the pre-master secret and to sniff the traffic.** The attack refined over the years and in some cases only requires only thousands of messages (feasible from a laptop).

ROBOT in 2017 is a variant of Bleichenbacher's attack that affected major websites (including facebook). That is possible even if using a hardware security module because it's not that it is possible to read the private key but performing some computation without knowing the key.

Other attacks against SSL/TLS

CRIME (2012) is an attack where the attacker is able to **inject chosen plaintext in the user requests** and measure the size of the encrypted traffic. Doing that, it may recover specific plaintext parts exploiting information leaked from the compression. This tells that **we should not enable compression**. Compression works by reading the message and if similar pieces are found they are replacing with one common element. If

an attacker is able to inject other text before TLS and it observe the traffic, it is possible to guess e.g. a password by looking at the dimension of the message: if the injected message is the right password the message will be just 1-2 bytes, while if the guess is bad the message will increase a lot.

BREACH (2013) deduces a secret within HTTP responses provided by a server that uses HTTP compression, inserts user input into HTTP responses and contains a secret (e.g. a CSRF token, ticket or cookie) in the HTTP responses. In this case the problem is the compression of HTTP.

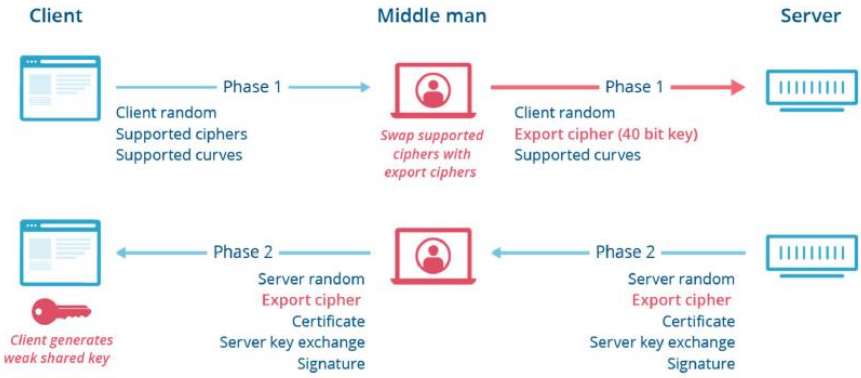
BEAST (Browser Exploit Against SSL/TLS) in 2011. It happened in an SSL channel using CBC with IV concatenation. A MITM may decrypt HTTP headers with a blockwise-adaptive chosen-plaintext attack where the attacker may decrypt HTTPS requests and steal information such as session cookies.

POODLE (Padding Oracle On Downgraded Legacy Encryption) is a MITM attack that exploits SSL-3 fallback to decrypt data. The variant in December 2014 exploits CBC errors in TLS-1.0-1.2 (so it works even if SSL-3 is disabled).

The FREAK attack

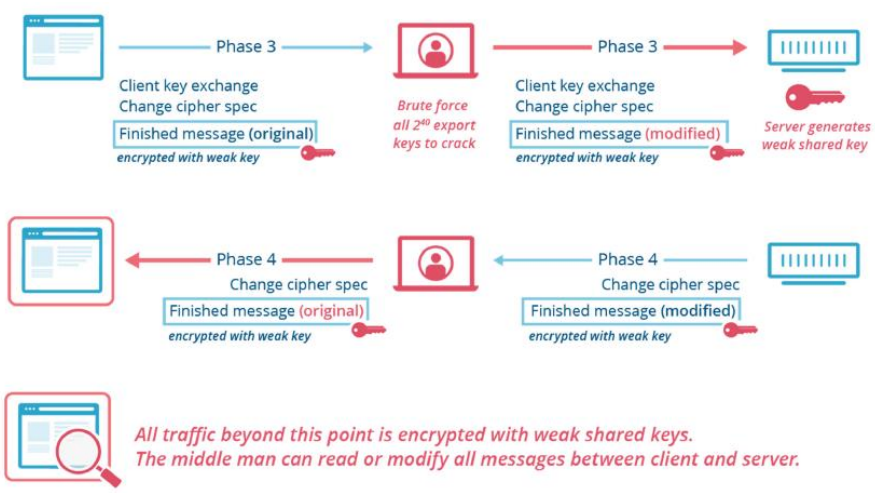
With **FREAK** (Factoring RSA Export Keys) in 2015 an attacker is able to downgrade to export-level the RSA keys (512-bit) which then permits factorization and channel decryption or to downgrade to export-level symmetric key (40-bit) then brute-force and crack the key.

As a consequence of all these attacks SSL-3 has been disabled on most browsers (e.g. since FX34) or may be disabled. For example, on Firefox (FX) is is possible to specify: *security.tls.version.min* = 1 (i.e. TLS-1.0, alias SSL-3.1) but SSL-3 is needed by IE6 (which is the last version available for Windows XP) and TLS cannot be disabled. In general, there should be test browser/server with Qualys SSL labs tests.



The client sends the client random, the list of supported ciphers and supported curves. A MITM deletes the list of the ciphers and inserts **export cipher (40 bit key)** which should not be used. The server receives the original client random and supported curves, but the changed cipher. Server answers with server random and then it must select one common algorithm, which can only be the weak one. Server can **refuse the connection**, but then client is impossible to connect or if server is badly configured, will accept, sending also certificate, server key exchange and signature. In this way **client generates a 40-bit key** that will become the shared key.

Client will perform client key exchange, change cipher spec and the finished message protected with the weak key. The *finished* is computed over what the client sent, but client sent a good list of ciphers. This would be the moment in which the server would detect the manipulation. Since the finished message is encrypted with the weak key, there are 2^{40} attempts to do. If attacker tries all possible keys then the correct one will be found to recompute the finished message, sending the modified message to be in agreement with the attacker changes. Then the server will generate the weak shared key and will use it. The attacker will be able to **sniff all the traffic** and to **create fake messages** since fake keys are used also for authentication and integrity. This example shows one way to manipulate the handshake without noticing.



Other attacks against SSL/TLS

There are other attacks against SSL/TLS, several of them are due to **implementation errors** so they have nothing to do with the protocol in general, for example:

- *Heartbleed*, *BERserk*, *goto fail*;
- *Lucky13* (feb-13) is an evolution of the Vaudenay's attack which inspired the creation of TLS 1.1. It is a side-channel attack based on time: a side channel attack is an attack in which you look at something else and in this case this is a **timing** side channel because by looking at how much time the client or the server take to perform a certain operation it is possible to deduce the number of zeros and ones present in the private key. Lucky13 is a variant of the Vaudenay's attack that worked even if the Vaudenay's attack was fixed.
- *Lucky microseconds* (nov-2015) is another variant of Lucky13 created to attack one specific implementation named **s2n** that is the Google TLS library claiming to be more secure and resistant to Lucky13.

There are also attacks that exploits protocol design errors:

- *SLOTH*, *CurveSwap*: these attacks are theoretical, so they exist but they are impossible or nearly impossible to implement.
- *WeakDH*, *LogJam*, *FREAK*, *SWEER32*: they require high computational resources.
- POODLE, ROBOT: dangerous attacks that do not require high resources.

As we have seen there are a lot of attacks against SSL/TLS, so how can we test for all of them? There is a free service that tests provided by Qualys: https://www.ssllabs.com/ssltest/.

TLS False Start

Another option for TLS which is defined in RFC-7918, which is an extension, so it is not mandatory to use that, is *TLS false start*.

This extension focuses on the problem of *how much time does it take to setup the TLS channel* and sending information: with TLS False Start the client is permitted to send **application data** together with the **ChangeCipherSpec** and **Finished** messages, **in a single TCP segment**, without waiting for the corresponding server messages.

ChangeCipherSpec and *Finished* messages **can stay together**, but we're now taking a risk because we are already sending data, which in principle is possible, but since there's no waiting for the finished message sent from the server to the client than we have a problem, because if that finished message is telling that the handshake has been manipulated and user has already sent data, than maybe these data have been decrypted for example.

So, we have an advantage because the latency of the setup is reduced to 1 RTT but this risk is accepted.

TLS False Start should work without any changes but there are some caveats:

- Chrome and Firefox require **ALPN** + **perfect forward secrecy** to activate TLS False Start.
- Safari requires only perfect forward secrecy.

To enable TLS False Start for all browser the server should:

- Advertise the supported protocols (via ALPN, application level protocol negotiation, e.g."h2, http/1.1").
- Be configured to prefer cipher suites with forward secrecy, since it is required by the majority of the browsers.

TLS False Start has nothing to do with security, it deals with speed and performance, but it is coming the risk. The decision to use it or not is taken **server-side**, and if it is enabled the client will eventually use it if the server is well configured.

TLS session tickets

This is another extension defined in RFC-5077. To continue a previous session (so without repeating the handshake) it requires to cache the session-ID at the server, but for each session-ID we need to store also the session parameters negotiated during the handshake, and if the server is a very successful one, then this cache may become very large for high traffic servers.

TLS session ticket is an extension that permits the server to send the session data to the client, so rather than the server keeping session data on its own memory (which is more secure and safe but heavy for high traffic servers) they are protected by the server and stored on the client side, so session data are:

- **Encrypted** with a **server** secret key (which is **not** the key being used to protect the TLS channel).
- Returned by the **client** when resuming a session.
- The concept is that in practise this moves the session cache to the client and does not overload the server.

It works only if it is supported by the browser, since it is an extension. Moreover, if the server is not just one but we are in a load balancing environment (several servers), then there is a problem because the session ticket could be received from Server 1, but then the load balancer will send next session resumption to Server 2 that will not have the key. So, if there is a group of servers load balanced it requires **key sharing among the various end-points**, that is more risky so we need also periodic key update.

TLS 1.3

TLS 1.3 is the newest version of the protocol released in August 2018 (RFC-8446): this new version has been trying to perform several improvements, for example:

- **Reducing handshake latency**: the issue is that typically it is needed to create a new TLS channel for each connection so reducing setup time is important.
- **Encrypting more of the handshake**: some of the attacks are possible because the handshake is completely in clear (for example FREAK attack is a problem because the man in the middle can read the list of ciphers), so TLS 1.3 is trying to encrypt more of the handshake, both for security but also for privacy because as part of the handshake the client can send his certificate, that could identify the person using browser.

- **Improving resiliency to cross-protocol attacks:** attacks between HTTP and TLS.
- **Removing legacy features** that are no longer needed

TLS 1.3: key exchange

In TLS 1.3 the key exchange part has been heavily changed: RSA key exchange is **no more permitted**, so RSA keys can still be used but **only for server authentication** not for key exchange. The point is that RSA key exchange is not enabling **perfect forward secrecy,** and this is also difficult to implement correctly, in addition in this way we are automatically protected against Bleichenbacher and ROBOT attacks that exploits RSA key exchange.

Rather than using RSA key exchange TLS 1.3 uses **Diffie Hellman ephemeral,** but arbitrary parameters are not permitted because there are attacks such as LogJam and WeakDH that forced the server to use small numbers for DH (just 512-bits), so the attacker was able to attack DH exchange.

In 2016 the researcher Sanso found that OpenSSL generates DH values without the required mathematical properties (implementation issues).

To avoid those kind of problem TLS 1.3 uses Diffie Hellman ephemeral with **few predefined groups** so clients and servers can only choose among these groups from a list (and not create parameters on their own).

TLS 1.3: message protection

Previous pitfalls:

- the use of **CBC mode** and **authenticate-then-encrypt** were the source of Lucky13, Lucky microseconds and POODLE attacks
- The use of **RC4** (stream encryption algorithm) leads (in 2013) in an attack in which the plaintext was recovered due to measurable biases, that are some bits created more easily than others.
- The use of compression at TLS level was the culprit for CRIME attack

To avoid these problems TLS 1.3 does not use CBC mode, does not use authenticate-then-encrypt and it **uses only AEAD (**Authenticated Encryption with Additional Data**)**.

In addition, TLS 1.3 completely dropped RC4, 3DES, Camellia, MD5 and SHA-1 support, and it uses only modern cryptographic algorithms and **no compression anymore**, which has been disabled. Unfortunately, there is still the problem if compression is used in application level (e.g. HTTP), but at least the compression within TLS itself has been deleted in TLS 1.3.

TLS 1.3: digital signature

Previous pitfalls:

- **RSA signature of ephemeral keys**: that was needed but that was done wrongly with the PKCS#1v1.5 schema.
- The **handshake was authenticated with a MAC** and not with a signature and this made possible some attacks such as FREAK.

TLS 1.3 uses:

- **RSA signature** with the modern secure **RSA-PSS schema**.
- **The whole handshake is digitally signed**, not just the ephemeral keys.
- In general, it uses the **modern signature schemes** resistant to several kinds of attacks.

TLS 1.3: ciphersuites

In TSL 1.3 rather than specifying a complete ciphersuite (that is huge set), it specifies only some **orthogonal elements**, in particular the ciphersuite specifies:

- Cipher (& mode) + HKDF (Hashed Key Derivation Function) hash.

- **No server authentication algorithm is specified** because the signature of the server is implicit, so his certificate will always be sent and it contains the type of algorithm (it can only be RSA, ECDSA or EdDSA) used for the signature.
- **No key exchange** specified, there is a fixed list composed of DHE/ECDHE, PSK (Pre-Shared Key), PSK+DHE/ECDHE.

This means that we have only 5 ciphersuites in TLS 1.3:

- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_ SHA256 (Chacha20 is the only permitted streaming algorithm)
- TLS_AES_128_CCM_SHA256
- TLS_AES_128_CCM_8_SHA256 (*deprecated*, 8 means a reduced authentication of data)

TLS 1.3: EdDSA

EdDSA (Edwards-curve Digital Signature Algorithm) which is similar to EC but not exactly equal. The problem is that the **DSA requires a PRNG** (Pseudo Random Number Generator) that can leak the private key if the underlying algorithm for random number generation is broken or made predictable. EdDSA **does not need a PRNG** but it picks a **nonce based on a hash of the private key and the message**, which means that after the private key is generated there's no more need for random number generator, on the contrary ECDSA needs the generation of a PRN for each signature, on the contrary, in EdDSA the PRN is generates only once for the generation of the private key. Then for computing signature there are no PRN.

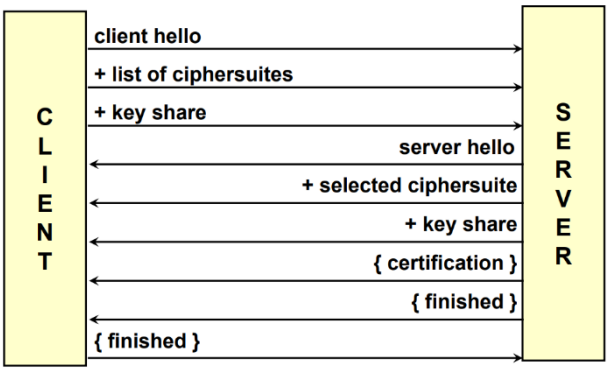
In general, with EdDSA we have **N-bit private and public keys and 2N-bit signature length** (signature is the double of the size of the private key).

For example, Ed25519 uses SHA-512 (sha-2) and Curve25519 with a 256-bit key, 512-bit signature and it is equivalent to 128-bit security (equivalent to AES-128 in terms of security).

Ed448 uses SHAKE256 (SHA-3) and Curve448 which has 456-bit key, 912-bit signature and 224-bit security (close to AES-256 security).

TLS 1.3: handshake

In TLS 1.3 the handshake has been simplified. The **client sends three messages**: the *client Hello*, the *list of ciphersuites supported* and its *part of the key share*. The server answers with *server hello*, the *selected ciphersuite*, its *portion of the key share* and then the "{}" in the picture means that *certification and both client-server finished* are **encrypted.**



Notes about TLS 1.3 handshake:

- For backward compatibility, TLS 1.2 messages are also sent and most of TLS 1.3 **features are in message extensions**, in this way if the client is capable to use TLS 1.3 it will understand the extensions and fully use TLS 1.3, vice versa it will use TLS 1.2.
- Basically, it's a **1-RTT handshake** that can be reduced to *0-RTT* upon resumption of a previous session (or by using Pre-Shared Key, which is rare). If a session is resumed only finished and then data is sent.

Some notations for the following arguments:

- $\{ data \}$ = protected by keys derived from a [sender]_handshake_traffic_secret (temporary key)
- $[data]$ = protected by keys derived from a [sender]_application_traffic_secret_N (long term key)

TLS 1.3: handshake: client request

The **Client Hello** contains:

- Client random
- Supported protocol version
- Supported ciphersuites

The client request also contains extensions for key exchange:

- key_share which is the portion of the client of (EC)DHE
- $signature\_algorithms$ which contains the list of supported algorithms
- $psk\_key\_exchange\_modes$ that contains the list of supported modes for key exchange
- pre_shared_key that is the list of pre shared key offered, if any (could be missing if there's no pre-shared key)

TLS 1.3: handshake: server response

The **Server Hello** contains: *Server random*, *Selected protocol version*, *Selected ciphersuite*.

In the extensions for key exchange there is:

- key_share is the portion of the key generated by the server using (EC)DHE
- pre_shared_key = selected PSK, if any

And then the server will send the **server parameters:**

- { $EncryptedExtensions$ } contains the response to all the non-cryptographic (not related to crypto parameters) client extensions
- { $CertificateRequest$ } that optionally request for client certificate

For **server authentication:**

- { $Certificate$ } is an X.509 certificate or a **raw public key** (instead of PKI) which is permitted by RFC-7250. This feature simplifies the management of the server since it does not need any more a certificate, but on the other side there are all the risks connected to it, since server is not using a certified public key. It means that the public key must be pre-installed in the browser of the client and must be protected.
- { $CertificateVerify$ } is the signature computed over the entire handshake with the private key corresponding to the certificate of the previous point.
- { $Finished$ } is the MAC over the entire handshake

All the elements between "{}" are encrypted using the **temporary handshake key.** Finally, the server can send the **application data** encrypted with the **session key** (the application-level key).

TLS 1.3: handshake: client finish

The client finish can send the client authentication if it was requested:

- { $Certificate$ } contains the X.509 certificate (or raw key, RFC-7250, in this case the server will need to store securely all the public keys of the possible clients)
- { $CertificateVerify$ } which is the signature over the entire handshake, the **explicit challenge response**.
- { $Finished$ } is the MAC over the entire handshake encrypted with the handshake key.

And finally, the client can send **application data** encrypted with the **session key**.

TLS 1.3 / Pre-Shared Keys

In general, a Pre-Shared Key replaces the session-id and the session tickets (no more present) because we are agreeing a key in a full handshake and then we can re-use it in other connections, so instead of negotiating a session and reminding all the parameters, the new TLS is computing a key and that key can be reused in the future for other connections, so a session will be identified as several connections reusing the same key negotiated in the first connection of that session.

PSK and (EC)DHE can be used together for forward secrecy, because PSK can be used for authentication and for key agreement (EC)DHE is used. PSK could also be OOB (e.g., generated out-of-band, from a passphrase), which is risky if the passphrase is not sufficiently random so that a brute-force attack could be possible. In general, OOB-PSK is present but **discouraged**.

TLS-1.3 / 0-RTT connections

Considering innovations on TLS 1.3, one key-point is to have the shortest possible setup time, in particular to be able to create a connection ideally in *0 RTT* (round trip time). 0 RTT means that the connection is opened and simultaneously data is sent, which seams a nonsense. What 0 RTT really mean for TLS is that when using a PSK (Pre-Shared Key), which means that a session has been already negotiated and as result we got a pre-shared key, client can send "**early data**" along with its first message (*client request*). While sending the client hello and all the initial part, the client can also send some data, protected with the key derived from this PSK.

Early data is protected with one specific key, which is named ***client_early_traffic_secret***, which is different from the key that will be used to protect all the other segments. Since this key rely on past things, there are 2 problems:

- This key does not provide **Perfect Forward Secrecy**, even if the keys have been negotiated with PFS. The reason is because it is derived from a pre-shared key so something that remains the same for all the connections within the same session. All connections that use PSK do not gain PFS.
- Additionally, there is the possibility of some kind of **replay attack**. The replay attack is possible because this key is generated only at the client side, with no contribution from the server side, because the *server random* has not been received yet since it is exchanged when there is the creation of a new connection and together with the pre-master secret can generate new keys.

There are some complex mitigations possible, especially when we don't have one TLS server, but multiple ones, that need to share the same session (typically in load-balancing environment), unless the TLS terminator is placed at the load-balancer itself. When there is internet and multiple clients connecting to high load server, typically a load balancer is put in front of it so that there is just one public interface and then as many servers in the back end providing the same services. If TLS is terminated on the server, it means that one client and that server have established the session. If within the session a first connection is open and then the client opens another connection maybe the load balanced could balance that to another server, which does not have the session information (and that's the problem). So, if for security the TLS connection is terminated at back-end servers then the back-end servers need to have common databases, so that data about session must not be stored locally at the server. It is not trivial but becomes a single point of failure. Even if the load balancer has been attacked, there is still security since channel is end-to-end.

There are other cases in which TLS is not on the servers. TLS can be ended on the load balancer, which acts as a **reverse proxy**. Client connects with TLS with the load balancer, the load balancer gets the http requests and then these requests are forwarded with plain http (without TLS) to the back-end servers because it is assumed that on the back end there is some security (e.g. physical security: load balancer, switch and servers are in the same room with no other network connections; IPSec). Placing TLS on the load balancer simplifies the architecture since all data are managed on the load balancer and maybe it could be also equipped with a HSM (Hardware Security Module, a cryptographic accelerator). The price to pay is the fact that there is a single point of failure: if that machine is attacked the whole security is gone, because for a moment data remain unprotected while exiting from the protected TLS.

TLS-1.3 / Incorrect share

In TLS 1.3 the client and the server have less choice about the key exchange and the authentication algorithms. It has been restricted the set of algorithms and the possible values inside those algorithms. In particular for ***(EC)DHE***, there are specific groups. The client, who sends the list of supported groups, does not have a match with the groups supported by the server (mismatch). So, the server will send an alert message with ***HelloRetryRequest*** and the client **must** restart the handshake from the ClientHello message with other groups. If also the new groups are unacceptable for the server, then the handshake will be aborted with an appropriate alert. There are some groups that are recommended that every client and server should implement, but sometimes the server manager selects different groups, because they consider those groups more secure and that is also a way to get rid of those clients that don't share the same feeling.

TLS and PKI

Even in TLS 1.3 we need public key certificates. It means that TLS has some kind of relation with the PKI (Public Key Infrastructure) since normally the certificates are released from PKI. PKI is needed for server (always) and optionally for client authentication, **unless true PSK authentication is adopted**, where '*true*' means not the pre-shared key which has been negotiated as part of a session, but the installation of manual pre-shared keys at the client and the server (which is a rare case, like some embedded systems).

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

- **It sends the whole certificate chain** (a certificate has been issued by a certification authority and the certificate of the certification authority has been issued by another certification authority and we go up until reaching the root CA). The certificate of the peer and all those of the intermediate CA are sent, but it is **never sent** the certificate of the root CA, because **it must be already present** on the other side and must be trusted. There are some attacks in which the attacker willingly is sending also the root CA, which is typically a fake one. If the other peer accepts the root CA without checking if it trusted or not, then the attacker wins. For example, if a monitoring system for TLS transactions is created, *it should always be checked that the certificate chain never contains the root CA*.
- The peer receiving this chain must perform **certificate validation of each certificate in the chain**. Here there is another possible attack: there are some browsers and servers that just check if the certificate of the end entity (the peer) is valid, which could maybe be valid but the CA that issued that certificate is invalid. Certificate validation **must be performed recursively** for each certificate in the chain. The validation implies: correctness of signature; validity of issue and validity date.
- It is important to check if although the certificate appears to be valid it has been revoked or not. **Revocation status** is an additional control that needs to be performed at each step of the chain verification. To check if a certificate has been revoked or not, it is needed to contact an authority within the PKI with two possible strategies:
    - The **CRL** (*Certificate Revocation List*) can be used but it contains the list of **all certificates** that have been revoked (it means that the list can be quite big), so downloading and storing the CRL can be a problem, also because one CRL is needed for each step in the chain.
    - **OCSP** (*Online Certificate Status Protocol*) is better from a performance point of view since it is possible to query an *OCSP server* and ask if a certificate is valid or not. The problem here is that when the request is sent to an OCSP server there is the leakage of information (privacy problem), since the OCSP server will know which servers we are visiting.

Both CRL and OCSP requires **one additional connection** and add some delay to the setup, because it is needed to verify the certificate before establishing the session. (e.g. for OCSP +300ms median, +1s average). Definitely, there is a problem of privacy and speed.

TLS and certificate status

What should happen if the CRL or OCSP are unreachable? It can be due to a server error, a network error or due to an access block of a firewall since many companies have very strict rules set on firewalls (due to security policy or insecure channel which is typical for OCSP since the response is signed itself and sent in clear). The possible approaches are:

- **Hard fail**: the page is not displayed and give to the user a security warning. For example, "*sorry, we can't display this data, because we could not verify the revocation status of the page*". This is the **most secure thing** but is **the most annoying** for the user. This could also be caused by a (D)DoS attack against the OCSP or CRL server.
- **Soft fail**: the page is displayed anyway. Even if it is not possible to verify the certificate, **it is assumed that it has not been revoked**.

Both approaches require additional load time, even more than getting the OCSP answer because time out expiration is needed, which means that the connection is done with TCP and then the timeout is set (and there is the waiting of it to expire). In order to avoid these problems, two different solutions are used.

Pushed CRL

Most of the times revoked certificates are originated by a compromised intermediate CA. When this happens **all the certificates** that were issued by that certification authority must be revoked (a huge number), so the browser vendors have decided to push not all revoked certificates but those big, revoked certificates. If an intermediate CA becomes invalid, then it is **immediately declared** as invalid. The CRL that contains that invalidity is taken and pushed during an update to the browser. Some examples:

- *Internet Explorer* is doing that every time there is a browser update, which is not so good. If for some reasons un update is postponed, then the user will never receive the information about this CA being revoked.
- *Firefox* uses ***oneCRL*** which is part of the **blocklisting process**, which is managed by the Firefox's developers and is updated every night. It contains the CRLs of revoked intermediate CAs and also the websites known as malicious.
- *Chrome* uses ***CRLsets*** which is managing these CRLs pushed every time there is an update. On the start, Chrome checks for new CRLsets.

In the following picture, there is the command to download the most updated CRL list from Firefox. Then with JQuery some fields (enabled, issuerName, SerialNumber) are extracted.

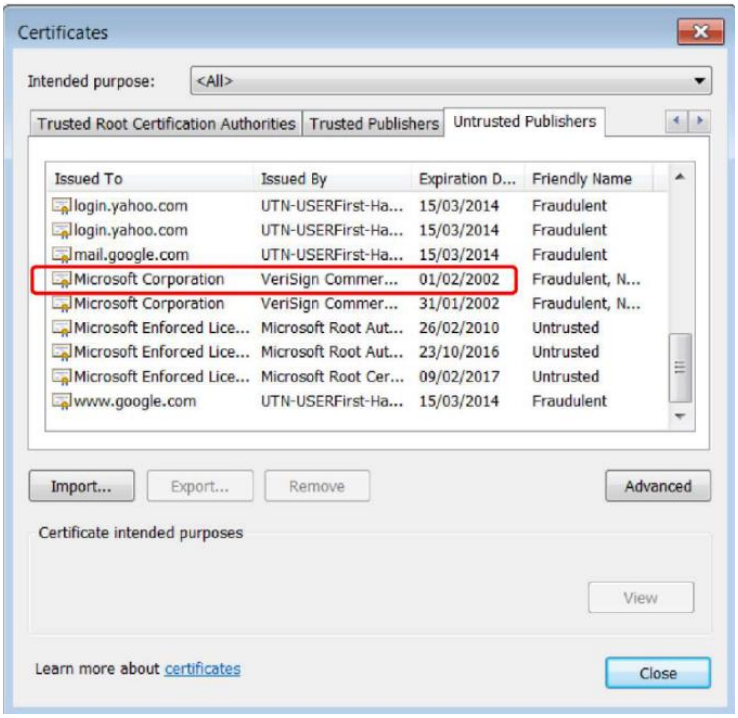
```
$ curl https://firefox.settings.services.mozilla.com/v1/buckets/
     security-state/collections/onecrl/records > crl.json
$ jq '[.data[] | {enabled,issuerName,serialNumber}
      | select(.enabled) | del(.enabled)]' crl.json > crl_short.json
```

```
https://github.com/agl/crlset-tools
```

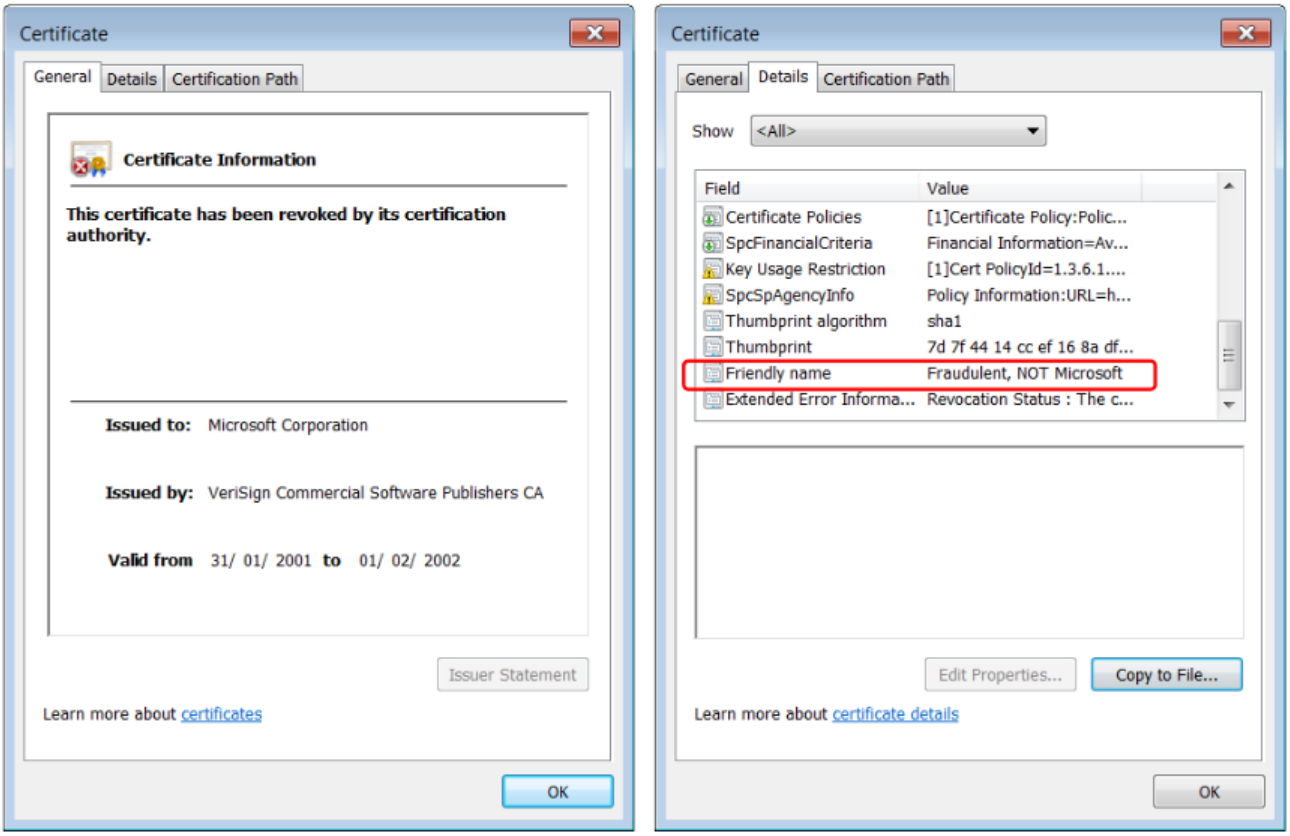
The last link is for the Chrome version, using github will download the crlset-tools, which is a set of tools to look and manage the CRLs sets for Chrome, Edge and Opera.

MS certificate problems

In the last version of Internet Explorer, certificates can be checked in 'Content' tab by following the path: *Internet Explorer* → *Internet options* → *Content* → *Certificates*. On Certificates menu there will be different tabs (as in the picture on the right) and the *untrusted publishers* section. This section which is updated every time there is a revocation for a big CA. The column *Friendly Name* is an additional comment which could say:



- **Fraudulent**: it means that someone went to the certification authority and persuaded them to be a representative of a specific domain/company (Yahoo, Google, Microsoft) and asking for a certificate and specific key that belongs to that domain. This means that the procedure followed by the certification authority is not very strong. This can happen since there are both technical and procedural parts. In other words, Fraudulent means that someone performed a fraud, so an attacker cheated while requesting a certificate e.g. presenting a document with a fake paper signature of a person in a company.
- **Untrusted**: looking the certificates in the picture it is possible to see that *Microsoft Root Certificate Authority* issued a certificate for *Microsoft Enforced Licensing Registration Authority CA* (used to sign licence keys of Windows). It means that there is a certificate create from Microsoft to Microsoft. Since it is not possible, it is marked as untrusted.



Looking the upper pictures in this case the attacker went to VeriSign providing a fake document asking for a certificate of Microsoft. VeriSign is the biggest certification authority in the world (Microsoft own 50% of VeriSign). From 2001 to 2002 the owner of the certificate was able to go around creating fake updates or software signed by Microsoft, since everything is signed by Microsoft is automatically trusted by Windows. Opening the *Certification Path* tab there will be the *Certificate Status* saying: "*This certificate was revoked by its certification authority*" and it should not be accepted by IE. This is specific for the client and there is nothing common to all the clients installed on an operating system. Each TLS client has its own way to manage the revocation.

These problems have been published in several places: by Microsoft itself and by the centre for incidents and vulnerabilities of US government.

- **https://us-cert.cisa.gov/ncas/current-activity/2012/06/04/Unauthorized-Microsoft-Digital-Certificates**
- **https://docs.microsoft.com/en-us/security-updates/SecurityAdvisories/2012/2718704?redirectedfrom=MSDN**

TLS: OCSP stapling - concept

Now let's consider how browsers and servers are managing the OCSP problems which can create problems not only in the verification but also about the privacy. To Stapling means keeping things together.

- *CRL and OCSP automatic download* is disabled because it takes too much time. That means that the browser is exposed to attacks using certificates that have been revoked.
- *Pushed CRL* contains only some revoked certificates because otherwise the CRL would need to become huge.
- *Browser behaviour is greatly variable* in this area. Depending on the browser you can get one feature or the other. There is no standard way to manage things.

Rather than relying on the client, we try to rely on the server for providing the revocation information. The concept is the **OCSP stapling**: when the server is sending its certificate, it will also send the revocation information in the form of a *recent OCSP answer* (it is like saying '*this is my certificate and this is the proof that my certificate is good*'). The client does not have to get the OCSP answer as a separate connection but is provided as part of the initial handshake.

Implementation

OCSP Stapling is a **TLS extension**: it is not part of the basic handshake so it needs to be supported both from the server and the client. Moreover, it is declared in the TLS handshake, when the server provides the server hello, it will alert the client that it will be using the extension.

Version 1 of OCSP Stapling is in RFC-6066 (extension "*status_request*"), while Version 2 is in RFC-6961 (extension "*status_request_v2*") with value *CertificateStatusRequest*.

The TLS server pre-fetches the OCSP answer from one OCSP server and provides the answer to the client in the handshake, as part of the server's certificate message. The server's certificate message does not contain only the certificate chain but also the OCSP answers for each certificate in the chain. That message will become bigger because for each certificate there must be one OCSP answer that demonstrates the non-revocation (the validity) of that certificate. This means that the OCSP responses are "*stapled*" together with the certificates.

The big benefit is that the **client privacy in respected**, because it is not the client asking to the OCSP server the information, but it is the server itself. **The OCSP server does not know who is the client requesting access**. Moreover, there is also the advantage that the client does not have to open a separate connection with the OCSP server but that information is downloaded as part of the normal TLS connection. There is both an advantage in speed and privacy.

The negative point is the freshness of the OCSP responses. Since the OCSP answer has been pre-generated, maybe it could be not so recent (server stores the answer for some time, maybe some minutes). **This freshness creates a small window of opportunity for fast attacks**. An attacker could connect to the server and get the certificate and the OCSP answer which says that the certificate is good. Then the attacker can attack the server to get the private key and create a fake server. When people connect to it, the attacker can send the correct certificate and the correct OCSP answer until the browser accepts it (does the browser accepts an OCSP answer generated 30 minutes ago?).

OCSP stapling is normally performed automatically by the server, but the client may request it. When client connects to the server normally it doesn't know if the server will send the OCSP response. For this reason, the client **may** send the CSR (*Certificate Status Request*) to the server as part of the ClientHello and this is explicitly requesting the transfer of OCSP responses in the TLS handshake.

This process is **not compulsory**: even if the client requests the transfer of OCSP responses, maybe the server is not supporting stapling. For this reason, **it may return OCSP responses for its certificate chain** in a new message which is ***CertificateStatus***.

The problems are:

- Servers **may** ignore the status request; they are not forced to provide this answer.
- Clients **may** decide to continue anyway the handshake even if OCSP responses are not provided.

The technology is there but there is a lot of uncertainty if it is being used or not. There should be **constant monitoring**: even if TLS connection is protected we should look what happens in the handshake and check if OCSP is provided, requested, not requested and so on, because that provides information on what is the real status of security and not the theoretical one.

Since there is this uncertainty, the solution is **forcing the stapling** through a new solution named ***OCSP Must Staple***.

OCSP Must Staple

When the server is requesting the certificate to the certification authority the server **may** declare to the certification authority that will always provide OCSP answers to the clients (even if not requested) and that **information can be added to the certificate**. Those kinds of servers present to the client a **certificate that include a certificate extension** named ***TLSFeatures*** defined in RFC-7633 which informs the client that whenever it connects to that server it **must** receive a valid OCSP response as part of the TLS handshake otherwise the client **should** reject the server certificate. It is like a **promise** that server will always send not only the certificate but also the OCSP responses, so if an answer is received and it does not contain OCSP responses then it is a fake server.

The benefit is that the client does not need to query the OCSP responder and there is an attack resistance because it prevents blocking OCSP responses (for a specific client) or DoS attack against OCSP responder.

Actors and duties

- CA **must include the extension into server certificates** if requested by the server's owner.
- The OCSP Responder **must** be available 24 hours every day and return valid OCSP responses, so it means that must be a robust infrastructure with several network accesses, backup servers, resistant to DoS attacks. Otherwise, server will not be able to open connections (since it must provides OCSP responses).
- The **TLS client must send the CSR extension** in the TLS in the ***ClientHello***, understand the OCSP Must-Staple extension (if present in the server's certificate) and reject the server certificate without an OCSP stapled response if there was the promise.
- The TLS server, to support the stapling, must pre-fetch and cache (for some time) the OCSP responses. It has also to provide an OCSP response as part of the TLS handshake and it has to be able to handle errors in communication with OCSP responders, since it may be temporary unavailable.
- The TLS server administrators should configure their servers to use OCSP Stapling and request a server certificate with *OCSP Must Staple* extension.

An open issue (and potential pitfall) is the **duration of OCSP stapled response** (e.g., 7 days for Cloudflare). Seven days is a huge time since if a server has been attacked there is a window of exposure of 7 days potentially.

SSH (Secure Shell) protocol

SSH is the main competitor to TLS in protocols for secure channels but compared to TLS is not that much used. SSH stands for **secure shell** because in the beginning it was a protocol used to create remote shell (interactive connection) in a secure way to another server.

A bit of history

SSH-1 was created in July 1995 by the security researcher Tatu Ylönen in response to a big hacking incident that caused the leak of thousands of usernames and passwords because a sniffer was placed on the backbone of the Helsinki University Technology and since communications were in clear, the sniffer was able to sniff anything. For this reason, he developed a way to establish a network connection without exchanging in clear usernames and passwords. It was developed quickly and gained a lot of success, since SSL was not exportable outside of the US. On the contrary, since SSH was developed in Europe that could be used everywhere in the world, this let SSH to be much better as security then SSL due to US regulation.

In December 1995 SSH Communication Security was created and two version of SSH were available, one commercial and one open source.

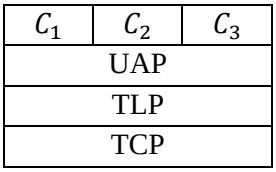
In 1999 a fork appeared: OpenSSH. This was created after Ylönen made SSH commercial only. With this fork they reimplemented the same protocol from scratch in OpenBSD 2.6. This is the major open-source version of SSH.

In 2006 the IETF created the standard **SSH-2** which is *incompatible* with SSH-1 and is nowadays widely used everywhere.

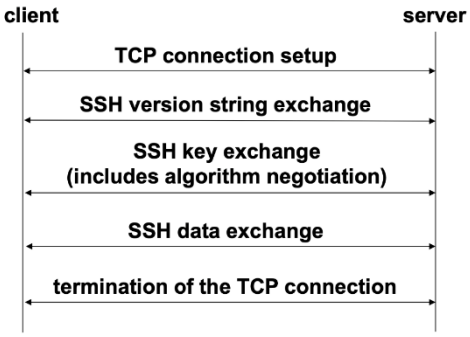
SSH architecture

SSH is using a 3-layer architecture:

- **Transport Layer Protocol**: it is not the transport layer like TCP, since SSH is built on top of TCP but they call it in the same way. It contains the initial connection between client and server, performs the server authentication, provides **confidentiality** and **integrity** with **perfect forward secrecy** and performs also **key exchange** both initially and periodically (because it is recommended to change keys after 1 GB of data is transmitted or after 1 hour as written in RFC-4253).
- **User Authentication Protocol:** it is put on top of the transport layer protocol. This protocol authenticates the client to the server. It is temporary run: even if username and password is used, that is protected (since there is already a secure channel) and it is temporary since it is used only for client authentication, then it disappears.
- **Connection Protocol:** it supports **multiple connections**, named **SSH channels**, over **a single secure basic channel** implemented with the Transport Layer Protocol. It means that there is only one TCP channel. Note that the multiple connections are always between the same couple of client and server since the single TCP connection is established between two specific nodes.



This behaviour is not normal in other security protocols. The Connection Protocol makes SSH capable of creating **tunnels** because it is possible to insert several different transmissions on top of one secure connection.



Transport Layer Protocol

It is the basic layer for providing protection of the transmission. The client is setting up a TCP connection to the server, after that there's the *version string exchange* since client and server must use the same version of SSH. Then there is the *key exchange* together with the negotiation of the algorithms that will be used for protection. After this it is possible to start *data exchange* that can be data required for user authentication protocol or the data required for the connection protocol. Finally, when the channel is closed, it is terminated also the TCP connection.

SSH TLP: Connection and version exchange

For **TCP connection setup** the server is listening on port **22** by default and the client is the one initiating the connection.

For **SSH version string exchange:** both sides must send a version string of the following form:

$$SSH - protocolversion - softwareversion\ SP(space)\ comments\ CR\ LF$$

Notice that **this string is in clear**. This is also used to indicate the capabilities of an implementation and it also triggers compatibility extensions. All packets that follow the version string exchange use the **Binary Packet Protocol** (the string is in US ASCII characters and are immediately read, but after that string the rest is in binary format) so interpretation is needed.

SSH Binary Packet Protocol

The binary packet protocol is represented in the picture: there are 4 bytes that declare the *packet length*; 1 byte declaring the *padding length* in case there is the use of a block cipher that requires padding; the *payload* may be compressed; the *random padding* that is random between 4-255 bytes. All the previous parts are encrypted when encryption and keys have been negotiated. The first packet of course has the null algorithm with the null key but after the key exchange and algorithm negotiation it is encrypted. Then there's a *MAC* of 4 bytes to protect the whole exchange. This is an **encrypt-then-authenticate** approach.

- *Packet length* doesn't include the MAC and does not include the packet length field itself.
- *Payload* (might be compressed) size is equal to $size = packet\_length - padding\_length - 1$ and the $max(uncompressed)$ size is $32768\ byte = 32\ KB$.
- *Random padding* is between 4 and 255 bytes, and it's created in such a way that the total packet length excluding the MAC must be a multiple of the $\max(8, cipher\_block\_size)$ (between 8 and the *cipher_block_size*) even if a stream cipher is used. This is peculiar: padding is added just to align the size of the packet and not only if a block algorithm is used.
- The *MAC* is computed over the **cleartext packet** and an **implicit sequence number**, as in TLS, since it is possible to easily number the packets when receiving them. The packet requires decryption before checking integrity so in some sense it's not actually *encrypt-then-authenticate* but *encrypt-and-authenticate* in the sense that when a packet is received it is not possible to immediately perform authentication, but there is first the need to decrypt and then compute and check if the MAC is correct or not. This makes a DoS attack possible since it is possible to modify just a bit to waste a lot of time in decrypting and then discovering that the packet is bad. TLS 1.3 in this aspect is way better since it uses authenticated encryption, so manipulation is immediately detected without performing decryption.

SSH TLP: Key exchange

The **algorithm negotiation** proceeds as follow:

- $SSH\_MSG\_HEXINIT$: this is the first message *SSH message key exchange init*
- $cookie$ (16 $random\ bytes$)
- $hex\_algorithms$: it is the list of key exchange algorithms which are the once that can be used
- $server\_host\_key\_algorithms$: the one for the server
- $encryption\_algorithms\_client\_to\_server, encryption\_algorithms\_server\_to\_client$: decision for encryption algorithms from both client to server and server to client. In TLS the same algorithm is used with different keys, to avoid someone taking data from one direction and putting it in the other direction. In this case there are different keys and maybe also different algorithms for the two directions.
- $mac\_algorithms\_client\_to\_server, mac\_algorithms\_server\_to\_client$: same as before.
- $compression\_algorithms\_client\_to\_server, compression\_algorithms\_server\_to\_client$: same as before. As TLS the compression is discouraged.
- $languages\_client\_to\_server, languages\_server\_to\_client$: since SSH is a protocol for exchanging text data (at least originally) then it supports also the declaration of languages.
- $first\_hex\_packet\_follows$ ($flag$) it is an attempt to guess agreed $hex\_algorithm$

SSH TLP: Algorithm specification

- $hex\_algorithms$ (Key exchange algorithm) is a list containing also the **hash** to be used. For example, *diffie-hellman-group1-sha1* or *ecdh-sha2-OID_of_curve* (OID = Object Identifier of the selected EC).
- $server\_host\_key\_algorithms$ is the declaration of which key the server is using for authentication because the server is authenticated as part of the Transport Layer Protocol. For example, it can be *ssh-rsa.*
- $encryption\_algorithms$ for the two directions, could be for example *aes-128-cbc, aes-256-ctr, aead_aes_128_gcm.* Even if the MAC is explicit, they could use authenticated encryption in the latest version.
- $mac\_algorithms$ for the two directions, can be for example *hmac-sha1, hmac-sha2-256, aead_aes_128_gcm.* If the MAC is part of the authenticated encryption, it is possible to declare the same algorithm for the MAC but 2 declaration are needed in any case.
- $compression\_algorithms$ can be for example *none, zlib.*

Complete list of all possible values is maintained by IANA: https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml.

SSH DH Key agreement (and server authN)

SSH always uses DH for key agreement but then, as part of key agreement, there's also server authN to consider. The procedure is the following:

- Client generates a random number $\boldsymbol{x}$, computes $\boldsymbol{e = g^x \bmod p}$
- Client sends the generated value $\boldsymbol{e}$ to the server
- Server generates $\boldsymbol{y}$, computes $\boldsymbol{f = g^y \bmod p}$
- Server generates the key by computing $\boldsymbol{K = e^y \bmod p = g^{xy} \bmod p}$ and the exchange hash $\boldsymbol{H = HASH(}$ $\boldsymbol{c\_version\_string \mid s\_version\_string \mid c\_kex\_init\_msg \mid s\_kex\_init\_msg \mid s\_host\_key\_Ks \mid e \mid f \mid K}$) and this is actually a MAC because it's computed over data and the key.
- Server generates a signature $\boldsymbol{sigH}$ over this MAC using the private part of the keys which may involve computation of another hash, because if the server has got an RSA key, then in order to create a digital signature it must perform another hash.
- Servers sends $\boldsymbol{Ks|f|sigH}$ to the client where Ks is the public key of the server, f the computed value and the signature. This provides server authentication because the signature is performed with the private part of the same key. So, the server is sending the public key not inside a certificate but is sent here in clear.
- The Client has to verify that this key is really the public host key of the server, and this is one of the biggest problems of SSH, because SSH doesn't use certificates but uses the model of **direct trust** to verify the public key of the server.
- Client computes $\boldsymbol{K = f^x \bmod p = g^{xy} \bmod p}$ and **H = HASH(...)**
- Client verifies that **sigH** is the signature created by server over **H**

Note that, in the end, the handshake is protected because if this key is trusted, which is a big IF, then when the client recomputes the hash can verify it against the signature made by the server. In some sense the server is signing the whole handshake because the whole information is in the value **H.** This is very different from TLS because in TLS the *finished* provided one MAC for the client and one MAC for the server, on the contrary here since client is not involved, it is the server that creates a digital signature over the exchange and then the client verifies that nothing bad occurred. In the finished created by the server in TLS there are only the messages sent by the server, but on the contrary here in this hash there are both the messages sent by the server and the ones sent by the client, so this signature is protecting the whole handshake.

Since this value **H** has been verified as correct and uniquely identifies this connection since it contains all the *parameters*, *the public key of the server* and the *two values generated randomly* then no other connection can have the same hash, and then **H** becomes the **session-id**. So, the session-id is the hash of the handshake, not a "classic" number, and it's unique.

In conclusion, the key sent from the server to the client is the same public key used in the computation above. In SSH server authentication is <u>always</u> an asymmetric challenge-response: there is an explicit signature but no explicit challenge, but it is possible to consider the whole handshake as the challenge because the server is signing it. The key which inserted in the computation and the key which is sent in clear is the *public key* of the server, while the signature is computed using the private key of the server. That is the problem, because we are sending the public key in clear, so we need to trust that key since there is not a certificate to prove the validity.

SSH Key derivation

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: $\boldsymbol{HASH(K \,||\, H \,||\, "A" \,||\, session\_id)}$, where K is the key that has been negotiated and A is simply a letter.
- From server to client: $\boldsymbol{HASH(K \,||\, H \,||\, "B" \,||\, session\_id)}$.

The encryption key is generated as follow:

- From client to server: $\boldsymbol{HASH(K \,||\, H \,||\, "C" \,||\, session\_id)}$.
- From server to client: $\boldsymbol{HASH(K \,||\, H \,||\, "D" \,||\, session\_id)}$.

The integrity key:

- From client to server: $\boldsymbol{HASH(K \,||\, H \,||\, "E" \,||\, session\_id)}$.
- From server to client: $\boldsymbol{HASH(K \,||\, H \,||\, "F" \,||\, session\_id)}$.

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).

SSH Encryption

Encryption algorithm is negotiated during the key exchange and can be different for each direction of the connection. The basic set specified in RFC is:

- (required) *3des-cbc* (with three keys, i.e. 168 bit key)
- (recommended) *blowfish-cbc, twofish128-cbc, aes128-cbc*
- (optional) *twofish256-cbc, twofish192-cbc, aes256-cbc, aes192-cbc, serpent256-cbc, serpent192-cbc, serpent128- cbc, arcfour, idea-cbc, cast128-cbc*

The Key and the IV are established during the key exchange as we saw earlier. All packets sent in one direction is a **single data stream** and the IV is passed from the end of one packet to the beginning of the next one. So it is needed only one IV and then the last value computed for one packet is the one used for the next packet since packets can't be lost and they come in the right sequence since we're on TCP.

SSH MAC

MAC algorithm and key are negotiated during the key exchange and can be different in each direction. Supported algorithms are:

- *hmac-sha1* (required) [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]

It is expected an update soon supporting SHA-2. The MAC is computed upon the key, the implicit sequence number concatenated with the clear text packet, as follows:

$$\boldsymbol{MAC = mac(key, seq\_number \,|\, cleartext\_packet)}$$

Sequence number is implicit (not sent with the packet) and represented on 4 bytes, initially 0 and incremented after each packet and is never reset even if keys and algorithms are renegotiated. It means that if there's an overflow the SSH channel needs to be closed and a new one has to be started. At most 2^{32} packets can be exchanged.

SSH Peer authentication – Server

It is an **asymmetric challenge-response** based on an explicit server signature of the key exchange hash H. There is no PKI so the client locally stores the public keys of the servers and this is very dangerous, typically each user stores them in its home directory: $\sim/.ssh/known\_hosts$ where there are couples server name and public key. That file should be manually initialized, so key should be received for example from the system manager, because when there is a connection to a server the software compares the key received by the server with the key stored in that file. The problem is that if the key is absent then it's offered at the first connection and will be automatically inserted in the file. In that case the key is offered and a fingerprint is shown, so even if there is not the actual public key in the file there's the hope that at least the fingerprint has been received so it is possible to compare it to the one shown and decide if to save the key or not. That is the most dangerous part of SSH, since this file can be changed by anyone who has access to it. **A good practice** is to protect **known_hosts file** for authentication and integrity and periodically perform audit/review of **known_hosts** to quickly detect added/deleted hosts or changed keys.

SSH Peer authentication – Client

Client authentication is part of the additional protocol *user authentication protocol*. It can be performed in two different ways:

- **Using username and password**, this is exchanged only after the protected channel is created. It's protected from sniffing but still opened to other attacks (e.g. on-line password enumeration)
- Optionally it is possible to use **asymmetric challenge-response,** but the problem again is that there are no certificates, so the public key of the authorized client is stored by the server in a file $\sim/local\_user/.ssh/authorized\_keys$**,** which means that each user can define which keys can be used to authenticate itself. Again, a good practice is to protect *authorized_keys* for authentication and integrity and periodic audit/review the authorized_keys file to quickly detect added/deleted users or changed keys.

User authentication is compulsory at minimum with username and password, unlike TLS.

SSH Port forwarding/tunnelling

It is possible to have multiple channels encapsulated on top of Transport Layer Protocol that opens the way to **tunnelling** using SSH which is also named **port forwarding**. This is a way to forward TCP traffic through SSH. TCP is insecure and over TCP there is the TLP and then again, another TCP channel (*tunnel*). For example, if it is wanted the protection of an insecure channel like POP3, SMTP, HTTP, which are insecure, that means that from the point of view of the application the client-server application will run its normal authentication over the encrypted tunnel.

There are two types of port forwarding:

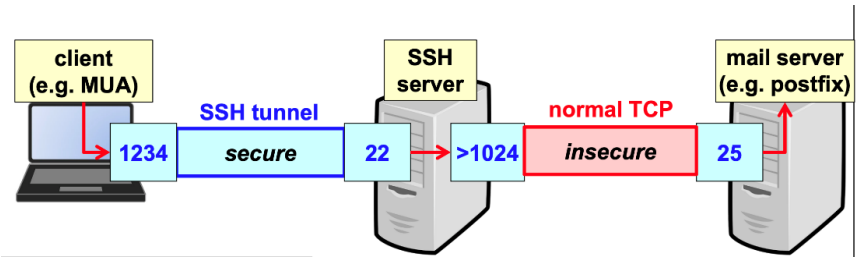
- **Local forwarding:** is used to create an *outgoing tunnel* to connect to an external service in a secure way even if the protocol being used is insecure.
- **Remote forwarding**: to create an *incoming tunnel* to offer a service to others in a secure way.

Both use the Connection Protocol to encapsulate a TCP channel inside an SSH one.

SSH Local port forwarding

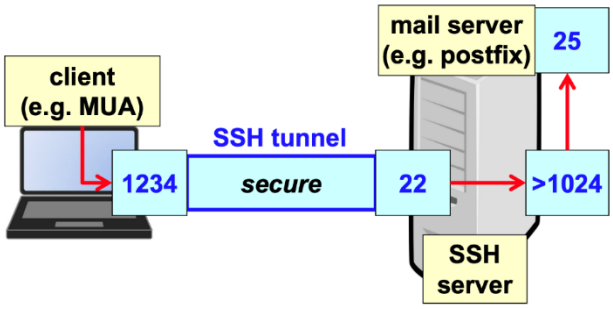
This is forwarding traffic from a local port to a remote port. For example, using MUA (a mail client) that wants to send a mail, normally using TCP on port 25 which is not secure at all so there's the danger that someone is listening in the local network. The mail server is responding to port 25.

If there is the availability of an SSH server outside, it is possible to create a tunnel between one local port, for example port 1234, up to port 22. So, there is the creation of a tunnel and then the MUA is configured to send mails to port 1234 on local host. This port, however, is connected to the tunnel so everything that goes to that port will travel the secure tunnel and then the SSH server will open the normal mail connection SMTP to port 25 of the destination. In this way **there is the protection of the part in the local network**.



The command in the linux shell is: $\boldsymbol{ssh\ -L\ 1234{:}\ mail\_server{:}\ 25\ user@ssh\_server}$ which means that the local port 1234 needs to connect to the *mail_server* port 25 through SSH and in the end the user is specified, then the user authentication is done. This can be useful if the client is behind a firewall that blocks insecure connections but not the secure ones.

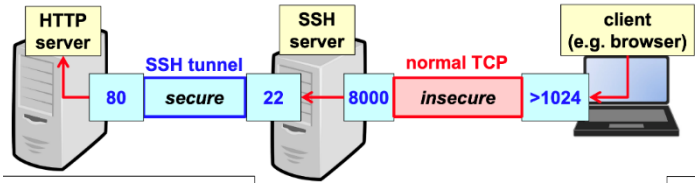
If there is not the trust in the last part of the connection (the insecure one), may be the owner of the mail server could have activated an SSH server. So the alternative is to connect with SSH directly to the mail server and then the mail server redirects the traffic internally (the insecure part exists but it is inside the same server).



The linux command in this case is: $\boldsymbol{ssh\ -L\ 1234{:}\ localhost{:}\ 25\ user@ssh\_server}$ which means that the local port 1234 should connect to port 25 of the remote localhost, where localhost is the SSH server which is also the mail server. So, there is no direct connection to port 25 but connection to port 22 and then internally it is forwarded to port 25.

SSH Remote port forwarding

This solution is used by servers when they want to be available to the outside people that may be passing through a firewall or maybe behind a NAT. It's used to forward traffic from a remote port at the SSH server to a local port of the SSH client (which is usually a server in this case). So,



an external user wants to connect to the local HTTP server (which is behind a NAT). The Linux command is: $\boldsymbol{ssh\ -R\ 8000{:}\ 127.0.0.1{:}\ 80\ user@ssh\_server}$ where port 8000 on the SSH server when receiving a connection will move that connection through SSH to the local port 80. So there is the creation of a listening point on the SSH server and anybody from outside in order to connect to the internal server needs to connect to port 8000 of this SSH server. Note that the client from the external does not need SSH and is not authenticated (the insecure part is normal HTTP). It can be run by any host with a browser and then a tunnel is created for the part in which there is the need of protection.

SSH causes of insecurity

- One problem is the direct trust in public keys since X.509 certs are not used, but some commercial versions use certificates, and openSSH has SSH certificates, but most implementations use direct trust.
- Users ignore warnings and blindly accept new server public key (when not listed in the file) when opening connections with new servers, which leads to MITM attacks.
- Weak server or client platform security: malwares can then change or read the known hosts or the authorised keys.
- Furthermore, any connection to a local forwarded port will be tunnelled even if coming from another node, this means that local forwarding works not only for internal but also for someone else trying to use our connection. To avoid this there should be the specification of the *local bind address* and so specify that port forwarding is wanted only for localhost: $\boldsymbol{ssh - L\ 127.0.0.1:1234:mail\_server:25\ user@ssh\_server}$ in this way only processes coming from localhost can use the tunnel.

Some attacks against SSH

BothanSpy

BothanSpy is probably a tool created by CIA, as exposed by Wikileaks, and targets *Xshell* (a Windows SSH client used in USA and South Korea) injecting a malicious DLL into a *Xshell process* to **steal username and password** for all those passwords authenticated connections and **username**, **private key file name**, **and passphrase** for public-key authenticated connections. This also exposes the risk of using an application with shared libraries.

It is designed for use with the *ShellTerm attack framework* that creates a covert channel with Command&Control server and DLL injection capabilities with direct communication with C&C, no data written to disk, hence difficult to detect for anti-malware because there's nothing on the disk that can be used for detection. It can also be used off-line: writes collected data to disk, encrypted with AES, for later collection when online.

Gyrfalcon

Another CIA tool that targets *openSSH in enterprise Linux* (RedHat, Centos, etc...) which pre-loads a malicious DLL to **intercept plaintext traffic** (i.e. before encryption and after decryption) collecting username and password but also actual data. It uses encrypted configuration file and encrypts file for captured data so it's difficult to understand what is collected. It requires root access, but it is not integrated with C&C and freely read/write files on disk, probably relies on the rare adoption of anti-malware in Linux. Furthermore, it is surprisingly not stealthy and unsophisticated.

Brute force attack against SSH

There's a false sense of security when using a secure channel leading to insecure authentication (i.e. reusable passwords). The typical brute-force attack consists in trying all passwords from a dictionary of well-known ones.

Example of an attack:

- Sep-2016, servers activated to test SSH brute-force attacks
- One IPv4 and one IPv6, in the cloud
- IPv4 server immediately attacked and conquered in 12 minutes as the password for root was "password" on purpose
- In a few minutes the machine was used for a DDoS (turned into zombie machine)
- On the contrary, after 1 week the IPv6 server was not even attacked because currently the most of the scanning of the internet happens in IPv4

Protection from brute force attack

For linux:

- Account lockout after N authentication failures using programs like **pam_tally2** or **pam_faillock.**
- Use **tcpwrappers** to permit/deny access to specific hosts or networks.
- Implement SSH rate control with **IPtables** (e.g. 5 connections per minute) slowing down a brute force attack that is trying passwords.
- **This is compulsory:** deny direct access as root via SSH. People have to connect with their own username and then can use sudo to become root.
- It may be useful to change the default port number from 22 to something else but people might still find out the new port (i.e., port scanning).
- Use **Fail2ban** to blacklist attackers after reading that there have been N failures in the log.
- Use **2FA** (e.g. by adopting Google Authenticator).
- The best possible option is to **disable password-based client authentication** configuring the server to accept only challenge-response authentication

Main applications of SSH

The main applications of SSH are:

- Remote interactive access (with text-based interface)
- Execution of commands on remote systems (without accessing the shell)
- As we have seen, creation of tunnels for various applications using the security provided by SSH

Nowadays it is directly available in Linux, Mac, and Windows, both the SSH client and the SSH server.

The X.509 standard and PKI

Public-key certificate (PKC)

PKC is a data structure to **securely** bind a **public key** to some attributes that identify the owner of the corresponding **private key**.

Typically, the securely bind is obtained with the **signature** by an authority, but it is possible to have other methods, for example nowadays there are **blockchains** that are a distributed ledger, so something which is stored in several places and if the majority agrees than the public key is trusted, or there could be **direct trust** (e.g. SSH) and personal signature.

The **attributes** are related to the owner of the corresponding private key, but the owner can be identified in several ways, so we use those attributes that are **meaningful** for the transaction being protected, for example if we are buying a house or a car, probably the attributes that we need are *fiscal code* or the buyer, if we are accessing to a web site the fiscal code is no more meaningful but we will use IP address or email address, often meaningful attributes are not known a priori because there is a great variability.

One of the most important things in PKC is that normally it is required if you want to achieve **non repudiation** of a digital signature, that means something which is **legally valid** and cannot be denied when you go in court, so non repudiation is needed for any digital signature used for legal matters (e.g. buying or selling goods, statements, etc.).

PKC is the **public complement** of the corresponding personal private key.

Asymmetric key-pair generation (SK + PK)

Key generation requires complex algorithms and often RNG (Random Number Generation) and after generation, private key needs to be protected:

- **When it is stored**: because if someone copy our private key than he will be able to identify ourself;
- **When it is used**: because when we use the private key to perform some operations like digital signature or decryption, we need to give the private key to some computational engine such as the CPU, and if the CPU is infected with some malware than the private key will be compromised.

The generation and the use of a key pair can be performed by a **software** application, for example all browsers have the capability to generate a key-pair, but computers may be infected with malwares and additionally it is possible to have weak keys (if the algorithm for generation is not properly implemented). For this reason, we don't rely directly on the browser, but we use some **dedicated hardware** (e.g., RSA smart-card), but in this case there are problems in the updating of algorithms and mechanism, and it is difficult or impossible to release a vulnerability patch since in general hardware update is **difficult**.

A third solution for the key-pair generation is to generate the keys using software (well implemented) and then to **inject the private key inside the hardware secure device**, this is often the case when you want to give keys to employees, because if the private key is under the control **only** of the employees than there is a problem: if the employee is using the key to encrypt some important data and after he leaves the company, you will not be able to read the data. For those cases company will give the keys, but a copy of those keys will be stored inside the company (e.g., if employee loses the key). This third solution is acceptable if the keys is restricted to only perform encryption, not digital signature, because there is a problem of **non-repudiation** since both the company and the employee know the private key. The most common solution is to have **two key-pair**, one for **digital signature** which is associated to non-repudiation (in this case the private key is yours and only yours) and the other one for **encryption** which may be subject to key recovery to help get back the data in case you lose control of the private key.

Certification Architecture

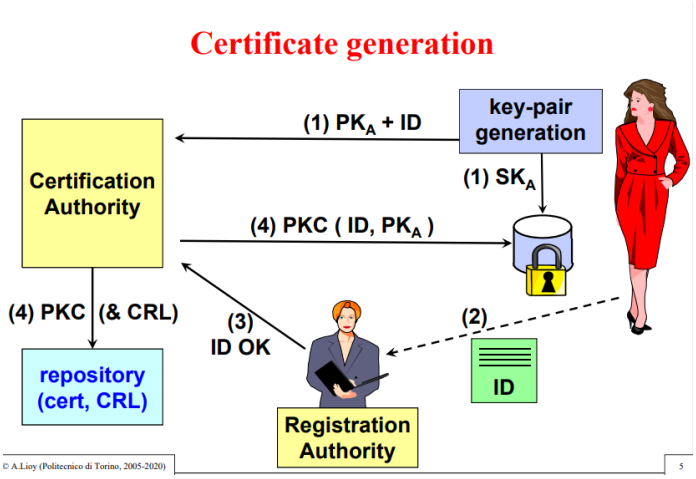
It is composed by:

- **Certification Authority (CA):** it is the actor of the system which is in charge to **generate** and **revoke** Public Key Certificates, the CA also **publishes** PKCs and the information about their status (e.g. CRL), the CA can be the only actor in the Certification Architecture but often it is supported by one or more RAs.
- **Registration Authority (RA):** actor that **verifies** claimed identity and attributes, so it authorizes PKC **issuing / revocation.**
- **Validation Authority (VA):** it could be external, and it is the actor who provides services to verify the validity status of a PKC, this could be done by the CA itself.
- **Revocation Authority (unofficial term, role can be assigned to RA or CA):** the revocation can be more urgent than the issuing. So, consider the case in which a private key has been stolen at 3:00 AM and may be that key is useful to access a bank account, it needs to be blocked immediately. In the night the CA is closed, so it is not possible to perform that operation (in this case the certificate revocation), so an office that is always open and ready to perform that operation is needed.

The CA must be always present, the other roles are present for supporting.

Certification generation

Here one of the possible solutions:



- There is a user that is generating its own key-pair (in various methods: browser, smart-card, etc.), once the generation has been completed the SK is **locally stored in a protected format** and simultaneously the PK and the attributes associated to the PK are sent to the CA in the form of **CSR** (Certification Request).
- The CA does not create immediately a PKC because it must **check the validity** of the attributes and of the requestor, which is done by the RA. It will check the validity of the identifier and of the identity of the requestor, depending on the **policy** which is applied by the RA. The policy is what kind of proof the requestor has to give (name, surname, ID card, fingerprint).
- The response is sent from the RA to the CA.
- If the response is good the CA will create the **Public Key Certificate** that will be returned to the requestor that will store it in the secure folder with the private key, since it is a public certificate, the CA is also **publishing** the certificate in a public repository which contains the certificates and typically also the CRLs.

There are also other possible schemas, for example:

- The **RA generates the key-pair**, **obtains the PKC** and **distributes** them on a secure device. This is the case of the Politecnico di Torino because when you want a PKC, you bring your smart-card to the RA of the Politecnico and the RA will insert the smart-card and execute the command “Create key-pair”, then the public key is sent to the CA, the certificate is returned to the RA that will insert it in the smart-card giving it back to the user. Typical for large companies, where employees are known.
- The user **first** visits the RA, shows the document and the possession of the attributes and gets a **code** (typically like a OTP), a unique code good only for one attempt to authenticate the request to the CA,

so after that visit it is possible to **personally** create a key-pair and on submit the CSR (Certification Request) will contain also that code. The code is typically computed with the MAC of the identity that has been proved by the RA and a symmetric secret **key,** shared with the CA and RA. $code = MAC(K, ID)$.

X.509 certificates

There are several ways to create certificates but nowadays the most widely adopted standard is X.509, that is a quite old standard that was created not by ITF (Internet itself) but by ITU (International Telecommunication Union), there are several versions:

- **V1:** it was created in 1988 and was not very successful
- **V2:** it was a minor version created in 1993 with some small modifications
- **V3:** with this version X.509 comes with successful modification in 1996 and it contains v2 + **extensions** good to use this certificate in internet environment + **attribute certificate** v1
- **V3:** in 2001 was released another version that was equal to v3 but with attribute certificate v2 (not very important nowadays)

The name X.509 comes from another standard called X.500 created for directory services (white pages), in the beginning of networking Era there were two competing networks, **TCP/IP** and **OSI**, TCP/IP was created by ITF, while OSI network was created by ITU, nowadays OSI is not used but all standard defining OSI are called X.*something* (e.g. X.25, X.400)**.**

Directory services: a directory is a list of items, and each entry of the directory has a list of attributes (for example a directory of the Politecnico di Torino could be the list of employees and students), X.500 (also named white pages) is a standard aims to have one directory all over the world (one entry for each person of the world).

X.509 is the only thing that has saved the failure of OSI thanks to the fact that the v3 has been developed together with ITF in order to make it suitable for internet applications, so X.509 has migrated from OSI to Internet.

X.509 is a solution to **identify** the owner of a **cryptographic key** and if you want to attack it and become an expert of X.509 you need to read ASN.1 (Abstract Syntax Notation 1) that is one way to describe generic object independently of the implementation.

PKC scope

The certificate contains information to **uniquely** associate a cryptographic key to an **entity,** the binding is guaranteed by a **Trusted Third Party (TTP), Third** because you have someone using the private key and someone which is receiving it and needs to know which is good or not, to do that you need the certificate that was created by TTP, that is **external** so a **third entity**.

Typically, the TTP is called **Certification Authority (CA),** which **digitally signs** each certificate.

The security policies of the CA is not only stating the kind of **verifications** that are performed before creating a certificate, but may also include **limitations,** for example by saying that this certificate is valid **only** for this specific association (for example Windows domain will automatically generates a certificate for each user, but this certificate has a strong limitation, because is valid for perform secure operations **inside** this Windows domain, outside it has no value).

CP and CPS

The certification policy is specified in two documents, the **CP (Certificate Policy)** and **CPS (Certification Practices Statement).**

The *Certification Policy* (CP) is a named set of rules that indicates the **applicability** of a PKC to a **particular community** (e.g. Windows Domain) and/or class of application (certificate for Internet Electronic Mail), with common security requirements.

The *Certification Practice Statement* (CPS) is a statement of the practices (procedures) employed by a CA when they issue a PKC.

A CP specifies **minimum requirements** and can be followed by many CAs (e.g., a government could have one CP to be implemented by **all** certification providers, so has a minimum you have to verify either the identity card or the password).

A CPS contains the implementation details, and it is **specific** for a single CA. Things are made respecting the CP but in this or that way.

X.500 directory service

X.509 was created to protect X.500 service, and it was the first application of X.509v1, but there were three main problems encountered:

- There was no guarantee about the quality of the CA (policy was missing in X.509v1).
- There was no specification on how to distribute the certificate, so there was a lack of directory infrastructure and so there was no way to access the certificate, because the certificate should have been made accessible with the same X.500 structure.
- It was also difficult to establish the *certification path* among two arbitrary users, so there was no way to specify a relationship among different CA and since the certificate is part of a **chain** it is a problem.

Remedies for X.509v1

They tried to fix previous problems with the following solutions:

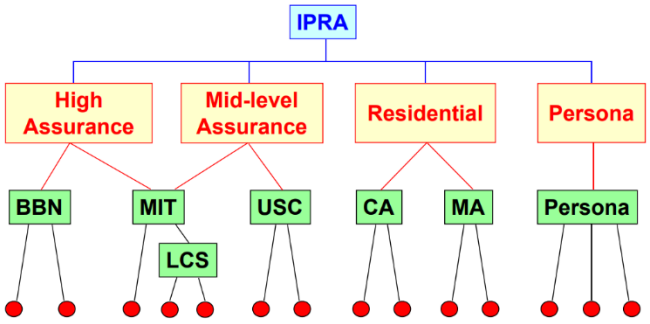
- Force the semantics in the application or in any case in a context external to the certificate: certificates are not improved but an assumption is made on how the certificate is being used. This was the path followed by PEM (*Privacy Enhanced Mail*) that was the first attempt to have secure mail in internet (*RFC-1422*), but it was a failure.
- Make certificate more flexible and expressive (X.509 v3), but the price to pay is the more **complexity**.

Internet PEM (RFC-1422)

RFC-1422 was foreseeing to have one hierarchy covering all the world, so only one **root CA** named **IPRA (Internet Policy Registration Authority),** one level below IPRA there were some special CA that were not certifying users or servers and they were named **Policy Certification Authority (PCA)** because they established the policies to issue the certificate. After that there were the **real CAs,** with a limitation, each CA could only appear with the name just below the previous one, for example they were foreseeing to have one national CA for each country, so in Italy it would have been the [CA n.1] C = IT and all the others Italian CAs should have started with IT, than something else, for example [CA n.2] C=IT O=Politecnico di Torino. C stands for Country and O for Organization.

On the top there is the **only root accepted** (IPRA) in this schema, IPRA will certify Policy Certification Authority:

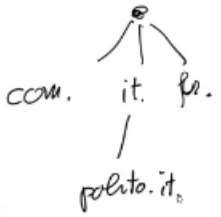
- **High Assurance Policy** that could be, for example, to perform a DNA test to be sure that someone is that specific human being
- **Mid-level Assurance Policy** that is a common kind of police, for example an identity document or driving license.



- **Residential Policy:** in some countries there are no identity documents (e.g. UK, US), so the identification is upon the single subject to tell the truth and if someone cheats there is the jail. Since there are no documents, to demonstrate something in a formal way one possibility is to take a bill (e.g. electricity bill of an house), that shows the address of your house. That is **Residential,** so the real test is "You have shown me a bill with a real address".
- **Persona**: it is a latin word which means "*mask*", so it works as an alias, in the sense that the certificate does not contain personal data (name, surname, address etc.). It's some kind of anonymous certificate, but it is not useless because associated to this certificate there will be a number (e.g. *Anon#37*), so the individual is not known, but when there are multiple documents signed, they are all signed by the same person, so real life individual is not known, but as a minimum it is possible to be sure that there is security and that it is **always interacting with the same person**. In some cases, certificates are useful even **without identifiers.**

Under the PCAs there are the CAs themselves: for example, BBN is a famous company working for the military in the US so most likely they use a High Assurance policy. The MIT may follow two policies (High Assurance or Mid-level Assurance).

The reason of the failure of this schema was the **one single point for all the world (IPRA),** who should manage the IPRA? The one which is in control the root can manage everything under that (can create fake certificates), so it immediately degenerated in a **political issue,** so this demonstrate that unfortunately it is not possible to have just one hierarchy for the whole world.



There is another system in Internet which is a single hierarchy, and which is not a failure: **DNS**. The DNS does not have that kind of problem because even if from a *logical point of view* there is one hierarchy (e.g., "." (dot) on the top and then below top there are "com.", "it.", "fr."… and then under "it." there is "polito.it.", and so on), but from the *physical point of view* there is not just one root DNS server (e.g., "." (dot) is not managed by just one server) because otherwise it would not resist the load of traffic (all the requests in the world). Nowadays, in the world there are at least 12 "dot" logical servers which are implemented with more than 100 physical servers. So, there is not a single point containing all the information. So, there is not one single entity that can create fake names, at least there are 12 entities that can create fake names, and those are distributed all over the world. *Note: See root-servers.org website.*

Therefore, it works. They compare each other and if someone is making false statements the other ones will immediately detect. This works, but this is just for translating names in addresses. For certifications that was not possible: Internet PEM hierarchy was just an attempt interesting to study, to understand why it was a failure and what to avoid in the future. It was useful to also introduce the various concept of policies: High-Assurance policy, Mid-level, Residential and Persona. Because those problems are possible: there could be a CA which is creating anonymous certificates. That is not stupid, it's a real need.

RFC-1422 problems

With this RFC-1422 format there were problems:

- The **hierarchical infrastructure limits the flexibility** (for example, if a company is an international one, should It be in Italy, in France, or where? Because it can be in only one place in that hierarchy, which is not easy).
- The **name subordination** (so, the fact that int the next level you must copy the name of the father and then add something else) **introduces undesired restrictions** to the assignment of X.500 names and not everybody likes that.
- The use of the Policy Certification Authority (PCA) with **few policies** was **not flexible in commercial applications**, where the participation of an operator to take a decision is impractical. Policies were not flexible: they were decided only once and for all.

X.509 version 3

So, in order to bypass the previous problems, they finally decided to modify X.509, in agreement with the IETF, in order to make them usable also for internet applications. At that time, in 1996, it was already clear that OSI would have been a failure. This X.509 version 3 standard was completed in June 1996 and it's a joint work of ISO/ITU and IETF.

X.509 version 3 groups together in a unique document all the modifications required to extend the definition of certificate and CRL. There are **two types of extensions** (*be aware*: the basis is X.509 version 1, while X.509 version 3 is equal to version 1, plus some extensions):

- **Public**: the extension has already been written in the standard, so anybody should know it and should be aware that this extension exists.
- **Private**: X.509 v3 is also making possible to creating our own extensions, unique for a certain group of people. So, if someone thinks that there is something missing, it is impossible to implement that. In the past at Politecnico di Torino they did something like that to store the student IDs inside the certificate, they created an extension specific for the Politecnico, until there was a public extension available. It gives us a lot of flexibility, but it is also bad because apart from the relative group of users no one else will understand that certificate.

Since there are extensions, then it comes out the need to have a certificate profile. A **certificate profile** is a list of the set of extensions that should be used for a specific purpose. For example, RFC-5280 contains the profile for Internet applications: both for Public Key Certificate and Certificate Revocation List (CRL). Given X.509 v3, which gives a lot of possibilities, RFC-5280 tells which set of extensions should be used and with which values if X.509 v3 is wanted for email security in TCP/IP (for example). So, profile is a way to selecting something from a general standard.

This code on the right is an example of ASN1 (*Abstract Syntax Notation 1*), which is the one used for defining certificates, CRLs, and so on.

The symbol "::=" is the one used for *definition*. A certificate is defined as a **sequence** (think about the structs in the C language). Its fields are:

- The *signatureAlgorithm* (field name), which belongs to *AlgorithmIdentifier* that is the type of the data.
- tbsCertificate (which means "to be signed Certificate") that is of the type TBSCertificate.
- SignatureValue which is a bit string.

Base syntax of a X.509 cert

```
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
}
```

So, in general a certificate contains the identifier of an algorithm, the data to be signed and the signature created by the CA.

The TBSCertificate is itself a *sequence* and it contains:

- *version*, which starts from 0 ('0' means v1, '1' means v2, '2' means v3, ...)
- *serialNumber*
- *signature*, that is not the value of the signature, but it is an identifier which tell us that this signature was signed with this algorithm
- *issuer*, who created the certificate
- *validity*: from – to (start of validity and end of validity)

- *subject*: the one controlling the private key
- *subjectPublicKeyInfo*, that is the public key
- *issuerUniqueID* and *subjectUniqueID* should have been unique identifiers in the world that identify the issuer and the subject, but they are normally never used
- **extensions**, which is the characteristic of X.509 v3 (without this field it would be a X.509 v1)

Critical extensions

Every time an extension is written it can be defined as **critical** or **non-critical**. There is always someone that is using the certificate to protect transactions, and someone that is accepting the certificate. The **acceptor** (later we will call it the *Relying Party* (RP), the one that is relying on the certificate for security) has a duty: it should perform verification. If, during the verification process, the acceptor detects that the certificate contains an extension which is marked as "*critical*" and it is unrecognized, then it **must** reject the certificate. Example: I receive the certificate, if there is something that I don't understand and that thing is marked critical, then I throw away that certificate and I close the transaction. On the contrary, if the extension is marked as non-critical and that extension is not understandable, it **may** be ignored (if I am very picky, I can decide to reject it as well). Let's imagine that a certificate of someone else is received and it also contains a photo of the person. That photo should be marked as non-critical since it may be useful if the individual is in front of you, but not if you are in an internet transaction. But it is possible to decide that the photo must be present in any case and so if not, it possible to reject it.

This different processing is entirely the **responsibility of the entity that performs the verification**: the **Relying Party** (*RP*).

Public extensions

Public extensions are those that are defined in the standard and that everybody should understand. They are divided in four classes:

- Additional information about the **key and the certificate policy**
- Additional attributes with the respect to the **certificate subject and certificate issuer**
- **Constraints about the certificate path**, so the sequence of certificates from the root to this place
- **CRL distribution points**: extensions to identify the place where it is possible to get the CRL

We often talk about **EE = End Entity**: it means the one which is at the end of the certification hierarchy and possesses public and private key.

Key and policy information

The first class contains 6 extensions, listed as follow:

CA_1	AKI_1
CA_2	AKI_2
CA_3	AKI_3

- **Authority key identifier (AKI):** any actor may have more than one key pair and if it has more than one then it may need a way to distinguish them: "this keypair, to which certificate is associated?". That is the purpose of the AKI. It **identifies a specific public key used to sign a certificate.** The identification can be performed by means of a key identifier which is typically the digest of the PK computed with a suitable hash algorithm or the pair $issuerName : serialNumber$: giving the name of the CA an example could be $POLITO\_CA: 19754$. That's the way to identify the key that was used to sign (because it's authority KI). The point is that one CA could create certificates using two different keys: one for low and one for high assurance. It is always **non-critical**. There is the tendency to say that if it is non-critical it's better to not waste time, move on and study something else, but based on prof. experience (Politecnico di Torino created the first PKI in Italy, in 1996), if this field is forgot there are some troubles. Because when we get one certificate and we need to go back and create the chain, most of the software applications that perform that action use AKI to build the chain. It means that most software applications contain a table with the various CAs and each of them is identified by its AKI, so when you get a certificate which contains an AKI (which says, "this certificate was signed by this

CA using this key", that key is used to select which CA, and then that certificate will contain another AKI and so you will be able to link to another CA, for example. In this way, starting from the EE (End Entity) you can reconstruct the entire path, until reaching the root. So, if we don't check this extension, even if non-critical, it is possible that nothing will work because software applications need it in practice in order to build the certificate chain.

- **Subject key identifier** (**SKI**): it is again either a digest or pair $issuerName : serialNumber$ to identify a specific public key used in an application (for example when public key is updated). But in this case, it is the **public key of the subject**. This is **non-critical**, and Professor never found any application that complain if this field is absent. This field could be omitted but, since the AKI is already being computing it is possible to compute also the SKI with no harm, just in case it will be used in the future.
- **Key Usage (KU)**: it is very important, since it identifies the application domain for which the public key can be used (e.g., for signature, or for encryption, etc.). It can **be critical** or **non-critical** (it is up to the CA, while creating the certificate). Of course, if it is critical then the certificate can be used only for the scopes for which the corresponding option is defined. In this case, CA is obliging the person who will own that certificate to use it only for some scopes and not for others. If the receiving user tries to use it anyway, without obeying to the restrictions, it will be not valid. There can be some values for the KU extension. In the base set there are values related to the permitted basic cryptographic operations that can be defined. These are:
    - $digitalSignature$ it is valid to appear both in the certificate of the CA and in the certificate of a user because the CA will perform signature of certificates and the user will perform signatures of documents.
    - $nonRepudiation$ it is possible only for the user since it is associated to the creation of some legal obligations which are part of the signed documents. So, that can appear only in the certificate of a user.
    - $keyEncipherment$ means that someone has generated a key and then it is performing an encryption with the public key $enc(pk, k)$. Someone can take the public key from a certificate and use it to encrypt another key which has been used to encrypt the document. This value is valid only for users.
    - $dataEncipherment$ is used when it is wanted to directly encrypt data with the private or public key. Here there is no limitation (only user or only CA or both, no one of these limitations) because this is quite strange, since asymmetric crypto is quite slow so typically it is never directly used to encrypt data, but if someone wants to do so, it is possible.
    - $keyAgreement$ is for supporting things like Diffie-Hellman in which there is an agreement, so there will be also the DH parameters and it is possible to decide if these parameters are only for performing encipherment or only for decipherment (limited to only one of the two).
    - $keyCertSign$ is only for CA: it is a permission to sign a certificate.
    - $CRLSign$ is only for CA: it is a permission to sign a CRL.
- **Private key usage period**: it defines the validity period of the private key. Inside the certificate there is already a validity: e.g., there could be a certificate valid from 2021 January 1st to 2021 December 31st. So, why is this extension needed? Well, this certificate is associated to a certain public key, which is associated to a certain private key. When does this certificate expires? It is possible to go to the CA and get another certificate for the same public key (*for the same keypair*), but maybe valid from 2022 January 1st to 2022 December 31st. So, the expiration of the certificate does not imply the expiration of the key. If this extension is used and it tells that the corresponding private key can only exist until 2021 December 31st, then it makes impossible to create a new certificate the next year, with the same public key. A new keypair must be generated. This extension is always marked as **non-critical** and the usage of this extension is discouraged, because it is creating limitation to the user. Depending on the environment, i.e., in a high-security environment it could be used since the more time the same key is used, if someone is sniffing the more data are exposed and accumulated for performing cryptoanalysis. Additionally, algorithms can change, and the keys should become longer, so for

example it could be possible place there a limit of 3 years since the key was created, because probably after 3 years more secure restrictions could be requested for creating a certificate.

- **Certificate policies**: it is a list of the policies followed when the certificate was issued and the purposes for which it can be used. The indication for the certificate policies can be either directly with the text message, so it is possible to put text inside the certificate, or it is possible to provide a pointer in the form of URI, or a pointer in form of an OID. An *Object Identifier* is a way used in internet, in standards, to uniquely identify, for example, a company and maybe documents inside a company and it is a sequence of numbers where each number has a specific meaning (e.g., the things that are established in internet are something corresponding to numbers that stands for US.DOD…, represented as 1.3.6.1. That is the OID for all the documents related to internet, because 1 in first position is for ISO (International Standards Organization), the 3 is telling that, within ISO, this object is owned by an organization clearly identified, then 6 is for DOD ([United States] Department of Defence) and the final 1 is for the internet documents; so even if internet is no more under the control of the DOD, historically all the documents and the procedures are labelled like this). So, it is possible to ask to get a unique OID for a company and then it is possible to publish the certificate policy assigning a number. It can be **critical or non-critical**. Typically, it is non-critical because understanding means to read the certificate policy. Since the certificate policy is telling "the certificates are given to only maybe 2 people, for example, professors", then, in addition to authentication it can provide also authorization, because if by reading the certificate policy it is discovered that certificates are given only to professors, and then there is an access control which is telling that "only professors can access this kind of documents", then it is possible to perform authorization as well.
- **Policy mappings**: indicates the correspondence (mapping) of policies among different certification domains, so may be there is a CA_1 which is following a certain certificate policy and another CA_2 following a different certificate policy. In this certificate there can be a mapping, which is saying "point number 3 of my certificate policy is equivalent to point number 7 and 8 of your certificate policies". It is present only in the CA certificates. This is typically non-critical, because it just help the analysis of the certificate policies of different CAs and to compare them. That is the reason why it appears only inside the certificate of a CA.

Certificate subject and certificate issuer attributes

The second group of extensions is the one conveying additional attribute to the subject and the issuer in the certificate. There are three of them: *subject alternative name; issuer alternative name; subject directory attributes*.

Typically, the identification (for both issuer and subject) is performed by the DN (Distinguished Name) which uses a strange syntax, for example:

C (Country) = IT,

O (Organization) = Politecnico di Torino,

CN (Common Name) = Antonio Lioy,

… and so on.

That notation, the one with the syntax "*component = value*" is named Distinguished Name and it is a remainder of the X.500 directory, because it is a hierarchy: country, and then the organization, and then the individuals, etc. It could be possible to have under the organization an *Organization Unit* (OU), like a department inside an organization, that is meaningful for the X.500, but it is no more used. For this reason, in the subject name and issuer name it is **compulsory** to use that format, so another place to put the name which is meaningful for the internet applications is needed. In the issuer and subject names, which are standards fields in X.509 v1, we must use DN or leave it empty. Then the other place to put other names that are meaningful are:

- **Subject Alternative Name (SAN):** it allows to use different formalisms to identify the owner of the certificate which are meaningful, for example it is possible to use the e-mail address if the key is owned

by an individual, the IP address (or MAC address) if the key is owned by a device, or the URL if the key is associated to a procedure which is offering a service through that specific URL. So, in general, it contains the meaningful identifier for the applications. There can be more than one SAN. This is always critical if the field subject-name is empty.

- **Issuer Alternative Name** (**IAN**): it allows to use different formalisms to identify the CA that issued a certificate or a CRL (e.g. e-mail address, IP address, URL). This is always critical if the field issuer-name is empty. It is less critical than before, because the issuer is just the issuer of the certificate, while the application is relying on the subject (not on the issuer), because you are protecting IP traffic, you are protecting access to the web, you are protecting your e-mail, you are creating a signed document. So, the same thing as before applies also to the issuer but this is seldom used: there is no purpose in using this. These below are the alternative names that it is possible to use:
  - *rfc822Name*, which is e-mail (e.g., lioy@polito.it).
  - *DNSName*, for example the name of a server (e.g., www.polito.it), and in that case means that the private key is associated to that server.
  - *IPAddress* but remember that a device can have more than one IP address in case there are multiple network cards or it can change the IP address, if you are using DHCP, so it is a bit risky when you use a certificate with an IP address inside it (since you must be sure that that will always be the IP address of that node).
  - *UniformResourceIdentifier (URI)*, if it is wanted to protect a specific access point to a web-based procedure.
  - *directoryName*, if it is used another kind of directory, different from X.509.
  - *X400Address* are the e-mail address used in the old OSI system which are quite similar (also in that case you use Country = IT / … / … / … and so on), but it is a kind of address not used anymore.
  - *ediPartyName*, interesting for *EDI (Electronic Data Interchange)* which is a format used by companies to automatically exchange information about products/parts (e.g., let's imagine that a car manufacturer wants to place an order for tires, then it is possible to use EDI to specify the kind of tires that is wanted, so it is possible to immediately exchange information, there's no need to have a piece of paper with written text "please send me 10 tires of this kind": it is an automatic processing of orders for ordering something. There are various standards, for example one is EDIFACT, which is the one used in factories for providing the goods that are needed for building something.
  - *registeredID*, it is any other kind of official identifier (e.g., there is one code, which is named DUN, which is a unique identifier for companies in the world)
  - *otherName*, it is the escape: if no one of those above is good, then it is possible to define something inside otherName and in that case it must be an **OID plus a value**. With the OID you define your own alternative name and then you put the value of that name.
- **Subject directory attributes**: in addition to names that are identifiers of the entity controlling the private key, it is possible to place inside the certificate some directory attributes associated to the owner of the certificate. So, the subject directory attributes allows to store directory attributes associated to the owner of the certificate. For example, the DoD (Department of Defense), in the United States, uses this field to store the "citizenship" (e.g., italian). The point is that the actual usage of this extension heavily depends upon the application (as no standard definitions exist). So, this is seldom used. It's public because it is in the standard, but the possible values are application-defined, so normally it's used only in a very closed environment. It is non-critical.

Certificate path constraints

This is the class containing extensions related to path constraints. They are:

- **Basic Constraints** (**BC**) is the most important one: it is a flag, and it is always present and marked as critical (even if non-critical) because the basic constraints indicate if the subject of the certificate is an EE (BC=false) or a CA (BC=true). Furthermore, only if BC=true, so if it is a CA, it is possible to define an additional value to specify the maximum depth of the certification sub-tree. It means that,

for instance, if there is a CA with BC=true and max. depth = 2, then that CA can certify another CA, which can certify only another one. So, there can be at most 2 levels under the current one. It may be critical or non-critical, but it is strongly suggested to always mark this extension as critical, since it's an important distinction (if it is a CA or if it is an EE). If this value is forgotten, it is possible to have a certificate which belongs to a normal user, and it may try to be a CA and create other certificates, which is not good.

- **Name Constraints** (**NC**): it appears only in the CA and it is posing a limitation to the space of names that can be certified by a CA (e.g., the Politecnico di Torino has received a certificate from the Italian CA and it could have a name constraints of the kind RFC822Name, because the syntax is the same as the alternative names, and then the restriction is **@polito.it*, so the constraint is: if there is an e-mail address in the certificates that are created, than it is possible to satisfy only addresses of the Politecnico di Torino). In the same way if there is a certificate for a CA that certifies network nodes, it could say that it is possible to satisfy only addresses that belong to the same class of IP addresses of the CA, and not for other addresses. We can have at least one specification between:
- *PermittedSubtree* (i.e., whitelist) (e.g., the one before: RFC822Name, with "*@polito.it")
- *ExcludedSubtree* (i.e., blacklist)

  Whitelist is processed always first. If something is not specified in the whitelist (for example directoryName is forgotten) then it is implicitly permitted. So, it is much better to place that in the excludedSubtree to avoid that kind of things. It's critical or non-critical (should be marked as non-critical to avoid compatibility problems with Apple products, since, due to a bug, if an Apple device receives a certificate with Name Constraint (NC) it is not understood and it reject the certificate).

- **Policy Constraints (PC**): used only by CAs to specify the constraints that could require an explicit identification by a policy or that inhibit the policy mapping for the rest of the certification path. This is seldom used, all this matter about the policies which is good as long as a pointer is provided, but all these policy mappings and things like this exist in the specifications, but in practice they are seldom used. It is critical or non-critical.

CRL distribution point

The last class contains only one extension, which is named CRL distribution point.

- **CRL distribution point** (aka CRLDP or CDP): it means that inside the certificate there is a pointer to the place where it is possible to download a CRL related to this specific certificate. This is important, because if there is someone performing a signature (of course with a private key) and it is sending to a user the certificate (with the public key). It is possible to perform a cryptographic verification on the signature but then another check must be performed: "is this certificate (trusted and) valid or not?". One possible way is to check if the certificate appears in a CRL (if it has been revoked), but to get the CRL inside this certificate (the one that I received from that someone) it may be possible to have the CRLDP, which is a pointer to the internet to that CRL. So, when a certificate is received, if the certificate contains the CRLDP (because this is an option, it is an extension), then it is possible to follow that pointer, download the CRL and so it will be known if the certificate is revoked or not. There are 3 ways for a pointer: one is directory entry (so it points to somewhere in a directory), or e-mail, or URL. The best choice is the URL, because it directly downloads the CRL. If an e-mail address is used, an e-mail is sent and then the CRL is received back in another e-mail, which is risky, because the CRL can be quite big: it might do not fit inside an e-mail. So, URL is suggested. It is critical or non-critical.

Private Extensions

It is possible to define private extensions, that are extensions common to a specific user community (i.e., a closed group). They are supported as syntax by X.509 v3 but the semantics is undefined and must be defined by those who create the private extensions. It is normally discouraged since it does not allow interoperability but there is an important case: **IETF-PKIX** defined **three private extensions** for the Internet user community.

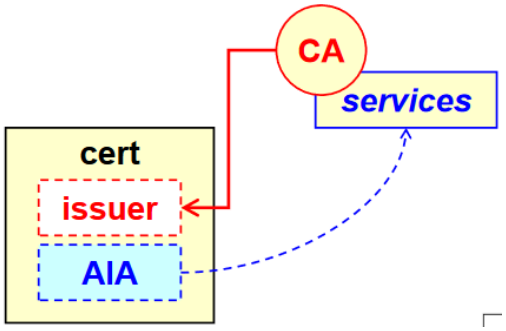
Actually, they are considered public extensions since it is assumed that anybody knows the standards used in Internet. The extensions are: Subject information access, Authority information access, CA information access.

Subject information access

When a certificate is received there is the subject identified with the DN (*distinguished name*), which uses the old X.500 format, or there is also the SAN (*Subject Alternative Name*) e.g., RFC 822. Those are **certain identifiers**. With subject information access there are two things specified: the **method** (e.g., http, ldap) to obtain additional information about the owner of a certificate and a **name** that indicates the location (*address*). This is particularly useful when a directory is not used for certificate distribution. The DN is like an entry in the directory, but if there is no directory (since X.500 is old) the DN normally doesn't point to anything. The SIA is a different way to provide more general information: it can be a pointer to a link, a phone number or a photo.

Authority information access

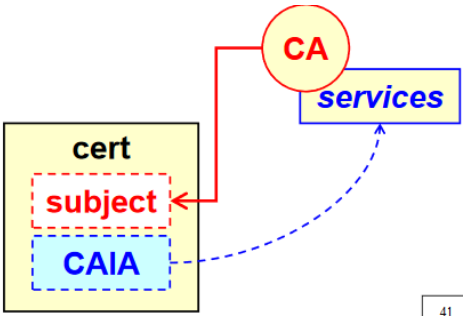
On the contrary, this is a very important one. When a new certificate is created for someone, it is issued by a CA. The AIA is a **back pointer** that goes **from one issued certificate to the service offered by the CA**. Among those services the most important one is the **certStatus**, because in that field it is possible to get the address of the **OCSP Server**. When a certificate is received and OCSP is used to check the validity, it is possible to look at the back pointer offered by AIA. In the AIA may exist other pointers (not compulsory) like: *certRetrieval* to retrieve the certificate of the CA itself, *cAPolicy* retrieves the policy of the CA, *caCerts* retrieve the certificates issued by the CA. It can be **critical or non-critical** but always marked as non-critical since there could be other ways to i.e., check the status of the certificate.



CA information access

In this case it is a **self-pointer**: the concept is the same, always a pointer to the services offered by the CA but the CA is the one which **possesses** the certificate. In the previous case it was a certificate issued from the CA while not it is the CA itself and the pointer is a self-pointer so that when the certificate of a CA is read, it is immediately clear which services it offers.

Again, in the certificate status there can be the OCSP pointer and it is marked again as **non-critical.**



RFC-2459

It is the original one which defined the **profile** of X.509v3 suggested by PKIX for use in Internet applications (e.g., IPsec, TLS, S/MIME). It tells among all the various fields and subfields that to use X.509 certificates with standard internet mechanisms there are here all the **interoperability suggestions**. The extensions are defined starting with the base OID that corresponds to the PKIX group: $ID - PKIX ::= 1.3.6.1.5.5.7$. It contains the list of **supported algorithms** as well as **suggestions for practical implementation**. E.g., inside a certificate there is always the **validity period** (from-to) in which to put date and time. Sometimes it may happen that someone tries to insert a local time but it is not unique because there is no information about the Time Zone of a specific time. It is important to specify **how dates and time are inserted** and in this case it is used **UTCtime**, **seconds are mandatory** and in order to achieve availability interoperability it is **mandatory to use Zulu time (GMT)**. Additionally, there should be always the year specified with **four digits**. In case only two digits are used, then the year must be interpreted as following: $51 \rightarrow 1951; 47 \rightarrow 2047$.

Extended key usage

In addition, or in substitution of *keyUsage* (that specifies for which cryptographic application the certificate can be used) since it was a field to specify cryptographic operations. This new field is oriented towards application. They can be used **simultaneously** and in case of conflicts there is the need to define which one has the priority. Some possible values are:

- (id-pkix.3.1) serverAuth [DS, KE, KA]
    - Server authentication like the one in TLS and the square brackets contains the compatibility flags for the basic keyUsage or the substitute. If a certificate for a web server is needed then it is possible to have one certificate that contains only server authentication or it is possible to have a certificate that contains only the basic key usage: Digital Signature (DS), Key Encryption (KE) and Key Agreement (KA). DS is needed to perform the signature on the challenge-response while KE and KA are the two possible to create a common key with the client. It is possible also to have a certificate which contains both of them (serverAuth + DS, KE, KA).
- (id-pkix.3.2) clientAuth [DS, KA]
    - In this case there is no KE as compatibility, because in TLS the client is typically creating the key and then sending it to server, so KE is not needed. At most KA, in case Diffie Hellman is used, may be used. DS is needed or is not possible to authenticate the client.
- (id-pkix.3.3) codeSigning [DS]
    - It is particularly important nowadays since many attacks are made by trying to inject malware inside the code developed by someone else. Developers who are creating software may want to use a signature to prove the integrity and the property of the code. If a certificate contains only codesigning flag, then it can be used **only** for this purpose (big restriction).
- (id-pkix.3.4) emailProtection [DS, NR, KE, KA]
    - It is the one with more compatibility flags because there is not only digital signature to authenticate emails and protect its integrity, not only KE and KA in case there is the need to have email encryption, but also **non-repudiation (NR)** because since X.509 is used, then the author of the signed mail cannot deny to have created that mail.
- (id-pkix.3.8) timeStamping [DS, NR]
    - It is a feature in which a server would create a signature containing date and time. Basically is like attaching date and time to a document. Contains DS but also NR.
- (id-pkix.3.9) ocspSigning [DS, NR]
    - It is the signing of an OCSP response. Contains DS but also NR.

Be careful: normally, NR is associated to an action performed by a human and performed voluntarily. With *timeStamping* and *ocspSigning* there are two exceptions in the sense that NR is wanted, meaning that the operator of the server is providing a service for security so it is not possible to deny telling someone, e.g., that a certificate was valid/invalid (and will take the responsibility of saying so).

Evolution of RFC-2459

RFC-2459 was the first incarnation of the PKIX profile. They later decided to split the content of the document in two parts: RFC-3280 + RFC-3279. The RFC-3280 defines the Internet **profile** of PKIs for certificates X.509v3 and CRL X.509v2 which later became obsolete by RFC-5280. The RFC-3279 documents the algorithms (identifiers, parameters, and encodings) used by RFC-3280. In this way it is possible to evolve independently, especially the algorithms, since they can be updated frequently in respect to the state of the art.

RFC-3279

This RFC specify the **algorithms that MUST be supported by the application** that use the RFC-3280 profile. It contains also quite old algorithms. They are:

- **For digest:**
  - MD-2, MD-5, SHA-1 (preferred)
- **Signing of certificates / CRL:**
  - RSA, DSA, ECDSA (Elliptic Curve DSA)
- **Keys of the subject (SubjectPublicKeyInfo):**
  - RSA, DSA, KEA, DH (Diffie-Hellman), ECDSA, ECDH (Elliptic Curve DH)

The underlined algorithms added in RFC-3279 with respect to RFC-2459.

Once the basic definitions are given, then it is possible to define optional algorithms. It is important to note that:

- RFC-4055: it adds better specification for the disgnature such as the *Probabilistic Signature Scheme*, the *Optimal Asymmetric Encryption Padding* and SHA-2 functions to compute the digest.
- RFC-4491: it satisfies the needs of Russia, since it contains specification for using the **GOST** algorithm family that are the ones used in Russia.
- RFC-5480: it adds the Elliptic Curve keys of all kinds
- RFC-5758: it adds support for new algorithms of SHA-2 family such as SHA-224 and SHA-256.
- RFC-8692: uses for the signature SHAKE128 and SHAKE256. SHA-3 is a family of algorithms, and it contains SHAKE128 and SHAKE256.

The RFC-3280 / -5280

It contains the profile which specifies not only values and fields but also **algorithms,** procedures, since **certificates should be processed in the same way all over the world**.

- It specifies the **path validation algorithm:** when there is an EE certificate, it was created by a CA that received a certificate from another CA and so on up to the root. It is possible to build or verify the chain by looking what is specified in the RFC.
- It specifies the details about the verification of the status of a certificate using both the basic or full CRL as well as the **Delta-CRL**.
- Adds an *ExtendedKeyUsage* (the *OCSPSigning*)
- Extensions for certificates
  - Inhibit any-policy: forbids the use of any policy. It means that the CA cannot avoid specifying its policy
  - Freshest CRL: pointer to a new kind of CRL (*Delta-CRL*) which contains only the difference with the basic CRL
  - Subject information access already discussed.
- Freshest CRL is also available for the CRL itself: if there is a certificate or a CRL, in both cases if there is the Delta-CRL is pointer than it is possible to access the differences with the last full CRL.
- The freshest CRL is the Delta CRL Distribution Point, which is always noncritical because there are many ways to check the status, so it is not possible to make one of them mandatory (full CRL, Delta CRL, OCSP).

Certificate revocation

Every time there is any kind of transaction which relies upon certificates providing some security features, there's the need to verify that the certificate is valid. Valid does not only means "to perform cryptographic verification" because when a certificate is received it is needed to check that it is cryptographically signed, verify that has been issued by a CA, that has been certified by another CA up to a trusted root CA, check if the validity period is ok. Even if all these things have been verified the certificate could be **invalid** because it has been **revoked**. For **each certificate** in the chain, it must be verified that each certificate is "ok or not" by a revocation point of view.

A certificate may be revoked before its natural expiration because of many reasons:

- **Upon request of the certificate owner**, that is the **subject** in certificate. This is typically due to a key compromise or a loss of the private key. Those two things are not equivalent, they are equivalent for revocation in the sense that the user will not own anymore the key, but the security implications are very different, because if the key is compromised means that someone got a copy of the private key and can act on his behalf. If the key is lost (maybe the file containing the key has been destroyed) in that case no one will be able to use the signature.
- **Upon request of the certificate sponsor** (**organization**). The certificate sponsor is not something evident in the certificate itself. Typically, the issuer act on behalf of someone else, e.g., Politecnico di Torino receives certificates from one company (*infoCert*), but the sponsor is Politecnico di Torino. The certificate is given to a teacher as an employee and when he is no more an employee of Politecnico the certificate should be revoked, because inside the certificate is written that he is an employee. The same if the company goes out of business.
- **Autonomously by the issuer** (the CA) due to an error if by mistake issued a wrong certificate, or due to a fraud.

Certificate status **MUST** be checked by the entity that accepts it to protect a transaction. For example, if someone is making a commercial offer and it is performed a digital signature, the one who receive the document with the digital signature *must* verify the signature. This is typically named **relying party** (RP).

Mechanisms for checking certificate status

There are various solutions to verify the certificate status:

- We do not consider a single certificate, but the certificate chain up to a trusted root and check if the certificate is expired or not and if all certificates are valid.
- If not expired, a *Public Key Certificate* (PKC) is valid unless otherwise stated. There are two possible mechanisms:
    - **CRL** (*Certificate Revocation List*) – is a list of revoked certificates (i.e. check it by yourself, if the specific certificate supporting your transaction is valid or not). The CRL must be protected because otherwise it could be possible to cancel, invalid certificates, or add valid certificate, so it is signed by the issuer (if possible) or sometimes by a delegated authority, often named **Revocation Authority**.
    - **OCSP** (*On-line Certificate Status Protocol*) - answer to precise queries. It provides an answer about the validity of one specific PKC and the validity is for *today* (**for this instant**). The question is "*Is the certificate valid now?*". To avoid fake answers, the answer is normally signed **by the server providing the answer** (not by CA otherwise it would need to be online and could be attacked). Having a signature by the server means that there is a problem of trust, because the signature must be performed with another certificate, so there is the need to verify the certificate of the OCSP Server.

X.509 CRL

CRL means Certificate Revocation List. It is the list of revoked certificates and for each revoked certificate there is also the **revocation date** and the **reason**. The *reason* is important as previously said for those application problems: if a key was copied or lost are two completely different issues.

CRLs are issued **periodically** and maintained by the certificate issuers. This means that there is the need to **re-issue a CRL even if nothing has changed**, just to give to the relying party assurance about its "freshness". This is done to avoid replay attacks, because an attacker can take an old signed CRL and submit it, as it is a new one. For this reason, in the security policy each CA is declaring what is the life of the CRL. At PoliTo there is a lifetime of at most 35 days.

CRLs are digitally signed by the CA that issued the certificate or by a revocation authority delegated by the CA. In this case, in which the CRL is not created by the CA itself, it is named **iCRL** (which is not the CRL of Apple, funny Lioy 😊) which means **indirect CRL**, because it is not created, not signed by the issuer itself but one delegate. If a revocation authority needs to sign an indirect CRL, it needs a certificate that as **KU** (key usage) has the field *CRLsign*, because that is attesting that the RA has been delegated by the issuer to sign the CRL. Otherwise, anybody who has got the signature could sign the CRL. The KU in some sense is also an authorization in this case.

X.509 CRL version 2

In the picture on the right there is the structure. *CRL v1* has been replaced quickly by the *CRL v2* and there is not a version 3. CRL version 2 goes on the same level as X.509 v3, because also CRL v2 was extended as well as X.509 certificates have extensions.

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
}
```

A CRL is a sequence of a *TBS certificate list* (TBS = *to be signed*), so the list of revoked certificates that must be signed, then the *Identifier* and the *signature* itself. The list of certificates to be signed is a sequence with the *version*, the *AlgorithmIdentifier*, the *CRL issuer* (who created the CRL), *thisUpdate* which contains the date and time in which the CRL was created. Optional, but extremely suggested, is *nextUpdate* which is a **promise**, even if nothing has changed at least in this date (not exactly in this date, maybe earlier) there will be a new one. *revokedCertficates* is a sequence in which each *userCertificate* is identified by the *CertificateSerialNumber*, the *date* and *time* when the certificate was revoked, and then there are **extensions of the single entry** and then there are **extensions of the whole CRL**. **Without the extensions** the *CertificateSerialNumber* is referring **to the issuer that must be the CA**. If we really want to create an indirect CRL this mechanism does not work, because the issuer is not the CA. In the basic structure the assumption is that the issuer is the CA; if an indirect CRL is being used, there is the need to use the extensions as we are going to explain.

Extensions of CRLv2

crlEntryExtensions:

- **Reason code** – provides the motivation, so why the certificate was revoked (e.g. key compromised, out of business, not affiliated anymore, etc.)
- **Hold instruction code** – this is discouraged by the IETF but not for example by the ARMY. They try to limit as much as possible all the possible incidents. For example, if a soldier goes on vacation he must not be able to perform any official action, so the certificate is put on HOLD, so there is one CRL_N in which the certificate C is marked as **hold**. When the soldier goes back in service, there will be another CRL_K in which the certificate C is removed from hold. Is like a **temporary suspension** of a

certificate. This is not appreciated by internet because in the next CRL_{K+1} the certificate does not appear anymore. It means that someone takes the last CRL it will never know that the certificate was not valid when it was in the HOLD status. This is a problem because it obliges to keep the history of all CRLs. The hold status is a temporary one while all the revocations are permanent.

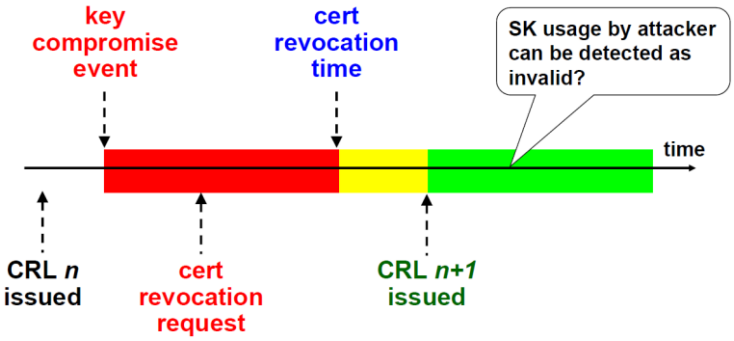
- **Invalidity date** – date and time from which the certificate is no more valid.
- **Certificate issuer** – if the issuer mentioned in the CRL is an indirect issuer is the Revocation Authority then for each certificate it must be specified the *serial number + the issuer*.

crlExtensions:

- *Authority key identifier* – similar to what we had in the certificate of the CA. It tells which keys, among the various, have been used.
- *Issuer alternative name* – in case we do not want to use the DN (distinguished name) but other identifiers.
- **CRL number** – this is important and useful only when creating *Delta-CRL* because it contains only the differences. CRL is big, if every time there is the need to re-create the whole CRL and other people need to download it, it is a waste of time. It is possible to create a base CRL and then use a delta-CRL indicator, that means that it is not the complete CRL but only the difference with respect number '*x*'. So, it is needed *the base + the deltas CRL indicator* to understand what the current CRL is. This is used to reduce the download time.
- *Delta CRL indicator* – explained above.
- *Issuing distribution point* – this is a pointer to a server to get the CRL. When a user has a CRL, it is possible to see from *nextUpdate* that is time to get a new one. If we placed this extension, we know where to get a fresh copy of this CRL.

But we have two similar entries: one is *revocationDate* and the other is *invalidity date*. Are these duplicates? Let's see the difference.

Certificate revocation timeline



On the horizontal axis we have the time from zero to infinite. At some point in time the CRL_n is created. Then there is a *key compromise event*, (e.g., someone stolen my credentials) so I should run to the CA and do a *certificate revocation request*. The time that elapses until the certificate is revoked is named *Certificate Revocation Time*, the one marked as *revocationDate*. But in the time when the credentials were compromised anything could be happened, but the CA does not know if this is truth or I am trying to cheat on them. So, the Certification revocation time will be the *revocationDate* and it is certified by the CA, the key compromise event can be optionally inserted and if present will be the *invalidity date*, but the responsibility in court is different because the first one oversees the CA/RA, the second one is in charge of the subject. Everything that happens in the red zone is dangerous because if someone is using the key in my place sending something to a Relying Party, the RP will get the CRL and verify that the certificate is good, and it will accept. The yellow zone is also risky because the CA takes a bit of time to create the CRL. So, the CRL_{n+1} will contain the revoked certificate and will be available to everybody. If the certificate is used in the green zone, the RP will check the CRL and will not accept it because it's revoked.

The yellow zone is also dangerous because the CRL has not been published yet. If we are a relying party when we receive something (protected in some way) in addition to check if the certificate has been verified or not, we should check the certificate policy to see how much time the yellow period is, since that is the **wait time**. It could be maybe 3 minutes, but also 24 hours or 48 and so on. That means that we should not check the latest

available CRL because that can be an old one. Moreover, what happens for CRL happens also for OCSP: in the green period CRL_{n+1} is issued or **OCSP database is updated**. That can be even worse, because maybe the OCSP server is managed by someone else that sometimes gets the new CRL and uses it to feed the OCSP server. It is not enough to say "I checked the revocation status" but we must also check what is happening to the time. If the key is used in the yellow period, who is responsible for that? It is the RP because in the moment in which is accepting the certificate it has the duty to verify the certificate policy (and cannot ignore the yellow period, so should wait for it).

Another question is, what if the *private key* (SK) is used during the green period? Are we protected? So I am the RP, I verify the certificate, download the CRL that says me that the certificate is revoked. Apparently, I am safe but maybe not, it depends upon the kind of transaction because if a document signed one year ago is received, the verification to perform is if the key was valid one year ago, and it was valid. This is named the problem of **Time of Use Time of Verification** (*TUTV*).

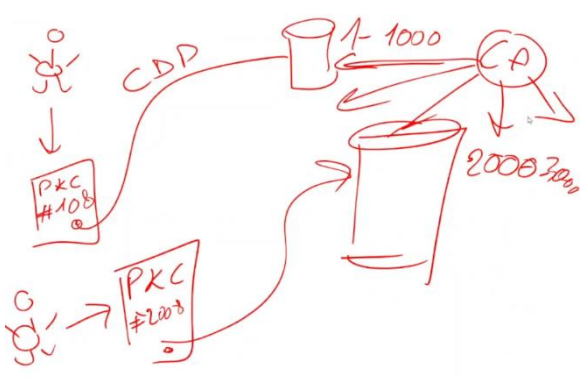
If time of use and time of verification are coincident in the green zone, like for online services, we are safe. But if they are different and typically time of use is older, there is a problem; who tells us that the document was signed one year ago? It could be possible to create a fake file with a fake date and sign it today. If date and time is self-asserted, we are exposed to that kind of attack. To use the CRL we need to be sure also about the date and time in which the key was used. If it is online service, there are no problems (it is signed now).

Efficient management of CRLs

CRLs can become very big because they contain the list of all certificates revoked up to a certain date and consequently become costly to download and to examine. There are various solutions:

- **Eliminate the revocation following the first CRL issued after the expiration date of the certificate** – (Lioy does not like this solution). If I have the certificate that was valid until a certain date (e.g., 12/31) and it was revoked (e.g., 10/5), the first CRL issued after the expiration will still contain the revoked certificate, but the one after that no more. That is meaningful if using the certificate for online transactions. But if the certificate is used for signatures that were performed long time ago if the certificate does not appear anymore in the CRL there is no way to prove it unless an archive of past CRLs is kept. It means that the CA does not care, and it's up to the RP to keep the burden of storing the CRLs.
    - Example: student alpha got 30 and Lioy creates a signature on the date of the exam. The signature must be verified for any of the students listed for graduation and this happen not at the same time. In the moment of graduation, he has to check if that signature is valid or not and if the revocation has been deleted nothing can be said. So, it means that when you sign something you must keep the *signature*, the *certificate chain* related to that certificate and the *revocation chain*. This means that for each certificate in that chain I must get the next CRL and store it by myself. Only in this way It Is possible to prove also in the future that certificate was valid in the moment of the signature. It is important also when the document was signed because a fake date could be used. This depends upon the procedure, because for Politecnico the document and the signature are server-based and in this case date and time are trusted because they are not created by the signer but inserted by the server.
- **Publish complete CRL and then only the differences** – in this case a CA is creating one *base CRL* (number N) and then creates *delta CRL* (difference with respect to N). This means that if you want to verify if a certificate was revoked or not you must get the base and all the deltas until the next base CRL. The download is faster but additional work is needed to **build the complete CRL**.
- **Partition the CRL in groups** (by properly using the CRLDP) – inside the certificate there is one option called CRLDP (*CRL distribution point*).

- Example: A person is sending me the Public Key Certificate, I would like to know if the certificate is revoked or not and I use the CRLDP to point to one CRL, but maybe this one is the CRL containing only the certificates between 1-1000. So, if the number is #108 it will be there. But if another person with a different certificate #2008 the CRLDP will be a different one with entries from 2000-3000. It means that one issuer does not have one CRL but has got several CRLs. If I want to know what all the revoked certificates are of a specific CA I have to download all the pieces from all CRLDP. There is no obligation, each CA will decide the way how to manage it. Therefore, they have *certificate policies* (theory) and *certificate practise statement* (how in practise we do things).



OCSP

On-line Certificate Status Protocol. It is a client server protocol to verify if a certificate is valid **NOW** (this is the major limitation), it **cannot answer about the past**.

- Example: If, as before, Lioy is creating the grades for an exam and there is a signature on them but does not have a CRL and this document has to be verified in 3 years it is needed **in the moment of the signature** to query the OCSP server and to save the OCSP answer. We do not archive the CRL but the OCSP answer, with all the problems related, because the OCSP answer is signed by the server itself, so the server needs to be trusted and to also keep the certificate of the server.

OCSP provides an answer about the validity of the certificate. The answer can be only:

- *Good*
- *Revoked* – providing also the *revocationTime* and *revocationReason* (similar to CRL)
- *Unknown* – if we are requesting information about a certificate that was never issued.

The responses are digitally signed by the server to avoid fake responses. The signature certificate of the OCSP server **cannot** be verified with OCSP itself. OCSP can be used directly (as a protocol) but most of the time is *encapsulated* within HTTP or HTTPS, in order to get some additional security. Even if there is HTTPS anyway the **answer is signed by the server**, we do not rely on TLS to protect the transaction.

Attacks against OCSP

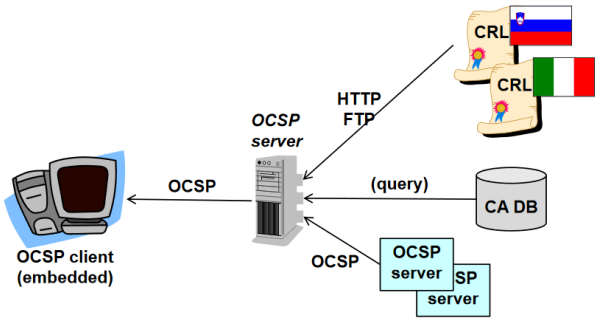
Some attacks against OCSP are:

- **Replay attack**: it is possible to copy the whole response saying that the certificate is valid and replay it after the certificate is revoked. If the question is "is certificate number 23 valid?" and it has been answered in the past it is possible to take an old answer for that certificate and replay it. But if there is also the date and time in the question the attacker cannot reply with an old answer. To defend against this attack, it is needed something that strictly relates the answer with the question, which is typically a **nonce**.
- **DoS attack**: by relying on the OCSP it is possible to try to flood the server with many requests, because each request requires a signature in real time which is a heavy process. The defence is to **pre-compute responses** and put to them 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).

OCSP source of information

The OCSP gets information about revocation by simply read public CRL repositories or directly from the CA database, so the CA is not creating a CRL but simply giving access to its database or it is possible to ask to a "colleague" OSCP server like a chain. There are some OCSP servers that are acting as a front-end. Let's suppose that a company is accepting certificates from CA#1 and CA#2 (the two in blue in the picture). In order to avoid complexity, it is possible to setup an own and proprietary OCSP servers with all browsers pointing to it. This server will act as a proxy and when it receives a query it will pass along to the appropriate OCSP server, and then the answer is stored.



Models of OCSP responder

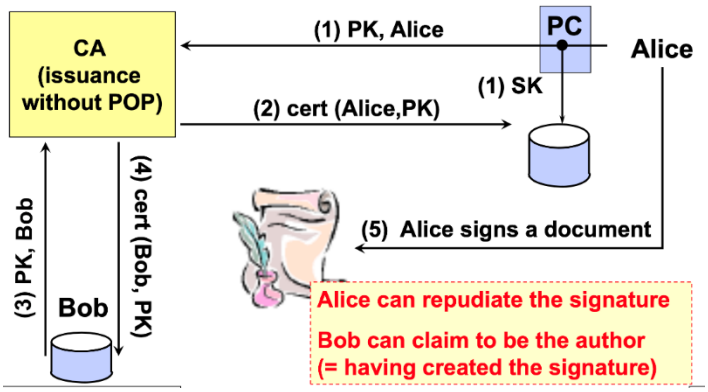
There are some types of models:

- **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.

Proof-Of-Possession (POP)

When the CA creates a certificate, they should have some reasonable assurance that the requestor is controlling the corresponding private key. If there's no such verification there are problems because if there is the issue of a certificate without checking the possession of the corresponding private key then there may be a problem with **non-repudiation**, which it's not always critical for encryption, but if non-repudiation is wanted (and it is anytime there is a digital signature) then POP is an important issue.



For example, there's Alice with its own device and a CA that issues certificates without POP. 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.

Countermeasures

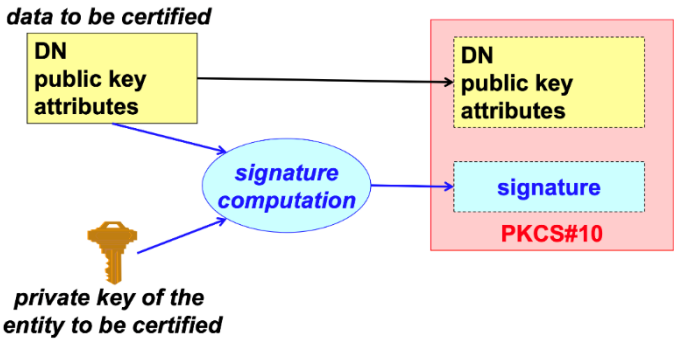
The best thing is to perform POP in the moment of performing the signature because in the moment in which the signature is created it could be possible to put a reference to the certificate that associate the public key to the identity, so typically the hash of the certificate is inserted together with the data. In this way the signature value is a function not only of the data, but also of the certificate. In this way non-repudiation is gained. Unfortunately, this is **currently unsupported** in signature standards, you must create a custom protocol.

The alternative solution is that the CA doesn't give the certificate to Bob, so when the CA is creating the certificate, it wants a POP when it receives a request, asking for the private key as POP. The key is usually sent out of band, this means that the keys are created by the CA and delivered to the user via a token (the smart card in Politecnico for example). The possession of the token is a proof. Another solution is: 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. When a request is performed there must be also a signature (which means using a private key), even if there is no certificate. In this way it is demonstrated the possess of the private key. For encryption keys, not used for signature, it is possible to use a challenge response protocol where the certificate is sent encrypted, and the other side must be able to decrypt it and send it back as a proof that of having the private key.

PKCS#10

This is the most used method, and it is documented in RFC-2986. It's also named **CSR** (*Certificate Signing Request*) and the request contains the **distinguished name** (the identifier that is wanted), the **public key** and optionally other **attributes**. There are some attributes that are quite important: the **challenge password**, it is a *one-time code* given from the registration authority that is useful for registration and revocation. If someone has stolen the private key, it is needed to ask for revocation, but maybe the CA is in a different place than where the victim is. To demonstrate that the victim is the owner of the certificate, it will use the special code for revocation given in the moment of creation of the certificate. There could be also other information or attributes about the requestor.

The schema shows the data to be certified: DN, public key and attributes. These elements are inserted in the first part of the PKCS#10. Since the POP is wanted, the private key is used to perform a signature over the data to be certified and the signature is placed in the second part of the PKCS#10. So, when the CA is receiving the signed data, it will verify the signature using the public key. In this way is possible to demonstrate that the user is the owner of the private key associated to that public key and then emits the certificate. This is the most used approach nowadays: POP at the moment of the request by performing this kind of signature.

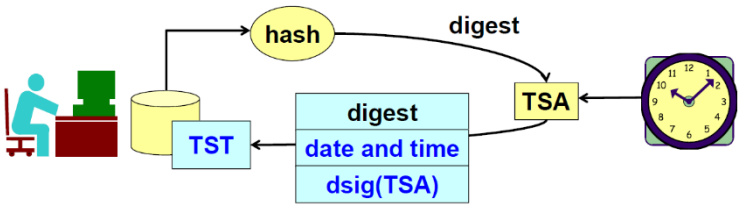


Time-stamping

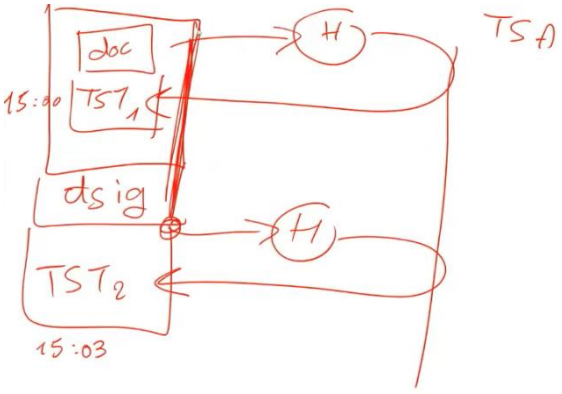
It's important in any workflow related to signing documents because it's important to be able to establish when a document was created. Be careful: many people wrongly think that timestamping is giving the date when a document was created, but it is false. Timestamp is a **proof of the creation of some data before a certain point in time**. In other words, 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.

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*).

The user has some data created at some point in time. It is a blob: can be encrypted data, plain text data, signed document, anything. The user wants to time-stamp those data 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 and will create the TST (blue structure in the picture) that will contain the *digest*, *the date and time* and the *digital signature* of this data performed by the time-stamping authority.



The "*dsig(TSA)*" is a signature of digest+date and not of the document and the signature keep both element together. The whole element is the TST, which is returned and then the requestor can then attach it to the data. If data is changed after receiving the TST token it will be detected because the digest will change. This is very useful when there is a deadline, because it is possible to demonstrate that data was created before the deadline. Again, this does not demonstrate in any way when the document was created. In case the data are a document with a digital signature, this does not say anything about the signature but the fact that it was created before the time stamping. Furthermore, if it is wanted to proof when a signature was performed, then it is possible to use two timestamps.



Looking at the picture, there is a document with the hash of the document sent to the TSA and then the TST_1 token put together with the document. Then, a signature is created over the doc+TST_1 and everything is hashed and sent again to the TSA. The TST_2 is computed over everything and that demonstrate that **document was created before that date and time** (doc ∃ < 15:00) and **the signature was created between the two**

timestamps (15:00 $< dsig <$ 15:03). It is still not possible to say when the document was created exactly. In Italy it's not called "timestamping" but "marca temporale".

PSE (Personal Security Environment)

Let's talk about private keys. Private keys have different requirements: they must be protected because they must be used only by the owner. This is normally performed through a PSE (*Personal Security Environment*). Note that most of the people think that PSE is useful only for private key. The PSE should also be used for **the certificates of the trusted root CAs**, that must be authentic. There is a famous incident in an application for verifying digital signatures (Italian company). That company protected the private keys of the customers but forgot to protect the trusted root CAs, so someone was able to add another root and then create a certificate for a guy named *Arsene Lupin* and that was perfectly valid, according to Italian law, when verified with that application. Root CAs do not need confidentiality but they do need authenticity.

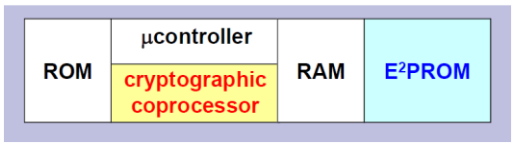
It is possible to implement PSE in **software**: typically it's an encrypted file because it contains also the private key and typically also the root CAs. Root CAs would not require encryption but it is done anyway since there is also the private key.

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.

Using the keys in different environments (*mobility*) is possible in both cases (software and hardware) but this brings to several problems.

Cryptographic smart card

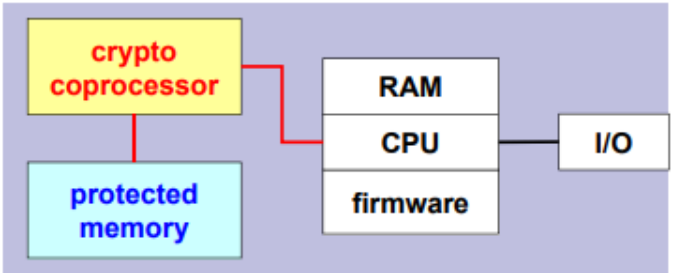
Typically, the used device was a cryptographic smart-card, which is a chip card with memory (at least) but it has also autonomous cryptographic capacity. The keys need to be managed by a trusted system, if there is an encrypted file but then the CPU is used to perform the computation it means that the key must be put in clear in RAM for the CPU to be used and if there's a malware then the key is stolen. On the contrary, 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.

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.

HSM - Hardware Security Module



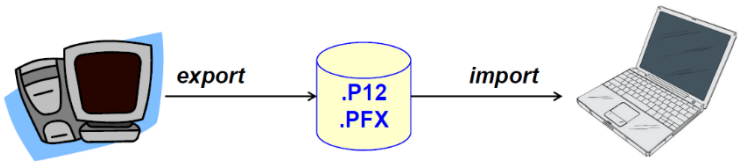
It is considered a cryptographic accelerator, because it not only has got the *protected memory*, where keys are saved and protected (even if an attacker got that memory, keys cannot be copied) but there is typically a *crypto coprocessor*, that implements the required cryptographic functions, typically for a server this is for asymmetric crypto (e.g., RSA), but also for symmetric crypto because the I/O of this module is very fast, it can be a PCI board, external device (USBI, SCSI, …) or IP network device (netHSM). This module is nearly a must if you are serious about electronic commerce, since a TLS server of this kind needs an HSM. This creates a problem with cloud network. In fact, if a server is hosted in the cloud, then it is a virtual server, so it is moved from one node to another, and if there is an HSM, it is mounted in a physical machine. So how is it possible to connect a cloud server with a physical HSM? Various cloud providers have different solutions for this (virtualized HSM, …). This is an interesting topic for a thesis.

ISO 7816-x standards for smart-card

That's important since it is needed **interoperability** between smart-cards and devices. They are all in the ISO 7816 and they are:

- Physical format
- Contact characteristics
- Electrical signals and protocols
- Inter-industry commands
- Application identifiers
- Inter-industry data elements
- SCQL (SC Query Language)
- **Inter-industry security commands**
- Commands for card management
- Electronic signals and answer to reset for sync cards
- **Personal verification through biometric methods**: smartcards integrated with biometric
- Cards with contacts – USB electrical interface and operating procedures
- Commands for application management in multi-application environment
- **Cryptographic information application**: specific for cryptographic operations

PKCS#12 format (Security Bag)



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. It contains a private key and one or more certificates, both personal and root ones. Normally, it is used to transport the digital identity of a user. In this way it is possible to move data from Google to Firefox, for example, exporting that key using PKCS#12 and importing it in system that is wanted. 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.

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
  - "-text" 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:

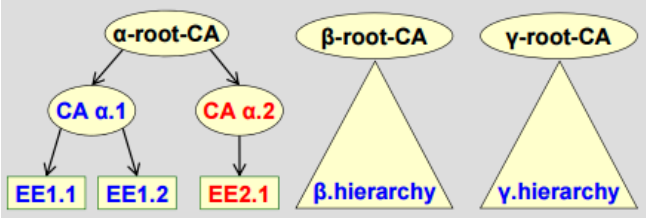
- **DER** (*Distinguished Encoding Rules*) which is a **binary encoding** of ASN.1 (it's a subset of BER, Basic Encoding Rules)
- **PEM** is the **armoured base64** encoding of DER

Then, in the picture on the right there are some useful options for openssl, useful for the lab.

`-in F`	input file
`-inform X`	input format (DER, PEM)
`-out F`	output file (only DER)
`-i`	indent the text output
`-noout`	do not provide the PKC as output
`-offset N`	begin the analysis from byte N
`-length N`	consider only the first N bytes
`-dump`	hexadecimal dump of the unknown data
`-oid F`	file containing supplemental OIDs

Hierarchical PKI

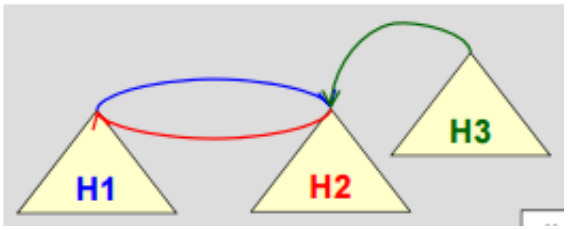
The certificate chain is rooted at some trusted place. That is not the only model. There are various PKI models. The one used so far is the **hierarchical PKI**, which is also the oldest one.



The Hierarchical PKI is a tree rooted at a self-signed **root** CA. While CA $\alpha.1$ was signed by the α-root-CA, the α-root-CA is self-signed. In a hierarchical PKI is very easy to build a **certification path** between any two EEs (end-entity). To understand the relation it is just needed to go up to the chain until reaching a common ancestor. This is the model supported by **all** applications. But due to legal, commercial, and political problems, there is **no single hierarchy**, but different hierarchies exist, letting the PKI becoming a **forest** of CAs.

Mesh PKI

An alternative is to create a **Mesh PKI** model, in which two hierarchical PKIs may unilaterally or bilaterally decide to trust each other by issuing a so called **cross-certificate.** Here some examples:

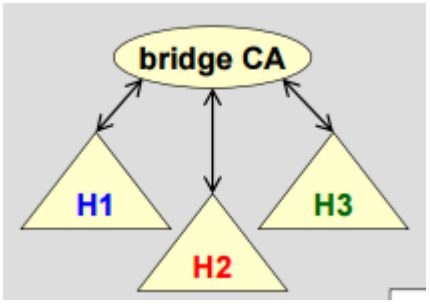


In the first picture, hierarchy H3 is trusting H2 (*unilaterally*). On the contrary, H1 trusts H2 and H2 trusts H1 (*bilaterally*). This kind of relations are expressed by the cross-certificate that is a **certificate issued by a root CA for another root CA**. The problem is that those have been defined in the standard, but they are not automatically recognized by standard applications, because if there is a cross-certificate, then the application does not know which certificate chain should consider. For example, if a certificate starts from H1, the application should stop on the root CA of H1 or continue to H2 using the link? There is no single path but multiple options.

Another problem is that if there are several hierarchies, then if complete trust among all hierarchies is wanted then a huge number of cross-certificates is needed, and it that grows as the square of the number of independent hierarchies $\frac{N(N-1)}{2}$**.** It is rarely used in practice due to the problem in applications.

Bridge PKI

The most used alternative to the hierarchical PKI is the **Bridge PKI**, that is a tradeoff of the previous model.



In the Bridge PKI, a special certification authority has been added, and it is called **Bridge CA**. It is cross-certified with each root CA (the black arrows in the picture) and this reduces the number of certificates (no more n^2 but n). In this way complete trust is reached with just N cross-certificates. Again, it is not recognized automatically by applications unless special applications are used. The most notable example is the US which created the federal PKI and modified the browsers to be used in the government office to recognize this bridge CA. More info is available at https://fpki.idmanagement.gov/. The problem is that maybe there is a TLS server in H1 that uses client authentication, but client has a certificate issued in H3. If server would perform verification, it would say that it is not valid, but if the server is able to go through the bridge and down on H3 then it would accept it. This is way a modified browser is needed.

What to do with PKC mistakenly issued or issued by Compromised CA?

It is difficult for domain owners to detect certificates fraudulently issued for their domain (servers). For example, imagine that we are the manager of servers of Politecnico di Torino. If a certificate is received everything is good and goes well. If an attacker is going to another CA and can create a certificate for Politecnico di Torino, is it possible to be informed of that? No, it is not possible. That was an error, person was not authorized by Politecnico, yet the attacker has a valid certificate. If client connect to the wrong server which contains a valid certificate, browser is not able to detect that. Browsers aren't good at detecting **rapidly** malicious websites if they receive (e.g., in a TLS connection):

- Mistakenly issued certificates
- Certificates issued by a compromised CA

If there is a certificate issued by mistake it takes time to detect the problem and it is needed to revoke all the certificates and make the browser aware of this problem. Until then, the browser is thinking to be connected to the real good server, so some attacks arise:

a. Fake server
b. MITM attacks

Mistakenly issued certificates: some examples

A. 2011: an intruder managed to issue itself a valid certificate for the domain $google.com$ and its subdomains from the prominent Dutch Certificate Authority DigiNotar. The certificate was issued in July 2011 but may have been used maliciously for weeks before the detection on August 28, 2011, because it was used for large-scale MITM attacks on multiple users in Iran. Again, there is the suspect of an operation of secret service, because if someone wants to spy on clients that use Google services, it is not easy. But if a fake website is created, then it is possible to have, e.g., the fake Gmail and use it to spy users by redirecting user traffic to the real google server and act as MITM.
B. 2011: the Comodo Group suffered from an attack which resulted in the issuance of nine fraudulent certificates for various domains owned by Google, Yahoo!, Skype and others.
C. July 2014: an unknown number of mis-issued certificates were issued by a sub-CA (one level down the root CA) of India CCA, the Indian NIC (*Network Information Center*). Due to the scope of the incident, in which tons of certificates were created, the sub-CA was wholly revoked, and India CCA for the future was constrained to a subset of India's top-level domains namespace. For example, they cannot issue anymore certificates for anything that ends with "*.in*" but only for specific domains.

HTTP Key Pinning (HPKP)

The owner of a website that wants to protect against the above problems, can try with HTTP Key Pinning. Since it is not possible to avoid some CA to be subverted, or some fraud to take place, it is possible to specify, when a client connects with the server, the digest of the public key of the server. Even if the certificate is sent, since there could be many fake certificates, the server will tell the client that this is the good one through the digest of the public key. The UA (User agent, browser) will put in the cache the key and refuses connecting to a site with a different key.

This method is a TOFU technique (Trust on First Use) so it works only if the first server to which the user connects is the real one, otherwise clients will accept only the fraudulent one. It is important to manage also key updates properly, usually including a backup key. URI are sent to the owner when clients connect with fake ones. Given all the problems with this solution, nowadays it is not so used, and *Certificate Transparency* approach is preferred.

Certificate Transparency (CT)

There is a whole organization of all the major providers that joined together to create this specification, which is available at https://certificate-transparency.org/

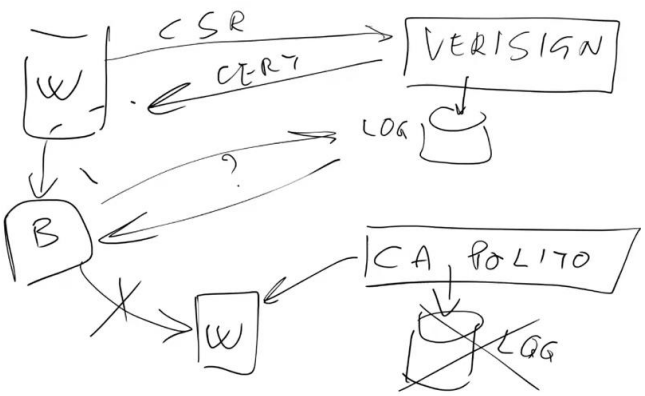
It is a group of people that want to create an open, global auditing and monitoring system where:

- Monitoring means the ability to verify, whenever is wanted, the current status of things
- Auditing means to be able, later, to reconstruct what has happened in a system

These operations should be based on public logs of issued certificates: all the CAs should create this log and permit public examination of them. This would enable domain owners to verify that no fraudulent certificates have been issued for their domain. It means that the owner of a website which receives a certificate that has been requested, can check if by accident there are other certificates issued in the name of the website owner even if that was not requested, because that would be a fraud. The domain owner should periodically perform this auditing operation to verify that there are no other unallowed certificates around. This was an experimental protocol originally proposed by Google and later standardized by the IETF *Public Notary Transparency* WG.

The purpose of this protocol is to **permit the issuance and the existence of TLS certificates** (only for web servers) open to analysis by domain owners, CAs and domain users. Anybody in the internet public should be able to consult these logs. This is the basis to enhance some other software, for example, the web clients (browsers) should only accept certificates that have been publicly logged, otherwise there could be a fake CA or something that circumvented the security measures of that specific CA. Vice versa, it should be impossible for a CA to issue a certificate without it being publicly visible to all (that is to put it in the Transparency Log). This is gradually being implemented.

This implies that every CA should implement CT, even companies. Let's make an example.



Verisign is a big CA which receives a CSR (Certification Request) and then gives back the certificate which is returned to the website. Then VeriSign has a log that is publicly accessible.

The web browser, in the moment that receive the certificate from the website it will check in the log to see if the certificate has been logged. It could be possible to create an own CA for internal use (e.g., the CA of Polito), and so, currently it is possible to create a certificate for own TLS machine and the browser can connect and everything goes well.

If in the future there will not be a log for that specific CA, then the browser could possibly not connect. This is imposing restriction not only on the official CAs but also on the private ones. The point is that the browser would not accept a certificate unless it has been properly logged. Creating a CA is becoming an increasingly complex matter which is not only a matter of properly issuing certificates and maintaining the revocation list but it must also comply with the CT.

Main ideas

The main idea is to make impossible or very difficult for a CA to issue a PKC for a domain without making it visible to the domain owner. It means that the internal procedures, both technical and manual, should always include the creation of an entry in the log, otherwise the certificate should not be created. Then, additionally, should be provided a protocol for open auditing and monitoring service so that anybody can determine if a certificate has been created by mistake or by fraud. Since it is not possible to reach each CA physically, a protocol is needed. Another main idea is to protect users from being provisioned (when a connection is established, and a fake certificate is received) by certificates that were miss-issued.

This system is complex and there are several actors that take part to CT in addition to the web server and web client: *Submitter, Loggers, Monitor, Auditors*.

CT – Log Servers

Every issued certificate is logged, so there will be around the world several log servers. Those are the core of the CT system and public entities. These servers maintain a **secure** log of TLS certificates. Security here is guaranteed by:

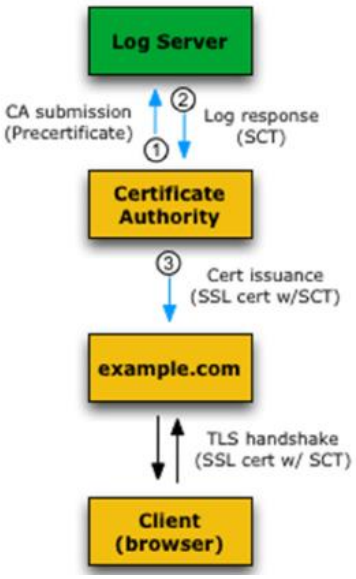
- **Append-only**: certificates can only be appended to the log. They cannot be deleted, modified, or even retroactively inserted into the log.
- **Cryptographically assured:** there is something like digital signature, but it is named *Merkle Tree Hashes*, which does not require a PK signature but provides anyway integrity and authentication. It is needed to prevent tampering and misbehavior.
- **Publicly auditable**: anyone can query a log using an HTTPS channel and verify that the log is *well behaved* (no modification, authenticity) or verify that the TLS certificate has been *legitimately appended* to the log.

CT-Operations

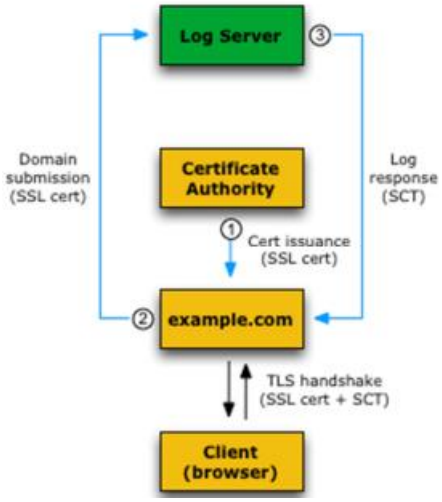
Anyone can submit a certificate to a log server, although most certificates will be submitted by CAs and by server operators. Potentially also those who got an internal CA could create an entry in a CT log. The logger provides each submitter with a **promise** to log the certificate within a certain amount of time, the **Signed certificate time-stamp (SCT)**. It is a kind of timestamping which is related to the signature of a certificate. The SCT goes along with the certificate for all its lifetime. Since SCT is not created by CAs, but directly from Log servers, it must be delivered somehow with the certificate.

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.



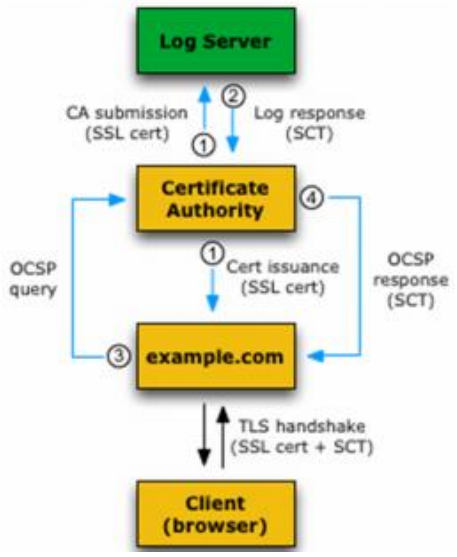
SCT via TLS extension



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.

CT – Submitters, Monitors and Auditors

Submitters are the ones the submit certificates (or partially completed ones) to the log servers and receive a SCT.

Certificate Monitors are actors not encountered yet and they are public or private services that watch/ control for misbehaving logs or suspicious certificates; they periodically scan information from log servers, inspect new entries, keep copies of the entire log (to avoid the possibility to lose it when deleted) and verify the consistency between published revisions of the log. This last mechanism is used to avoid the *substitution attack*: if there is a certificate log today, and another one of the previous weeks, since new entries can only be appended, the log should increase with all the previous data being present. If someone completely remove logs and replace it with fake ones, it is possible to detect it because of that copy of the log of previous week.

Certificate Auditors are lightweight software components that perform two functions:

a. **Verify overall integrity of logs**, because periodically fetch and verify the log proofs (where a log proof is a signed cryptographic hash that a log uses to prove it's in good standing, performed by Merkle tree Hashes)
b. **Verify that a particular certificate appears in a log**: the CT framework requires the presence into a log of all TLS certificates; if a certificate has not been registered in a log, it could be a fraud or the CT that is not implemented yet. In any case, a TLS client may (in the future might change into must**)** refuse to connect with them (for more security a client must refuse this kind of connection).

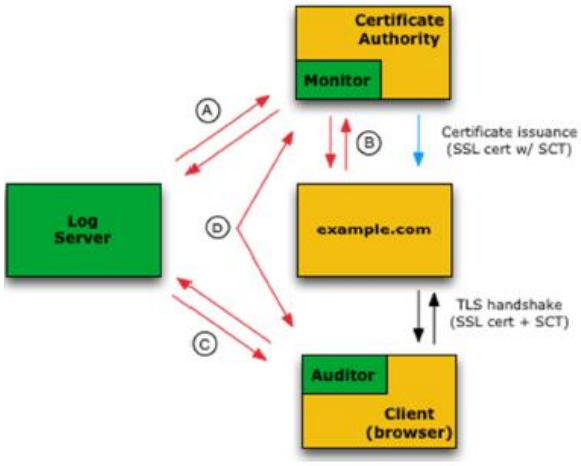
If a TLS client (via auditors) determines that a certificate is not present in a log, it can use the SCT from the log as evidence that the log has not behaved correctly and auditors and monitors exchange information about logs through a *gossip protocol*.

A possible CT system configuration

In the picture, it is possible to notice various actors: the CA, the domain, the client (browser) and the log server. The **monitor** is part of the CA while **auditor** is part of client.

In the step:

- Monitors watch logs for suspicious certs and verify that all logged certs are visible
- Certificate owners do not ask to log servers, but query monitors to verify that there are no illegitimate certs logged for their domain
- Auditors verify that logs are behaving properly and can also verify that a particular cert has been logged, usually the certs that the browser relates to.
- Monitors and auditors exchange info about logs to detect forked or branched logs, because can happen that a log server is attacked, and the logs have been modified/replaced.



Note that this is only an example, because the general certification of CT does not prescribe any particular configuration to follow.

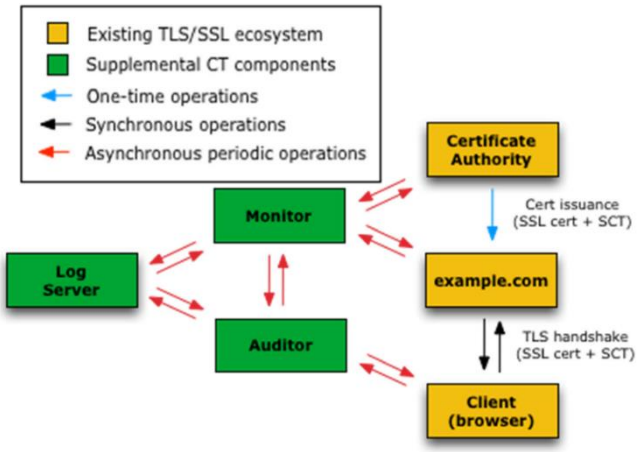
A possible configuration can be:

- The CA obtains an SCT from a log server and incorporates it into the TLS certificate using an X509v3 extension
- CA issues the certificate (with SCT attached) to the server
- During the TLS handshake, the TLS client receives the TLS certificate (and thus also the SCT)
- TLS client validates both certificate and the log signature on the SCT to verify that the SCT was issued by a valid log and that the SCT was issued for that certificate.
    - If discrepancies are found, the TLS client may reject a certificate.

Monitors can be operated by the CA while auditor should be built into the browser. Browsers periodically send a batch of SCTs to its integrated auditing component and ask whether the SCTs have been legitimately added to the log or not. Auditor asynchronously contact the logs and perform the verification.

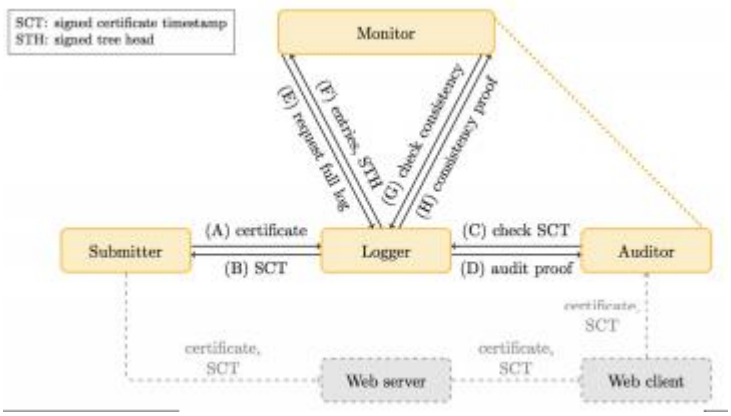
Monitors and auditors operate as standalone entities, providing paid or unpaid services both to CAs and server operators.

Another possible configuration can be the following:



Monitors and auditors are not part of the CA and the browser (as it was in the previous schema) but external entities. This would permit to minimize the environmental software change on CAs and web browsers, if not for including the SCT in the certificate (since it is an extension, it should be anyway supported). Now monitor and auditor talk with the log server and accept requests by all interested parties. The red arrows are asynchronous periodic operation (when needed/scheduled).

The following schema shows the different interactions between the various entities of the CT:



Submitter, that in this case can be a CA or a domain owner, submits the certificate to the **logger** and receives the SCT. The **auditor** will periodically contact the *logger* to check the SCT. The submitter provided the SCT to the web server and it provided to the web client that will contact the auditor to verify if the certificate is good, so that is how the auditor gets the SCT. The logger will answer with an audit proof.

Logger will interact also with the monitor, which will request the full log and the logger will answer with all the entries in the log + the signed tree hash. The monitor will check the consistency and the logger will return the consistency proof (remembering that monitors and auditors contact each other for other confirm).

CT – current log servers

Currently, which log servers are used depends on the browsers. There are 2-3 different engines for browsers, the widely used is Chromium and it uses 12 valid log servers on October 2020: *Google, CloudFlare, DigiCert, Certly, Izenpe, WoSign, Venafi, CNNIC, StartCom, Sectigo, Let's Encrypt, TrustAsia*. Some of them are CAs, but others are web providers. The log servers do not need to be strictly associated to CAs but they can be external independent entities.

ACME Protocol

Is it possible to make your own certification authority be automatically recognized by the browser? The answer is no, since it is needed to insert manually insert the root of the CA into the browser, if that is not already present. This implies lot of technical, legal and financial implications since most of the browsers vendors requires to pay to insert it.

To simplify it, there have been an international effort. Normally, creating a public certificate (certificate automatically recognized by browsers) is costly and variable (100-200 euro per year). Since cost prevents people to adopt on large scale TLS, there was an international effort led by the ISRG (*Internet Security Research Group*) to create a CA named "*Let's Encrypt*" free of charge and automatically recognized.

Let's Encrypt is the only CA which is creating valid certificates automatically recognized by browsers without making clients paying fees. The costs relative to this service are brought by donators.

The key point for being able to operate a free and costless CA was in automating as much as possible all the operations. This is defined in the **ACME protocol** (*Automated Certificate Management Environment*), defined in RFC-8555 in March 2019. It is a protocol for PKC management between EE and CA, previously adopted only for Let's Encrypt, but nowadays there are also many others CAs that are adopting it. It allows PKC to be automatically requested and issued, without human interaction, based on exchanging *JSON documents* over HTTPS connections.

To implement it, an **ACME agent** (*client*) installed on a web server is needed, to prove to the CA that the server controls a domain. Once the CA verifies it, it will permit the client to request and install certificates as many as needed.

Let's Encrypt gives out certificate for a short duration, since it is an automatic process, it would permit to reissue the certificate frequently. In this way the client can be programmed to perform certificates operations at fixed intervals. There's no need to manually generate individual PKCS#10 requests, no need to prove domain ownership, download and configure server certificates using ad-hoc procedures. Currently there are over 100 open-source ACME clients.

To exploit ACME the following steps must be followed:

- An account at the CA (Let's Encrypt) must be created.
- Domain validation
- Issuance of certificate
- Revocation of certificate

Account creation

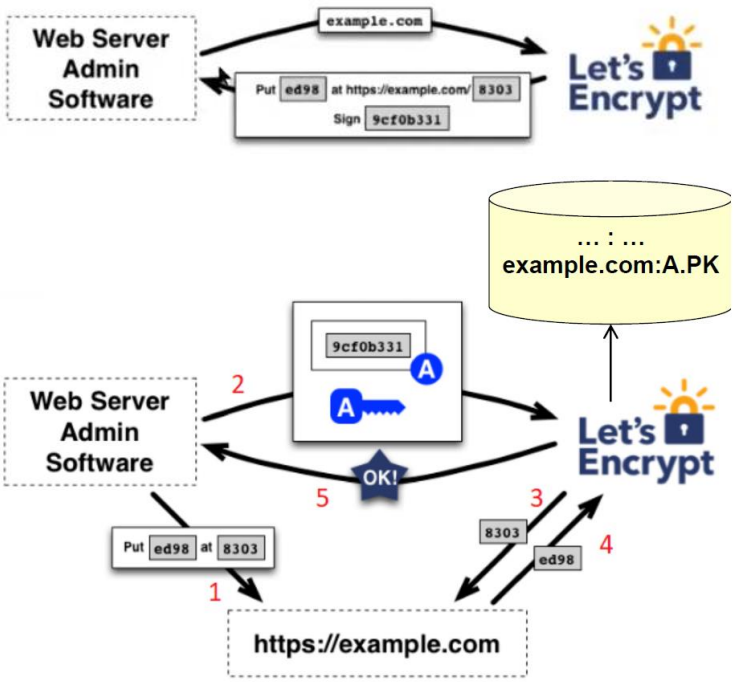
Account creation needs to be performed only once, and it must be performed before asking for issuance/revocation of domain certificate(s). Account creation is performed via the ACME client that **generates** a private/public **key pair** used for authorization purposes, named the "**authorized key pair**". The **authorized public key** is associated to the account registered at LE (Let's Encrypt). The **authorized private key** will be used to sign the certificate requests, while the authorized public key will be used by LE to validate the received certificate requests. A single account may be associated to one or more domains.

Domain Validation

It proceeds in this way:

- ACME client must **prove** to the CA **the control of the domain** for which it will request certificates
- ACME client **claims ownership of a particular domain**.
- The CA **challenges the client** to store a value at a specific path inside the domain (*step 1*) or in a DNS record and sign a nonce with the authorized private key → **proof of possession**.
- When the challenge is solved, the **client sends the signed nonce to the CA** to start the validation process (*step 2*).
- The CA downloads the response from the domain and **verifies its correctness** (*step 3 and 4*) and the signature's one
- If it is all correct, **CA approves domain ownership** (*step 5*), and the client is enabled to request certificates.

In the picture a web server at administrative software claims to be *example.com* and it receives a challenge from the CA to put a specific value in a specific file. It also requires making a signature over a nonce.

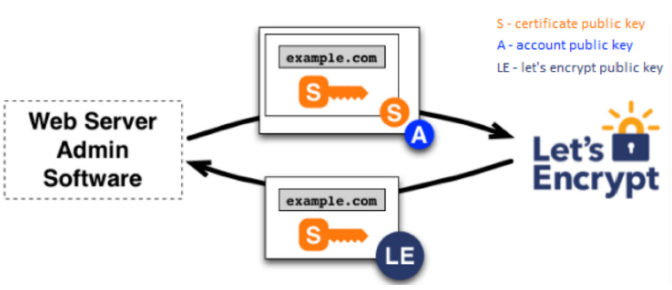


For domain validation, the *nonce signed with the private key + the attacked public key* is sent back to LE. Then it will consult the file with that specific number and check that it contains the value. Once the step 4 + verification of the signature are positive, then LE will approve the domain ownership. LE stores in its internal database the fact that the domain "*example.com*" is associated with the client A and the public key of the client for future certificate requests.

Certificate request and issuing

The client **uses the authorized key pair** to request, renew and revoke certificates for the domain and **constructs a PKCS#10 CSR**, and asks the CA to issue a certificate with a specified public key. The CSR includes a signature by the private key corresponding to the public key of the CSR, but the client also signs the whole CSR with the authorized private key for that domain. Be aware: when PKCS#10 was discussed, we noticed that it is a Proof of Possession, because it contains the public key that wants to be certified and a signature to demonstrate that the corresponding private key is possessed, and it is **self-consistent**. In this case, the PKCS#10 request is further externally signed with the authorized private key in order to show that this is a valid request for that domain.

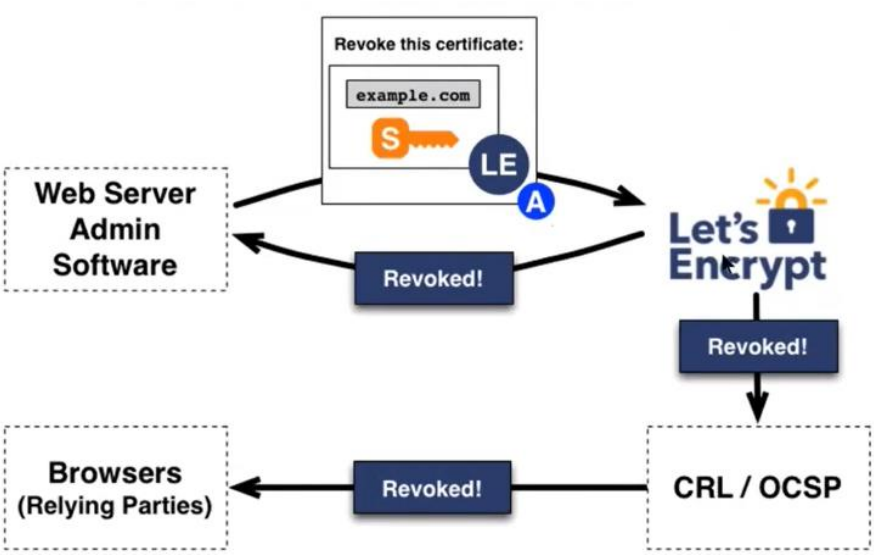
Looking the picture, the upper central box is the PKCS#10 which contains the request to certify *example.com* and it contains the public key. The letter S is the signature performed with the corresponding private key. All of this is signed with the private key corresponding to the account of LE. It will then verify that everything is ok and will return the certificate signed by the LE.



ACME certificate revocation

The steps are the following:

- Client **signs a revocation request** with the authorized private key for the corresponding domain
- The CA verifies that the key is authorized
- If so, it revokes the certificate and includes this information in the proper revocation mechanism (CRL and/or OCSP)



The schema of the certificate revocation shows the upper central box which is the revocation request which contains the certificate (domain + key and sign) and the whole request is signed with the authorized private key. LE will send a confirmation to the Web Server Administrative Software, but it will also publish it in its own mechanisms. When relying parties will contact the revocation services will be able to see that the certificate was revoked.

Raw and XML digital signature and encryption

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/or 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.

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 RFC**s (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

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*

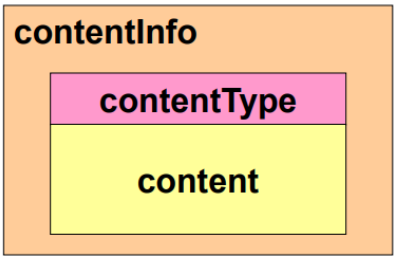
These algorithms were very basic and quite old, but with new RFCs the situation improved:

- 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

CMS: structure

At the highest level there is a container that has generic information about the content (*contentInfo*), then every block inside that has got a type (*contentType*) and the real *content*.



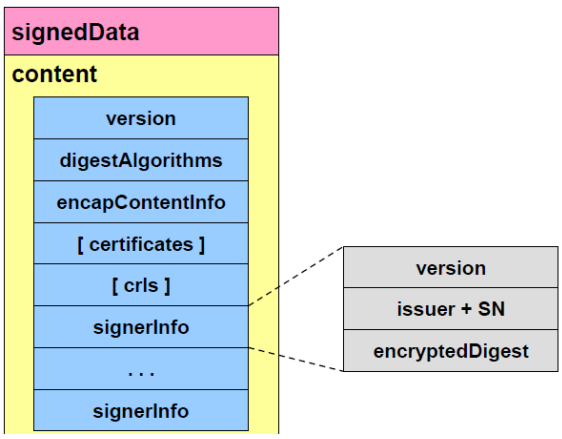
CMS: contentType

There are different *contentType* available:

- **Data**: it is an encoding (with ASN.1) of a generic sequence of bytes. This is the basis.
- **signedData: encoded data + parallel digital signatures**, so a single document can have more signatures, which are called **parallel**, because all signatures compute the hash of the document, so the order of signing is not relevant, and they sign only the document. It is needed every time simultaneously signature of a document from more people is required.
- **envelopedData:** it contains the **encrypted data** (with a suitable symmetric algorithms) and then it also contains the **key encrypted for recipient**(s). It means that to decrypt the content you must be one of the valid recipients because the PKCS#7 envelop also contains the key, which is encrypted with the public key so that only the valid recipients.
- **authenticatedData:** it contains the **data** with the **MAC**, and since MAC requires a shared key, but it is created by us, there is also the **key encrypted for recipient**(s). There is the assumption that the recipients have a public key, because that will be the way in which we will share with them the symmetric key used in the MAC.
- **DigestedData:** it is an unprotected format because it contains only the **data** plus the **digest.** It seems to be unprotected, but since this is a recursive format maybe then they will be used inside another format.
- **encryptedData:** contains the **encrypted data** with a **symmetric algorithm**, but it is assumed that the key has already been shared outside, so the key needed for decryption is **not contained**.

CMS: signedData

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



- **Version**: for example, PKCS#7 2.1
- **digestAlgorithms**: this is a **list** because, since the signed data can be signed by different people, maybe they can use different algorithms. For this reason, are reported here all the algorithms that will be used to sign the data by various signers (maybe only one if the algorithm is in common). The algorithms must be listed before the signature because then there is the data (*encapsulated content info*), so by reading data from the beginning the algorithms are first known, so while the content is being read it is possible to immediately compute those digests (in parallel) because it will be compared with the digest value present in the various

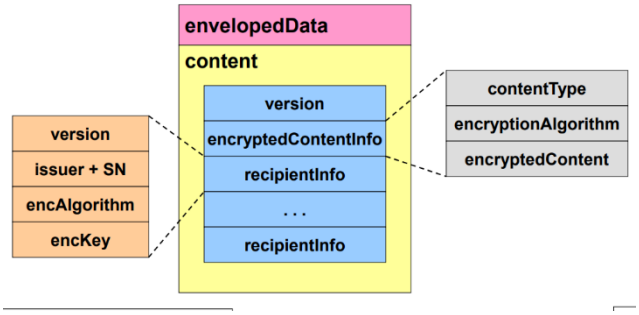
signatures, so this is for the **easy of processing** (this is permitting me to process the data in sequence without never go back).

- **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. This format can be used also as an *archival format* because you are storing the document, the signatures and everything that is needed to validate the signatures. The only thing that remains outside this envelop are the *root certificates* (that are never inserted because they are self-signed so you must trust them).
- **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.

CMS: envelopedData

When we talk about encryption the standard format is *envelopedData* (which is the contentType), its content has:

- **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*
- **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***.**



CMS is widely used for performing generic encryption and generic signature on raw data. It is also a valid signature format according to laws in different states, also including Italy. It is possible to find files which have the extension **p7m**, these files contain something that has been either signed or encrypted including the content, while it is get something of type **p7s**, this is a signature which is of type **detached.** In this case *encapsulatedContentInfo* does not contain the data, but contains a **pointer** to the data, that are in another place. This is useful, for example, when you want to sign something which is stored in a database, and you don't want to take it out of the database. This solution is flexible but increases some risks, because, for example, if someone is updating (in authorized way) the data, how is it possible to recompute the signature since there is only a pointer from the signature to the data?

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).

The standard provides definition for: the **signature format** but also the **procedures** to create and verify the signature. This is because XML is a text-based format and even if you think that text is the easiest thing to be brought from one system to another that is exactly the contrary: it is one of the most difficult objects to manage, so for this reason the procedures are also required. This is completely **independent of the transport** (HTTP, SMTP, FTP, …) and/or the storage technique. It is something for the intrinsic protection of data.

The first edition, named **1st recommendation**, is also specified in RFC-3275 while the most recent edition of version 1 can be found only on the *w3c website*, and it is dated April 2013. After that there is a major release (2nd edition) again available only on *w3c website*.

Xmldsig namespace

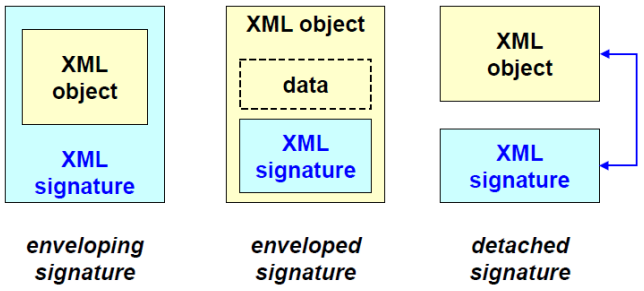
To identify the elements that are present in the standard, there is the need of specific identifiers. In XML terminology it is **namespace**. Specific identifiers are inside the following URL: http://www.w3.org/2000/09/xmldsig# . It is common, in order to avoid repeating the string, to define an **abbreviation**. The heading of an XML document in the picture, which always start with that header, contains the declaration of the document type (*DOCTYPE*) and it contains *XML Signature* following the document type definition and then it is locally defined the 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">
...
```

General characteristics of XMLdsig

In similar way to what we just discussed for PKCS#7, also for XMLdsig the signed object may be: **internal** (*with the signature*) or **external** (*it can only be an object referenced by a URI*). The **target** (object signed) can be **XML** (*native object*) or it can be **non XML** (*encapsulated object*) and in that sense it is possible to interpret also XMLdsig as a **row format** because it supports signature or anything which can be encapsulated. For encapsulation anything will be transformed in text. An important difference with PKCS#7 is that here we **attached meanings to the signatures** that are present with different semantics. An object can be **signed**, **co-signed** (parallel signature that have agreed to sign the same document), **witnessed** ("*yes I have taken part to this sign and I sign it not because I am involved but as an external witness that this thing happened*"), **notarized** (even stronger, a notary is not only an external witness but it has been authorized by the country in which it operates to act as a legal representative of the State). Note that XMLdsig uses keys that have been made available in some other ways and does not address key creation and certification. Keys are already created and made available in some way.

Signature types



- **Enveloping signature**: there is the XML signature which is *outside* and the XML object which is inside. It is the same as the PKCS#7 signed data, because in that case the external part was the PKCS#7 signed data while in the inside there was the blob being signed.
- **Enveloped signature**: since XML signature is specific for XML objects, then it is possible to create an XML document and reserve a place where to put the signature. This is not possible with PKCS#7 because it is not a generic data format, but only an envelope. This kind of thing, named *enveloped signature*, it is available for other format (e.g., PDF where there is the real data and a place for the signature). To create an enveloped signature, you must look to a specific application-level data format that has space inside them to put a signature. This is one of the most used things nowadays.
- **Detached signature**: there is the object in one place and the signature in another one. Like the PKCS#7 detached signature but, in that case, there was a problem because there was only one-way pointer but in XML there is a double pointer in both directions.

Canonicalization

XML is basically a text format, but text is not the same on all computer systems. We all use ASCII, but the line terminator might be different, but there is also the problem that in XML we are allowed to have *empty tags*, *line delimiters, hexadecimal values, order of the attributes, structure,* …Imagine that there is an XML object named "Person":

- $< person\ id = "12345"\ name = "Antonio"/>$
- $< person\ name = "Antonio"\ id\ = "12345"/>$
- $< person\ name = Antonio\ id\ = "12345"/></person >$

They are all the same data, in the second one the order just changes and in the last one the only difference is that the closing tag is made explicit. If these three things are the same, is the signature the same? Unfortunately, it is not because the hash will be different. Since we're signing the abstract data, we want the signature over these 3 objects to be the same and to be valid independent of the representation. Canonicalization algorithms are needed.

Canonicalization algorithms

Canonicalization algorithm means that before generating a signature you must put the data in a **specific order** and in a **specific format**. When data and signature is received, before verifying the signature you need to perform the same operation. The canonical representation means that we must create the physical representation of the XPath node set as an octet sequence. The algorithms change over the time, the default is **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.

Canonical XML

First, the canonicalization way is declared, then it will say:

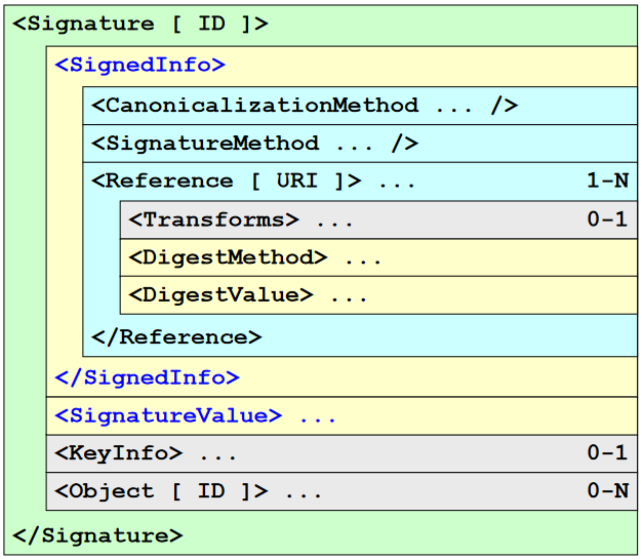
- Encode the document in UTF-8
- Normalize line breaks to #xA, before parsing
- Normalize attribute values, as if by a validating processor: it means that in some cases you are allowed to say that attribute $a = 123$ but the correct format might be $a = "123"$ (with quotes)

- Replace character and parsed entity references: substitute the, e.g., $\&x$ and substitute it with the extended format in any place
- Replace CDATA sections with their character content: CDATA sections contains literal data represented with, e.g., hexadecimal and they need to be replaced with their content
- Remove XML declaration and DTD (*Document Type Definition*) since they are meaningless
- Convert empty elements to start-end tag pairs: e.g., $< com >< /com >$
- **Normalize whitespace** outside of the document element and within start and end tags
- **Retain all whitespace in character content** (but characters removed during line feed normalization)
- Set attribute value delimiters to **quotation marks**
- **Replace special characters** in attribute values and character content by character references
- Remove superfluous namespace declarations from each element
- Add default attributes to each element
- Impose lexicographic order on the namespace declarations and attributes of each element

Now that the document is prepared to be in a canonical standard format, it is possible to proceed with the **structure** of an **XML signature**.

Structure of XML signature

The topmost external object is **signature object** with an *optional identifier* (needed in case there are multiple signatures on the same document). It contains *signedInfo* which is the information about the signature: *CanonicalizationMethod*, *SignatureMethod*, a *reference* to different things (discussed later) with a URI that can be an external pointer, then there are some *transformations* that can be applied to data before computing the hash, then there is the *digest* and the *value* of the digest. It is already present in the signature, but we have it also separately. In the *SignatureValue* there is the real signature and *0* or *1* is the information about the key (*KeyInfo*) since we're not obliged to put information about the key. In the same way there could be the object signed or not (0-N). With one signature it is possible to sign multiple objects.



SignedInfo

It is always the first element of a signature. It contains the *CanonicalizationMethod***,** which is a data normalization technique used before creating the signature. The *SignatureMethod* is the algorithm used for signature generation (e.g., RSA+SHA1). The *Reference [URI]* is the pointer to the actual data that have been signed. Note that if there is no URI in that case the pointer is an object inside the XML file and this permits to sign only **portions** of an XML because each object has an identifier, and the reference can be for object X. In this way it is possible to link the signature with objects (external with URI, internal with ID). For each reference it is possible to have additional information such as:

- *Transforms* (optional): data transformation before signature (e.g., selection via XPath)
- *DigestMethod* (compulsory): hash algorithm (e.g., SHA1)
- *DigestValue* (compulsory): hash value

The reference [URI] is the pointer to the element that has been signed. It **may be omitted** in the sense that can happen if the signed element is clear, given the context. For example, when an XML signature for an application-level message is used and the application specification specifies that the signature applies to the

whole application message. The **null URI** (e.g., URI="") is used to generate an **enveloped signature** which means that it is inside the object. The **self URI** (e.g., URI="#id") to generate an **enveloping signature** because now we are giving the identifier of the content that you have under you. The **external URI** (e.g., URI="http://...") is used to generate a **detached signature**.

Reference type

The reference may point to various element types:

- *Object*: a generic object
- *SignatureProperty*: it is possible to sign a signature attribute (e.g., date and time when object was created, HSM id). In that case an example can be: $Type = "\&dsig;SignatureProperties"$
- Manifest: it is the possibility to sign a **set of references** and in this case the type is: $Type = "\&dsig;Manifest"$

Manifest

It is a list of references **external** to SignedInfo but pointed to by SignedInfo. It means that the $< SignedInfo >< />$ tag will contain a reference to the $< Manifest >$ which is somewhere else inside the XML file. 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 contrary, any reference in the manifest is **not individually validated** since the procedure considers only the Manifest as a whole. So, 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. You are verifying that the hash is correct for the list and is not the duty of the XML signature to check that also the linked documents have the correct hash, it is an **application duty** to validate them and to decide what to do if one reference fails the validation. So, the signature is only my intention to sign all these things, but then it's up to you what to do if any of those elements fails the validation.

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.

```
<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>
```

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.

It is possible to understand that this gives *flexibility*, but it's also a **risk**, because if you blindly apply XML sign and you think "*okay, I have a digital signature, I am protected*", but if your implementation is not pointing directly to the objects, but is pointing through a manifest, maybe someone is changing any of those values (in the references) you will not detect it, because the automatic procedure of the XML sign is only checking if the SHA1 of the DigestValue field is actually the SHA1 of the manifest (manifest as a whole): we don't follow the links, it's up to the applications. So, since you are the application developer, it's up to you decide that if

the signature is a manifest-type one, then you have the duty to follow the links and to verify the integrity of each element referenced inside the manifest.

Transforms

The transforms may contain one or more $< Transform >$ 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 is very important because it permits to select which parts of the XML object are signed. So, depends on what you are signing, if everything is meaningful or if there are some variable parts that are meaningless, for example, let's imagine that we have a file and we are signing the whole file, including its access control part, which records when the file was last accessed: this information probably would change over time, but that is not meaningful, because we want to check that the content of the file has not changed. So, the auxiliary date and time that tell you when the last time it was accessed should be excluded by the computation, otherwise you should re-sign every time someone is accessing the data. So, through XPath it is possible to specify which parts of the object you want to use in the computation of the digest or not.

DigestMethod and DigestValue

Digest method represents the **cryptographic hash algorithm**. SHA1 is compulsory, but of course there are other algorithms that are also possible nowadays.

The algorithm must receive as input a *byte stream*, so you will first perform URI dereference, to get the object pointed by the URI, then the *Transform* is applied then is obtained:

- A set of XPath nodes. From that set of nodes, the byte stream is generated via Canonical XML.
- A byte stream and then its digest is directly computed if there is not XPath.

The digest value is the value generated *Base64 encoded* since everything must be a text, so it is not possible to include the binary result of a hash function, but it is needed to be transformed in text.

SignatureMethod

All the available signature methods can be found on *https://www.w3.org/TR/xmldsig-core2/#sec-SignatureMethod*.

Currently, the mandatory algorithms used to generate the signature are:

- For **digest**: *SHA-256* (preferred) and *SHA1* (discouraged, maintained only for backward compatibility)
- For **MAC**: *HMAC-SHA-256* (preferred) and *HMAC-SHA1* (discouraged, maintained only for backward compatibility)
    - With optional parameter the "*truncation length*", because in some cases an HMAC is computed and then truncated in order to get a shorter MAC
- For **digital signature**: *RSAwithSHA256* or *ECDSAwithSHA256* (better), while *DSAwithSHA1* is used only for **verification** (it means that if some documents were signed years ago with this system, it is possible to accept it and to verify the documents, but no new signature should be generated nowadays with that standard)

When there is the declaration of the signature method, again the pointer is used (the "http://..." can be abbreviated with "dsig", followed by the name of the method used, like "hmac-sha256" in that case). Then there is an optional parameter, if it is wanted to truncate the digest that would be of 256 bits, and it is specified in the *HMACOutputLength*, in order to truncate to 128 bits.

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

SignatureValue

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).

Security requirements

In the following list there are all the security requirements, which are out of the scope of XML signature:

- **Confidentiality** either it's a matter of transport by TLS, or XML Encryption is used but it is not part of XML signature.
- **Message authentication**: message integrity and creator authentication provided by MAC or digital signature, but there is no assurance about the sender since these things are often applied to data inside a protocol. Remember the difference between author of the data and sender of the data because it is possible to send something that was signed by someone else, and those are two different things. And in this case, it is author/creator, but not sender. It means that **sender is not authenticated**, that is something related to transmission.
- **Sender and receiver authentication**: this is provided by some external things, for example by TLS client authentication. But when it is used, there is no assurance about the message creator.

Nowadays there are several cases in which there is a TLS channel used to transfer XML data, and this is a comparison in some sense: if what is transmitted also contains the signature, then it is get the property of message authentication. If plain XML inside TLS is used, then it is get the property of sender/receiver authentication. To get both properties, an XML signed inside a TLS channel must be used.

To guarantee that the signer is also the sender, the same X.509 certificate should be used for both TLS client authentication and creation of the XML signature. This is, for example, done in the old standard *SOAP* when they are creating the digital signature (SOAP-DSIG) or creating a TLS channel (SOAP-TLS). In this case, the element pointing to the key (the $< ds{:}\, KeyInfo >$ element) can be omitted, because the key will be the same as the one used in the TLS channel.

Weaknesses

XML signature does not cover any kind of replay of filtering, which means that a malicious receiver may declare to have received the message multiple times, even if it was sent only once. So, while defining the XML format, it should be defined also a **nonce** (e.g., a timestamp) to avoid those fake claims.

A malicious receiver may forward the signed message to a third party, which may claim to have received it directly from the sender since there is **no indication of the destination**. So, if it is wanted, it is possible to include information about the **intended receiver** inside the signed message.

Recap of the countermeasures: add *timestamp* and *receiver identity* in the **signed body** of the message.

An example of detached signature is the one in the picture. The used algorithm for signature is *dsha-sha1* (by looking at *SignatureMethod* URL).

```
<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>
```

Reference is detached because it is a pointer to the web. So here it has been created something which has been signed and this "something" is stored in another place. **Transforms** will just perform in this case the *standard canonicalization*. It means that, when looking at the web page and its data, before computing the hash value the specified canonicalization process is applied. The *DigestMethod Algorithm* is SHA1 (because of previous dsa-sha1 declaration). The digest value in base64 is put in the *DigestValue* tag.

The *SignatureValue* itself is "*JK34s…*", but then the info about the signer is needed. This information is contained in the *KeyInfo*, which was not explained before since it is not part of the XML dsig standard, because that covers only the signature. There are other standards that are covering in XML the management of cryptographic keys, and this is a piece of that used in XML dsig, in which, *KeyInfo* is an object that declares the key being used, *x509Data* is a wrapper that will contain the certificate and *x509Certificate* will contain the base64 encoding of the certificate itself. There is no pointer, such as Issuer and SerialNumber, but the certificate itself is directly enclosed inside the signature.

```
<SignatureValue>JK34s...</SignatureValue>
 <KeyInfo>
   <X509Data>
     <X509Certificate>
     MII...AABvi
     </X509Certificate>
   </X509Data>
 </KeyInfo>
</Signature>
```

XML encryption

It is a W3C standard simpler than XML signature, but still important because confidentiality may be needed in some cases. So, again, it's a syntax to represent the encrypted data, but also a procedure to encrypt and decrypt data.

First version v1.0 was released in December 2002, while the second version (v1.1, April 2013) is the current one.

XMLenc: characteristics

The difference here in the flexibility that we have seen in the signature, is about what we are encrypting. There are different possibilities:

- It is possible to encrypt a **whole XML element** (starting from the initial tag until the end tag)
- It is possible to encrypt the **content of an XML** (so the tag is not encrypted but only the content inside the tag)
- It is possible to encrypt a **whole XML document** (which is bigger than a single element)
- It is possible to encrypt just a **portion**: *EncryptedData* or *EncryptedKey*, and this is named *super-encryption*

The result is an XML Encryption element which may contain or reference 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.

XMLenc: syntax

The XML encryption contains:

- **EncryptedType**: it is the abstract type and can be *EncryptedData* or *EncryptedKey*, according to what has been encrypted.
- **EncryptionMethod**: declaration of the used algorithm.
- **ds:KeyInfo**: it is used here to identify the key used for encryption
- **CipherData**: the actual encrypted data. It contains one of these two elements:
    - *CipherValue*: the encrypted content, encoded in base64. This means that everything which is encrypted in XML will become bigger. It is typically $+\frac{1}{3}$ of the initial dimension. If there is something that originally was 3MB, after performing XML encryption it becomes 4MB just for the CipherValue, since base64 has that property.
- *CipherReference*: a pointer to the data which is encrypted and which is external (not contained inside the XML encrypted object)
- **EncryptionProperties**: additional information about the generation of the *EncryptedType* (e.g., which is the original data that has been encrypted, for example, JPEG, WORD, or something else).
- **ID** (optional), just for reference in other parts.
- **Type** (optional): specification of the document type (element or content)
- MimeType (optional): used to declare, using MIME, what was the original content before encryption.

XMLenc: required algorithms

In v1.0 there were:

- **encryption** = *3DES-CBC* (for backward compatibility), but also *AES-128/256-CBC*
- **key transport** = *RSA-v1.5*, *RSA-OAEP* (RSA Optimal Encapsulation, preferred). It is used to give the encrypted thing to the recipient.
- **symmetric key wrap** = *3DES* (for backward compatibility, should never use it) / *AES-128* / *AES-256 KeyWrap*
- **digest** = *SHA1*, *SHA256* (recommended). The digest is not a signature but it's useful.
- **authentication** = *XMLdsig*

In v1.1 there is the addition of the GCM, so the ability to have authenticated encryption:

- **encryption** = *3DES-CBC*, *AES-128/256-CBC*, **AES128-GCM**
- **key derivation = ConcatKDF.** In the previous course there was the PKDF2 or HKDF, but unfortunately XMLenc is using another thing: ConcatKDF.
- **key transport / key agreement** = *RSA-OAEP / ECDH*
- **symmetric key wrap** = 3DES / AES-128 / AES-256 KeyWrap
- **digest** = *SHA1* (discouraged), *SHA256*

XMLEnc: examples

In the picture there is an XML document with several XML elements. The picture is related to information about payment. If follows the standard created by the organization specified in *PaymentInfo* tag. It contains an element for the *name* for the author of the payment and then information about the *credit card* used by this person for the payment (limit, currency, number, issuer, expiration). Those elements all together would permit the recipient to charge the proper amount, but this is sensitive information so it should be protected.

```
<?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>
```

Suppose to protect just a part of that, which means the encryption of a single element. The most important part could be the data of the credit card itself. So, the general information and the name of the author will remain in clear, but then the whole element *CreditCard* will be encrypted, as in the next picture.

The *CreditCard* element has been **substituted** by the *EncryptedData* element, which is saying that a whole "*#Element*" has been encrypted, following the specifications "*xmlenc#*", and the *CipherData* will contain the value which is the base64 representation of that information.

There is no algorithm specification: that is not compulsory, maybe the application has defined that a certain kind of algorithm is always mandatory, and the key agreement has been performed externally.

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://cards.org/payment'>
  <Name>Gianni Pautasso</Name>
  <EncryptedData
    Type='http://www.w3.org/2001/04/xmlenc#Element'
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>A2BB45CE6...</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

```
<CreditCard ...>
  <Number>4091 2450 ...</Number>
  <Issuer>MyBank</Issuer>
  <Expiration>06/10</Expiration>
</CreditCard>
```

Now, let's image to encrypt just a portion of that, for example the serial number of the credit card.

In this case, inside the element *Number* the content is replaced. In fact, there is the *EncryptedData* which tells that XML encryption is being used for the encryption of the "#Content" of an element. The *cipherData* in this case it is inside the *Number* element and *CipherValue* contains the base64 encoding of the encryption of just the number of the credit card. The other information will remain in clear.

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://cards.org/payment'>
  <Name>Gianni Pautasso</Name>
  <CreditCard Limit='1,500' Currency='EUR'>
  <Number> <EncryptedData
      Type='http://www.w3.org/2001/04/xmlenc#Content'
      xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <CipherData>
        <CipherValue>F3C55D77...</CipherValue>
      </CipherData>
      </EncryptedData> </Number>
    <Issuer>MyBank</Issuer>
    <Expiration>06/10</Expiration>
  </CreditCard>
</PaymentInfo>
```

```
4091 2450 0288 5657
```

It is possible to also encrypt the **whole object**. In this case there is a generic document in which it is told that everything is encrypted data. The *MimeType* is *txt/xml* (because we are encrypting an XML object). The standard is specified in the Type field URL (*xmlenc#*) and the *CipherData* contains the base64 encoding of the encryption of the **whole PaymentInfo**. This is interesting, because it's hiding to an external observer even what is the kind of thing/action that is being performed, because there is not even that declaration (*PaymentInfo*) anymore. Unless an attacker can understand from the endpoint (the receiver) that this was related to a payment action, it has no clue about the real action, and everything is completely hidden.

```
<?xml version='1.0'?>
<EncryptedData
  MimeType='txt/xml'
  Type='http://www.w3.org/2001/04/xmlenc#'>
  <CipherData>
    <CipherValue>37BBC923A4...</CipherValue>
  </CipherData>
</EncryptedData>
```

```
<PaymentInfo ...>
  <Name>Gianni Pautasso</Name>
  <CreditCard ...>
    <Number>4091 2450 ...</Number>
    <Issuer>MyBank</Issuer>
    <Expiration>06/10</Expiration>
  </CreditCard>
</PaymentInfo>
```

Let's now make **explicit the algorithm and the key being used**. It is still the encryption of the whole element, but now the algorithm to be used is not assumed to be already known to the parties, but it's specified inside the element.

```
<?xml version='1.0'?>
<EncryptedData
  MimeType='txt/xml'
  Type='http://www.w3.org/2001/04/xmlenc#'
  xmlns='http://www.w3.org/2001/04/xmlenc#'>
  <EncryptionMethod Algorithm=
    'http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:KeyName>Alice-BoB</ds:KeyName>
  </ds:KeyInfo>
  <CipherData>
    <CipherValue>37BBC923A4...</CipherValue>
  </CipherData>
</EncryptedData>
```

There is an "**EncryptionMethod Algorihm=**" element in which is declared the algorithm being used (*#tripledes-cbc*). But then a key must be specified. In this case, the standard from XMLdsig is used (KeyName) to give a **name** to this key ("*Alice-Bob*"), which is just a reference to something external, and now is just made explicit. It's just a name which is meaningful to the two parties. Then there will be, of course, the normal CipherValue. There could be many more cases: for example, it is possible to put inside the key which is encrypted with the public key of the recipient, and so on. All those parts are not in the encryption, but in the key specification of XML (*KeyManagement specification*, which is yet another standard with all the complexity it requires).

JOSE (JSON Signature and Encryption)

JSON is the predominant data format in web applications especially in mobile environments and IoT systems that don't have a native support for XML. 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, …).

There are various standards:

- 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)"

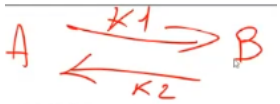
To become good in this section they must all be studied since they are all used (e.g., the algorithms are specified, and we will use them). Note that although there is a list of separate standards, globally they are known as **JOSE**. They are all considered as sub-chapters of JOSE.

JSON Web Signature (JWS)

JWS has been developed to sign arbitrary content using a compact JSON-based representation. Beware, as already happened with XML signature, **the signature term is used in an improper way** which it's not

recommended to use: JWS is a system in which it is possible to perform a **true digital signature,** but this term is also used to perform a MAC. Even tough MAC is providing authentication and integrity, normally the term "signature" is reserved only to things in which the roles are distinguished (the signer and the verifier). On the contrary, with MAC this is not possible since it uses a symmetric key (signer and verifier are interchangeable). If it is wanted to do something like a digital signature with MAC pay attention to which keys are being used, because to distinguish signer and verifier two different symmetric keys are needed.

The knowledge is shared: when A wants to improperly sign something with a symmetric key it should use K1, which is also known to B but at least it says that it is from A to B (but B could create the same message and claim it becomes from A).



When performing security analysis and someone tells "This document is signed with JOSE or JWS" you must challenge its knowledge about signature: is it digital signature or a MAC? Because the consequence is huge.

The representation of a signed object JWS is based on three parts: **header**, **payload,** and the **signature** itself. In JWS there is **compact serialization**: it tries to make the representation as short as possible. The parts are *base64url encoded* and *concatenated* separated by dot character as following:

$$header\_b64u\text{"."}\ payload\_b64u\text{"."}\ signature\_b64u$$

In this way everything is URL-safe (base64 is not URL-safe, while base64url is) and compact. It is done using only one signature (*signature_b64u*, placed at the end).

Base64 and Base64url encodings

The basic definition is in RFC-4648 in which there is not only the Base64 but also Base16 and Base32. Base64 encodes **6 bits at a time** using an 8 bits ASCII character, leading to a 33% of size increase (since 6 bits become 8). The allowed characters are the classic ones "*A-Z a-z 0-9 + /*" along with "=" which is used for **padding** since 8 is not a multiple of 6 (the total size must be a multiple of 6). This is **not** URL safe because of the characters "/=+" that are used in URLs. Since in JWS the various parts must be URL-safe, a particular version of *base64* called *base64url* is used. Base64url uses **"–" and "_"** (instead of "+" and "/") and **does not perform padding.** Rather than substituting the "=", it is simply deleted.

In the base64 example there is a JSON object with 2 fields which is put in output with an echo (without line feed since it is not needed). It is then passed into "*openssl enc*" command but it is not encryption since no algorithm is specified and "*-a*" means encoding in base64. The output is the one shown in the picture.

```
# base64
$ echo -n '{"family":"Lioy","given":"Antonio"}'
   | openssl enc -a
eyJmYW1pbHkiOiJMaW95IiwiZ2l2ZW4iOiJBbnRvbmlvIn0=

# base64url
$ echo -n '{"family":"Lioy","given":"Antonio"}'
  | openssl enc -a | tr -d = | tr "+/" "-_"
eyJmYW1pbHkiOiJMaW95IiwiZ2l2ZW4iOiJBbnRvbmlvIn0
```

Then the same is string is passed to openssl and the equal is deleted, then the "+/" is translated into "-_" to obtain the *base64url* encoding

JWS header parameter

It is composed of various fields:

- "*alg*" = signature algorithm, it's required otherwise it wouldn't be a JWS.
- "*jku*" = JSON Web Key URL, the key is not placed here but available in some other place
- "*x5t*" = base64url-encoded SHA-1 thumbprint of the DER encoding of the X.509 signature certificate in case a real digital signature with a private and public key has been used.
- "*x5t#S256*" = as x5t but the SHA-256 thumbprint is given.
- "*x5c*" = base64-encoded DER encoding of the signature certificate or JSON array if its certificate chain is provided (always excluding the root because the root is self-signed)

- "*x5u*" = X.509 URL to get the *PEM-encoded* signature certificate or chain provided in order from the end entity and up to the root (which is not there).
- "*kid*" = key identifier, if the two parties have agreed about sharing a key then the key is here.
- "*typ*" = type for signed content

Note that URLs must use an integrity-protected channel with server authentication such as TLS which means that HTTPS must be used.

JSON Web Algorithm (JWA): identifiers for JWS

The compact algorithm ("alg" in the header field) identifiers are:

- (req) "none" - no signature
- (req) "HS256" - HMAC SHA-256
- "RS256" - RSASSA-PKCS-v1_5 using SHA-256, this is recommended
- "ES256" - ECDSA using curve P-256 and SHA-256, this is strongly 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 something better than the required ones.

JWS header and payload example

Note that there's a CRLF on every line but not in the last one, after the closed bracket. The specified header is a JWT (Java Web Token) signed with HMAC SHA-256. Then the example shows the base64url encode of the header.

```
# plain header
# (beware! CR-LF at each EOL but the last one)
{
"typ":"JWT",
"alg":"HS256"
}
```

```
# base64url encoded header
ew0KInR5cCI6IkpXVCIsDQoiYWxnIjoiSFMyNTYiDQp9
```

The payload shows again a JWT, issued by Polito, with the specified expiration date and another custom field which is the content of the token. For the object representing the Polito as this professor has got the value true.

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

The base64url encode is written on two lines for readability, but it is only one very long line.

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

Generation of the JWS signature

The signature covers **both** the header and the payload, otherwise if the signature is only covering the content, the header could be changed and, for example, an attacker could change the signature algorithm or the content type, raising problems in the verification of the signature or in the interpretation of the content.

The signing input is the concatenation of the encoded header and payload separated by period, as we seen. This enables direct signing of the output representation (which means what will be actually send across the network).

The generated signature is then base64-encoded and appended to the signing input after a period. An example of signed input is the one shown in the picture on the right. Again, the line is broken for readability.

```
# signing input (w/ line break for readability)
ew0KInR5cCI6IkpXVCIsDQoiYWxnIjoiSFMyNTYiDQp9
.
ew0KImlzcyI6InBvbGl0by5pdCIsDQoiZXhwIjoiMTYwOTQ1OTE5OSIs
DQoiaHR0cDovL3BvbGl0by5pdC9pc19wcm9mZXNzb3IiOnRydWUNCn0
```

JSON Web Key (JWK)

To perform the signature, keys are needed. We have seen that keys are pointed by, but they were not in the example (it was assumed that the two parties already know which key is being used). To make it explicit JWK is used. JWK is the representation of asymmetric or symmetric keys.

The "*kty*" field specifies the type of key which can be: *EC* (strongly recommended), *RSA* (required), oct (octal representation of the keys, still required) or OKP. It is possible to have the "*kid*" key identifier field, which is by name.

There are various parameters depending on the type of the key and algorithm, for example:

- For RSA: "n" (modulus) and "e" (exponent)
- For AES: "k" (key value)

JWK examples

The first example shows a 256-bit symmetric key. The key type is **octal** and after that there's the base64url representation of the key.

The second example is a 2048-bit RSA public key as specified in the key type. After the key type there is the separated encoding of the modulus and the exponent. The final dots at the modulus means just that Lioy didn't represent the whole key. They *key identifier* ("*kid*") is an easier way to refer to that key in the rest of the token and signature.

```
# 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"
}
```

JSON Web Encryption (JWE)

JOSE also supports encryption. The JWE permits the encryption of arbitrary content using JSON representation. There are three parts: *header*, *encrypted key* and *chipertext*. There are two possible representations: native **JWE compact serialization** and **JWE JSON serialization**. Note that the key is encrypted too, so two algorithms need to be specified in the header, the one for the key and the other for ciphertext.

JWE header parameters

The header parameters are:

- "*alg*" = encryption algorithm for the key
- "*enc*" = encryption algorithm for the content

Then there are various key identification parameters (as in JWS):

- "*jku*" = JSON Web Key URL
- "*x5t*" = SHA-1 thumbprint of the X.509 certificate
- "*x5t#S256*" = SHA-256 thumbprint of the X.509 certificate
- "*x5c*"= X.509 certificate
- "*x5u*" = X.509 certificate URL
- "*kid*" = key identifier

JWE compact serialization

There are two options for the serialization. In this case (*JWE compact serialization*) it is like what has been done for JWS, as following:

$$header\_b64u \text{ "." } encrypted\_CEK\_b64u \text{ "." } iv\_b64u \text{ "." } ciphertext\_b64u \text{ "." } authN\_tag\_b64u$$

There is first the header encoded in base64url, then the separator, then the encrypted key, then the initialization value encoded in base64url (if an initialization value is needed) and it is the case of CBC, CTR or other symmetric algorithms such as chacha20. Then there is another dot, the ciphertext itself and finally, if authenticated encryption has been used, then there will be an authentication tag.

- It is a compact and URL-safe representation.
- 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 **no unprotected headers** and **no associated data** (so the tag is computed on everything).

JWE JSON serialization

In this case the encrypted content is represented as a JSON object. There are various top-level members that are all optional but the ciphertext which is the only one that is compulsory:

- "*protected*" = contains integrity-protected headers (contains list of headers protected with authentication)
- "*unprotected*" = contains the headers that must not be computed in authentication part
- "*iv*" = initialization vector
- "aad" = *additional authenticated data* if you want to have other data computed in the authentication part, which are not part of the headers but part of the content. In this case the JSON may have content split in two part: the real encrypted content and the additional data
- "*ciphertext*" = the actual encrypted content (compulsory)
- "*tag*" = authentication tag
- "*recipients*" = in this case, with JSON serialization, there is the possibility to have multiple recipients represented as an array of JSON objects that identify each recipient and provide the content encryption key, encrypted with her public key. For each of them there is
    - *Header {alg, kid}*
    - *Encrypted key*

This is more application-oriented when there is a multitude of consumers of JSON messages.

The example shows the protected header, the unprotected one (in which is possible to identify the key). Then the "*recipients"* field shows the header which contains the algorithm and the key identifier and then the encrypted key encrypted with the public key specified in the identifier.

```
{"enc":"A128CBC-HS256"}
{
"protected":"eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
"unprotected":{"jku":"https://srv.example.it/keys.jwks"},
"recipients":[
   { "header":{"alg":"RSA1_5","kid":"2011-04-29"},
     "encrypted_key":"UGhIOguC7IuEvf_NPVaXsGMoL..." },
   { "header": {"alg":"A128KW","kid":"7"},
     "encrypted_key":"6KB707dM9YTIgHtLvtgf9lociz..." } ],
"iv":"AxY8DCtDaGlsbGljb3RoZQ",
"ciphertext":"KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqC...",
"tag":"Mz-VPPyU4RlcuYv1IwIvzw"
}
iv = 03 16 3c 0c 2b 43 68 69 6c 6c 69 63 6f 74 68 65
```

In the second recipient there is the "a128kw" algorithm which means AES-128-KW (Key Wrap) in which the symmetric key is encrypted with another symmetric key. In this case there is some previous agreement about a key identified by the number 7 and then there is the encrypted key.

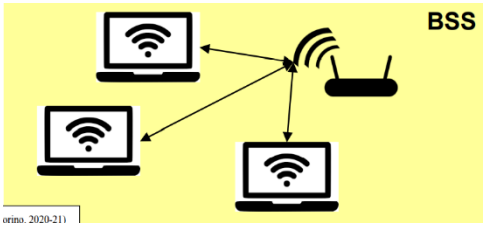
Then there is an initialization vector, the ciphertext itself and the authentication tag. Note that the used algorithm is not specified, since **it is not compulsory.**

In the example it is assumed that the encryption algorithm (declared outside) was "A128CBC-HS256" (AES 128 CBC with HMAC SHA256 for the authentication tag). Note that it is not needed to use authenticated encryption, because this is *encryption and authentication* (the algorithm specified in the "enc" field outside). The format, although named encryption, is actually supporting both encryption and MAC. It could be possible to use the same format specifying null for encryption and then just using the authentication part (but for that JWS is preferred). This is done to avoid using encryption and then to encapsulate again inside a signature (if they are both needed the standard is this one).

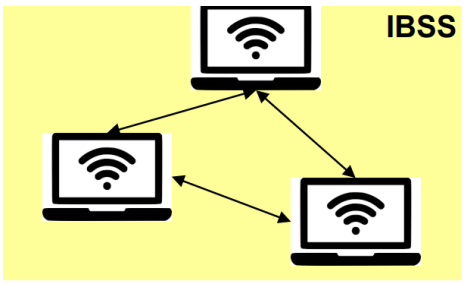
Security of 802.11 wireless networks

802.11 in brief

Some terminology:



- **AP**: **access point**. It is the point connecting the wireless part with the wired one.
- **STA**: **wireless STAtion** connected to the network through the wireless
- In case working in infrastructure mode
  - **Basic Service Set** (BSS) = **N x STA + 1 x AP** – so n stations connected to one access point
  - **1 Distribution System** (DS) = network of **N x BSS** (backbone) – putting together various BSS connected through backbone will get 1 DS
  - **Extended Service Set** (ESS) = **1 x DS + N x BSS** – is the set of the DS interconnecting the BSS and the BSS itself.



- The AP may periodically transmit a **beacon** that contain the identifier of the wireless network SSID and the MAC header, *beacon interval*, *date rate*, *timestamp*, etc. Those are all management information which are providing already some information to a potential attacker.
- **Ad-hoc mode** – in case there's no AP we will talk about **Independent Basic Service Set** (IBSS) = *N x STA*, which is a mesh created by stations that are talking directly to each other

WEP (Wired Equivalent Privacy)

It is an integral part of the standard *IEEE 802.11*. Unfortunately, it did not achieve his target. The target was to **make the wi-fi network as secure as the wired one** (a wired network is insecure itself). They assumed that they do not need strong security mechanisms. The services provided are:

- **Network access control** which is based on **authentication** – access control is typically authorization but having authorization without authentication is meaningless. So, with WEP the first step is to authenticate to join a BSS or IBSS.
- **Message confidentiality and integrity** – we are always talking of P2P (point to point) confidentiality and integrity because the communication is between one station and one AP. So, this is the protected part, but then the AP goes on the backbone and those messages are not protected, so it is only limited to the wireless part.

WEP in brief

Main problems of WEP:

- **STA authentication**
  - *None* = Open System Authentication (it is not an authentication, and it is not named like that).
  - *Shared-key* = symmetric challenge-response authentication
- **Confidentiality via RC4** – because it is fast
- **Integrity via CRC-32** – should not use it, it is a MAC or hash which is 32 bits long, so performing 2^{32} will take a short time.
- The initial version (1999) was limited to **64-bit key length** due to US export regulation. Later it was expanded to **128-bit** and finally to **256-bit**. Nowadays 128-bit is the most common version.
- Various weaknesses have been found and in 2004 it was deprecated, in favor of WPA/WPA2. But it is still used in several systems.

WEP – Open Systems Authentication

The first procedure is to **do not perform authentication** for joining the network

- **Probe request** and **probe response**: two messages of the wireless network
- STA → AP: the station is sending to the AP a request for open system authentication and self-declaring its own identity (*STA identity*)
- AP → STA: the access point sends to the station the open systems authentication response
- At the end there is an association request and response

But after that, transmissions are **protected for confidentiality and integrity with a shared-key**, and this is bad. Clients without a proper key will be unable to transmit and receive data. There is no authentication, but it is assumed that the key is well-known and that is what happens several times when there is an association to a network, but it is required to associate a key. The point is that **the key is shared between all the elements** which means that: if we have STA1 and STA 2 connecting to the same AP the key is the same for both. With open systems authentication the same key is used in all the nodes, and anybody can sniff the traffic if someone get the key.

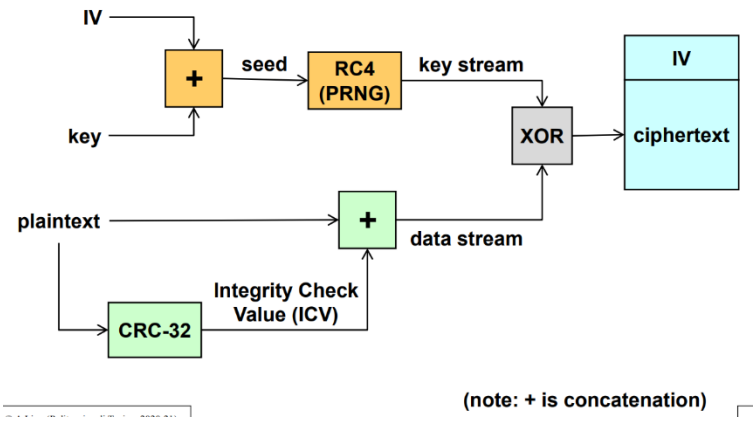
WEP – Shared Key Authentication

With shared key authentication procedure before joining the network (so before the association) the **station must authenticate itself as a valid one** by proving possession of the shared key. It used a **symmetric challenge-response** protocol:

- **probe request** and **response:** wireless part
- STA → AP: authentication request that contains STA identity
- AP → STA: the access point sends a challenge **C** which is **128 bit** long
- STA → AP: authentication response *R = enc(K, C)*, is the encryption of the challenge with the pre-shared key
- AP → STA: the access point will be able to perform the same computation and declare **authentication failure or success**
- If success we will be able to exchange the messages for associating the wireless node.

WEP – confidentiality and integrity

Traffic is protected with confidentiality and integrity. The algorithm is the following.



There is the **plaintext** (the payload of the packet) that we are exchanging over the network. The **integrity** and in part **authentication** are computed via **CRC-32** with an *Integrity Check Value* (ICV) which is **32-bit** long. This is put at the end of the plaintext, so the data stream to be encrypted is the payload concatenated with the CRC-32. RC4 is basically using a **XOR for encryption** but the part of RC4 is before, it is in generating the key-stream where the IV and key are concatenated. This operation provides the seed and then there is the **kernel,** which is the core of the RC4 algorithm that is just a pseudo-random number generation. Finally, the bit-to-bit xoring of the two streams is performed and it will provide the **ciphertext** that will be placed inside the packet with the IV put before it.

Note: in computation of the integrity value there is **no key**, the hash is computed and then it is encrypted together with the payload, so the protection of the hash is not independent of the encryption but is performed in the same way.

WEP – key types

There are two types of keys:

- **Default key**: it is also named *shared*/*group*/*multicast*/*broadcast* key or simply key. One single key is shared by all stations and the AP. This is not a pairwise key (peer-to-peer key) but a group key because it is shared between all computers. This means that any station (STA) can sniff all the traffic in the Basic Service Set (BSS). What is worst is that a station can act as another one, or it can take the role of the Access Point (AP), since it is shared and symmetric. Once the key of a BSS is known, then it is possible to **sniff all the traffic**, **claim to be another node** in the network (since it is possible to change the MAC to match that) or it is possible to even **jam the AP** and answer in another frequency to act as an AP.
- **Key mapping key**: it is also named *individual/per-station/unique* key. In this case it is a shared key between the AP and one specific station. In this case there will be a **real pairwise keys** between the access point and the various stations. The problem is that the AP must keep in memory one key for each authorized STA and that is a **management problem** because the AP needs to distribute those keys and to keep the list inside. This type of key is **more secure but more difficult** since it is needed to address the problem of key distribution between all the various access points if the user can move from an access point to another and how to give the correct key to the various stations.

Although there is a more secure alternative, due to management problems normally it is used the **default key** solution.

WEP Problems

Let's now look at the problems that are generated by the way in which integrity and confidentiality are obtained.

WEP problems – the IV

The IV (initialization vector) used to produce the RC4 keystream is **only 24-bit long**. A short IV field means that the same RC4 keystream will be used to encrypt different texts, even when using long keys: because of the *birthday paradox*. The shared key remains unchanged, since when a common key is used, it is rarely taken the burden to say to everyone that the keys changed. That means that the key stream has **52% reuse probability after 5000 packets** have been exchanged in one BSS (Basic Service Set), and 95% after 10000 packets. A busy AP sends about 1000 packets/s, which means that it happens after a few minutes.

This is **independent of the key size** because it is a problem in the IV. Since the IV is only 24 bits, the key size is typically 40 (24+40=64bits) or 104 (24+104=128bits). The fact that the key is composed by the real key and the IV, this IV must be different for each packet, but after 2^{24} it will be again reused. But, given the birthday paradox, if we are looking for all possible pairs it means that after 5k or 10k we will have the same IV.

Statistical attacks can be used to recover the plaintext due to **IV collision** and this is named the "**two-time pad attack**".

WEP problems – IV and the "two-time pad"

The point is that you should **never encrypt two messages with the same keystream KS**. Performing the following operations:

- C1 = $P1\ xor\ KS$ – the first plaintext is xored with the keystream and get C1
- C2 = $P2\ xor\ KS$ – then the second one
- $C1\ xor\ C2\ =\ P1\ xor\ KS\ xor\ P2\ xor\ KS\ =\ P1\ xor\ P2$ – when the same value is xored twice it disappears, so the xor of the two ciphertexts is generating the xor of the two plaintexts. But if an attacker is part of the network maybe it could send one of the two plaintexts and so by xoring with that message it is possible to get the other plaintext.

Moreover, collisions are very easy to find for several reasons:

- *The IV is reset to zero every time the AP is reset*
- *IV is incremented by one at each packet*
- *The same key is used in both directions*
- *The use of default key by all STAs*

WEP problems – "decryption dictionary"

This is another type of attack. If the attacker knows the plaintext P and the ciphertext C, e.g., because the attacker sent a ping (the content of the ping is well known, and it is known also the expected answer), the attacker can discover the corresponding keystream:

- $C = P \; xor \; KS$
- $C \; xor \; P = P \; xor \; KS \; xor \; P = KS$ – because P xor P is cancelled.

So, the keystream can be retrieved by xoring one plaintext with the corresponding ciphertext. The keystream is generated by RC4 as following: $KS = RC4(K, IV)$. The attacker can construct a dictionary of pairs $\{IV : KS\}$, of course assuming that K is not changed. This shows the possible defenses: change the key very often, not once a week or once a month, because this kind of attack can be performed in few minutes.

When a packet with an IV present in the table is intercepted, then P can be easily recovered using the corresponding KS, $P = C \; xor \; KS$.

Note that in this case the attacker does not know the key K, since it is not assumed that it is part of the BSS, the attacker simply accumulates packets.

The size of the dictionary is at most 24 GB = 1500 x 2^{24} B, which is the size of one packet multiplied by the number of possible IVs. By considering different keys there will be 24 GB for each key.

WEP problems – the CRC

The *CRC-32 checksum* is weak because it can be easily manipulated to produce a valid *integrity check value* (ICV) for a fake message. CRC-32 properties are the following:

- It is **independent of key K and initialization value IV**. It is computed on the payload before encryption
- It is a **linear operation**, i.e., $crc(A \; xor \; B) = crc(A) \; xor \; crc(B)$
- Note that also RC4 is a linear because basically it is just a XOR. It means that if there is a key-stream $KS = RC4(K, IV)$ and a ciphertext $C = KS \; xor \; M$ (message), then $C \; xor \; D = KS \; xor \; (M \; xor \; D)$.

Now the attacker wants the receiver to accept a fake packet $F = M \; xor \; D$ (where D is some difference that he wants to induce in M, e.g., in the amount of money to be transferred).

- The attacker computes $C' = C \; xor \; < D, crc(D) >$
- Then transmits (IV, C') to the receiver
- The receiver decrypts the modified packet to find
  - $< M', c' > = C' \; xor \; KS(IV, K)$
  - $= C \; xor \; < D, crc(D) > \; xor \; KS(IV, K)$
  - $= < M, crc(M) > \; xor \; < D, crc(D) >$
  - $= < M \; xor \; D, crc(M) xor \; crc(D) >$
  - $= < M \; xor \; D, crc(M \; xor \; D) >$
  - $= < F, crc(F) >$
- Check that $C' = crc(M')$, which it does as they're both $crc(F)$
- **So, the receiver accepts the message F as authentic!**

WEP problems – message injection

The attacker in this case can create a **completely fake message** with **message injection**. If the attacker knows (IV, C) and the corresponding plaintext P, then can compute key-stream $KS(IV, K) = C\ xor\ < P, crc(P) >$. To inject a fake message F (which is not the modification of an existing one) the attacker:

- Computes $C' = KS\ xor\ < F, crc(F) >$
- Then transmits (IV, C') to the receiver which will accept it as good. Note that here the attacker is using the same key-stream with the same IV, because there is no check that the IV is not reused. On the contrary, it will be reused, because if any check on the reuse of the IV is put, then the network will not work because after 2^{24} it would be needed to reset the whole network.

This attack works well even by using a better integrity function, SHA1 or SHA-512, because it exploits the fact that the ICV does **NOT** depend on any key but only upon the data itself.

WEP problems – fake authentication

The message injection attack can be used also to perform an authentication without knowing the key, not the open system authentication but the *shared key authentication*, which has been designed as a challenge response protocol which means that if you don't know the key then you will not be able to respond to the challenge. But let's observe the authentication process:

- (**) [AP > STA] $R = challenge$ (in clear) - The access point is sending the challenge R in clear
- (**) [STA > AP] $response = (IV, C),\ where\ C = KS(IV, K)\ xor\ < R, crc(R) >$. The station is sending to the AP the response, which is composed by the IV and the ciphertext, which is the keystream computed with that IV and the key xored with the random number used as challenge and the CRC of the same number. Since R is sent in clear it is possible to compute $crc(R)$.
- The attacker knows the KS for a certain IV (as in the message injection attack)
- This part (**) is done by observing another station. The attacker waits that the honest part authenticates and gets the keystream used for his response. Then, the attacker tries to authenticate:
    - [AP > attacker] $Z = challenge$ – the AP will send to the attacker a different challenge
    - The attacker can't reuse the honest answer, but it computes
      $$C' = KS(IV, K)\ xor\ < Z, crc(Z) >$$
      which is the keystream used in the honest authentication with another packet. The only point is that the attacker needs to use the same IV of the honest authentications. Thank to fact that the IV is in clear before the encrypted text the attacker got it.
    - [attacker > AP] $response = (IV, C')$
    - **The attacker is authenticated!** Even if he doesn't know the key.

Even the kind of authentication that should be more secure (the challenge-response one) is not secure at all. WEP has got many problems that are affecting *confidentiality*, *integrity*, *modification of the message*, *injection of fake message and even fake authentication*.

WEP problem – message decryption

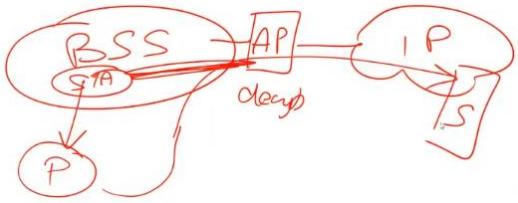
The ability to modify and inject packets also permits the attacker to decrypt existing packets.

The attacker cannot decrypt the packets itself, since he doesn't know the shared key K, and RC4 is a strong cipher. But the AP (Access point) knows the key K. So, it is possible to trick the AP in decrypting the packet and telling the attacker the result. This is basically the fact that RC4 is a XOR operation, and even if is strong, XORing twice a string means that it returns to the original version.

It is possible to perform this attack in few different ways:

- **Double-encryption**: send to the AP an encrypted packet using the *injection attack*. The content of this packet will be the content of the target packet that the attacker wants to decrypt. The AP will send back the response, encrypting what is sent, which means that it is being decrypted. Definitely, a second encryption with RC4 is equivalent to a decryption.

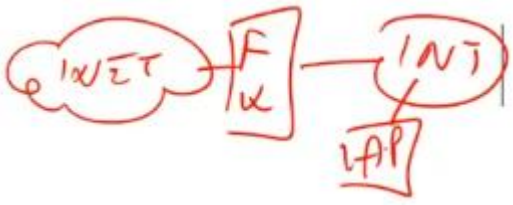
- **IP redirection**: packets are sent to the AP for an attacker-controlled external node. There is the BSS, with an AP, connected with a wired IP network. One packet is being captured, and the attacker wants to decrypt it. Thanks to the fact that the protection is only between the station and the AP, if that packet is taken and sent from the attacker to an external destination, the AP will perform the decryption and then will send it to the destination, which will be a server that the attacker controls, to see the message in clear.



- **Reaction attacks**: learn one bit at a time, depending if the resulting ack is good or not. It is more complex and takes more time.

WEP – countermeasures

Don't assume Wi-Fi with WEP is secure: it's NOT! You should **treat your wireless network protected with WEP as a public network**. That means that the AP must be located **outside** the firewall and NOT inside the network, for example **on the DMZ** (Demilitarized-zone), that is a good place. Most companies do this mistake, setting AP on intranet, but with WEP, it is insecure. The AP has to be placed between internet and the firewall, since it is something completely insecure as it is internet.



For any kind of traffic originated from the wireless network, which must enter the intranet, then other kind of protections are needed (upper layer) such as *VPN, IPsec, SSH, TLS*... Since there is the firewall, it can block if it is seeing coming a packet coming from the AP which is not protected with one of the previous security mechanisms, then it is forbidden.

The problem is not just in the shared key, but in many other places: short IV, the CRC32 computed only on data and not on the key, … Better key management does not solve the problem.

There are 2 solutions that must be adopted both:

- a **long IV**, 32 bits it's too short and it leads to the need of reuse of keystream. It must be so long that it never repeats for the lifetime of the shared secret (and not duplicated across machines sharing the same secret).
- a **strong MAC** instead of CRC, which must **depend** on K and IV and not only the basic data.

802.11i

It is a new standard to address the insecurity of WEP. This is defining what is called a **Robust Security Network** (*RSN*) t**hough specific association** (*RSNA*) between STAs and, additionally, it has improved the authentication phase and it has enhanced the *data encapsulation mechanism*, using two different systems:

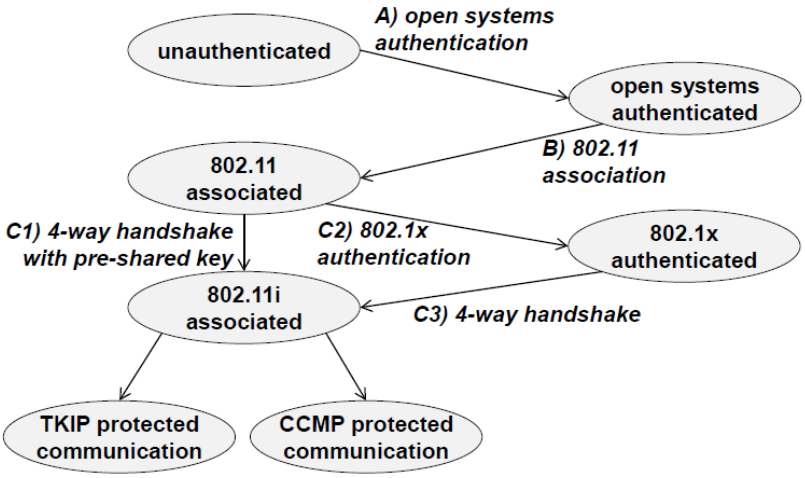
- **CCMP**
- **TKIP**, which is optional, in the sense that it was a temporary solution which provided *Transition Security Network*. Nowadays TKIP should never be used, since it was temporary for those systems which didn't have the hardware capability to implement CCMP.

Another improvement is in **the key management and in the key establishment**: there is a **four-way handshake** and **group-key handshake.**

There are also **enhanced authentication mechanism for STAs**: the **pre-shared key (PSK)** and the integration of IEEE 802.1x/EAP methods, introduced last year in ISS course.

802.11i state machine

The starting point is the *unauthenticated state*. If *open systems authentication* is used, then the next state will be *open systems authenticated* which will perform the 802.11 association, which is the *802.11 associated* state. At this point, in which, as we said, there is *no security*, the *802.11i phase* is started. For example, the 4-way handshake with one pre-shared key is used to become *802.11i associated*. The other option is to do not perform authentication using the wireless network, but by running first an *802.1x authentication phase*, passing through the *802.1x authenticated* phase. After being authenticated it is possible to run the *4-way handshake* to create the keys. After that step, again, there will be the *802.11i associated* phase.

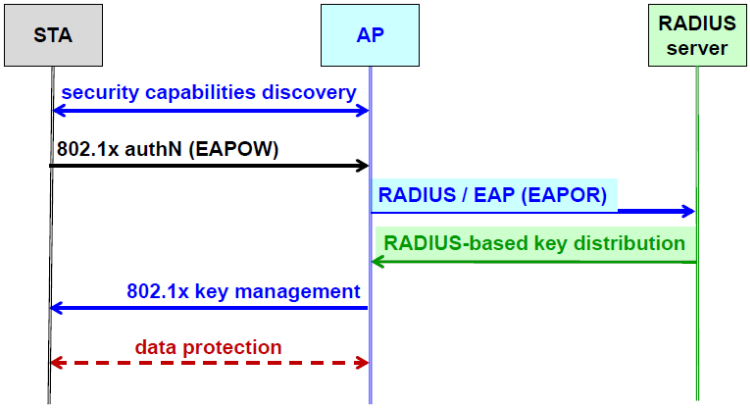


The choice between the two solutions depends on the kind of network. If it is the AP of a restaurant or internet café the preferred one is the C1 solution. If there is a company with several AP and a centralized authentication, then the C2 solution is used, since in this solution the 802.1x is integrated with RADIUS (centralized AP) and the same kind of authentication is used also for the wireless network, typically via EAP.

From the 802.11i associated phase it is possible to choose between TKIP (which should not be used) and CCMP.

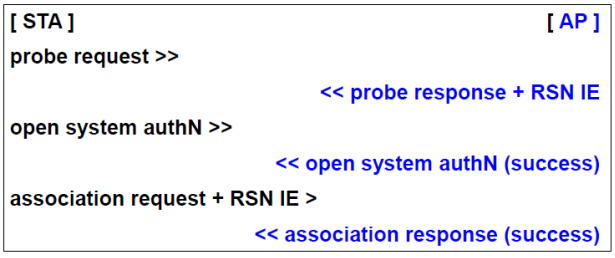
RSN

A robust security network is defined through the 4-way handshake. The STA is talking to access point to discover which security capabilities are available in this specific AP. Then the STA uses *802.1x Authentication* EAP Over Wireless, which means that the AP acts as the "ethernas" (probably Lioy's mistake, check it out), and will encapsulate the packet over RADIUS, to talk with the RADIUS server. Then, RADIUS will use a key distribution mechanism to provide the key and the AP will provide the key to the STA using *802.1x key management* protocol, in this way data will be protected.



RSN – Security capabilities discovery

STA sends to the AP a *probe request*. The AP will send back to the STA a *probe response* plus a packet which is specific for RSN, the *RSN IE*. Then, STA will send the *open system authentication*, the old one, which will automatically success (since there is no real authentication) and then the STA will perform the *association request* and will add the *RSN Identifier* that the STA needs. The "RSN IE" are the parts in which both agree, because first the AP sends the description of what is able to do, and then the station will select which measure it wants to implement.



The RSN IE (*Information Element*) conveys the following information:

- All enabled authentication suite, which can be:
  - 802.1X used for both authentication and key management
  - No authentication, 802.1X used only for key management
- Rather than having one fixed ciphersuite (in WEP it was RC4 with CRC32, here there are:
  - All enabled unicast cipher suite
  - Multicast cipher suite

The RSN cipher suites available:

- WEP (40 bit key), WEP-104 (104 bit key): be careful that even if the network is labeled WPA, still they have backward compatibility. Apparently improved but implementing same old features.
- TKIP (temporary, phased out)
- WRAP (AES-OCB, optional, patent problems. OCB is an Authenticated Encryption system)
- **CCMP** (AES with CTR-CBC, compulsory, the preferred one): everything labeled WPA or WPA2 must support CBC mode and that should be the one negotiated.

All the rest is more for backward compatibility, but as always when there is a negotiation, we run the risk to use something insecure. At least a monitoring system to check that by accident or attack we don't negotiate something old should be present.

As a system manager of the AP, it is possible to simply deny the use of WEP, TKIP and WRAP. Of course, old client that can't use CCMP will not be able to connect with the AP.

WRAP

WRAP *was Wireless Robust Authenticated Protocol* and performed *authenticated encryption* with *AES-OCB* (Offset Codebook Mode). It was the first selection in 802.11 but later was made optional due to patent problems, basically due to high license costs.

TKIP

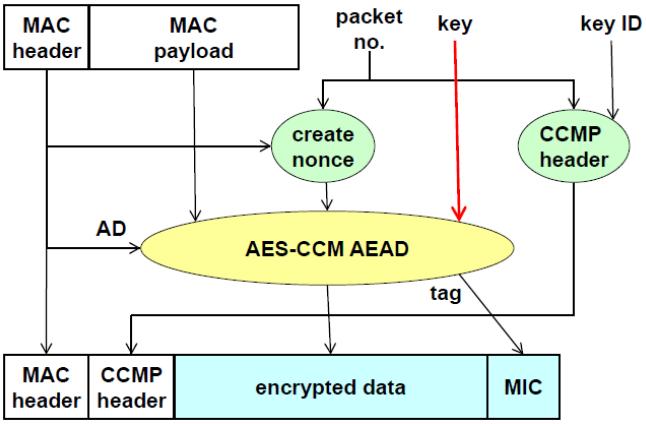
This is the *Temporal Key Integrity Protocol*, which uses WEP as final format, so the format is the same as in WEP, but it has improved, since it uses **per-packet 128-bit key dynamically generated**. It means that there is a key mixing phases, where it mixes: the transmission key TK, the sender MAC, the sequence number, and then it generates a key and an IV for WEP encryption. Same format, but different philosophy.

Rather than using CRC, **Michael** algorithm is used for *integrity*. It is an imperfect MAC, because it uses a 64-bit key, but the actual protection is of just 20 bits. It is not a good solution because there is also the possibility to retrieve the keystream from short packets to use it for re-injection and spoofing.

For anti-replay, TKIP uses **IV as a packet sequence number**. So sequence number is incremented with each packet sent and discard out-of-order packets when IV received is the same or smaller of the previous one.

CCMP

It is the preferred because it is the one which gives real security. CCMP is *Counter Mode with CBC-MAC* Protocol. The MIC of the header and payload is generated with CBC-MAC, which implies the use of key. The MAC is no more independent of the key. Then, MIC and payload are ciphered with AES-CTR. Encryption and authentication use the **same key**.



The starting point are the header and the payload of the packet. The header, of course, will remain in clear and, since it is AEAD (Authenticated encryption with Associated Data), anyway the MAC is used as AD, so it is protected for authentication and integrity. The MAC, together with the packet number, is used to create a *nonce*, which is the IV for the algorithm. The payload is the part to be be encrypted, and both Key Identifier and packet number are used as header of the CCMP. Then, the AES_CCM mode is generating the encrypted data, and the tag, which is put as MIC in the end. This is a correct implementation of the principles of authenticated encryption, also using the packet number so that is possible to achieve anti-reply protection.

It is true that packets should not be lost, but since there are existing interferences one packet might be jammed or disturbed too much. So, in generally, it is not protected against cancellation attacks (it is not possible to assume that all packets are received: there can be lost packets).

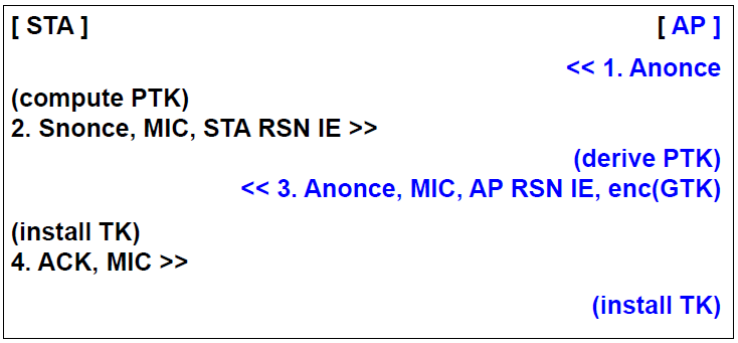
802.11i key hierarchy

Keys in 802.11i are quite complex, due to the hierarchy.

- **MSK** (*Master Session key*) is created with the initial phase, so during the 802.1x/EAP authentication or during the pre-shared key authentication.
- **PMK** (*Pairwise Master Key*) which shows that there are different keys between the different stations and the **GMK** (*Group Master Key*), a specific key specifically used to protect the broadcast and multicast traffic. These keys are generated from MSK.
- **PTK** (*Pairwise Transient Key*) is used to protect **unicast traffic** of one STA with the AP, which is different for each station.
- **GTK** (*Group Temporal Key*) is used to protect multicast and broadcast traffic between the AP and all the associated stations (i.e., everything inside one BSS). This key is shared between all members of BSS.

4-way handshake

This is needed to demonstrate to each other that the STA and the AP know the PMK or the PSK, via EAPoL-Key messages.



The AP sends a nonce and thanks to that the station can compute the PTK. Then it sends the nonce of the station, the MIC and the station information element. The AP, with that information, can derive the PTK and send its nonce, the MIC, its information element and the $enc(GTK)$. This is done because the station has got a pairwise key and that will be used to encrypt the group-key (given only to the authenticated stations that

become part of the group). Then, the station will install the transient key (TK) and will send back a final ACK with the MIC and the AP will do the same by installing the same transient key.

For pairwise keys these are all the used formulas (no need to remember them):

$$PTK = EAPoL - PRF(PMK, ANonce\ |SNonce\ |\ AP\_MAC\_addr\ |\ STA\_MAC\_addr\ )$$

It is possible to see that pairwise keys contain the MAC address of the AP and the STA, that's why they depend upon that specific pair.

$$PTK = KCK\ |\ KEK\ |\ TK\ |\ Tx\ |Rx$$

Then, the PTK is the concatenation on all the other keys. The algorithm is quite complex:

- 128-bit KCK (Key Confirmation Key) for MIC in msg 2, 3, 4
- 128-bit KEK (Key Encryption Key) is needed to encrypt sensitive data (such as RSN IE, group key)
- 128-bit TK (Temporal Key) used for unicast data encryption
- 64-bit Tx key (Transmission Key), used in TKIP for MIC of AP > STA packets
- 64-bit Rx key (Receive Key), used in TKIP for MIC of STA > AP packets

Note: even if TKIP is not used, the packet PTK will include everything. So, every time all the keys are generated and concatenated in the PTK, done to simply and not running different handshakes and different key generations.

The MIC in messages 2 and 3 protects from alteration in discovering the RSN IE exchange. On the contrary the MIC in the messages is used for mutual authentication (against MITM).

Group key handshake

$$GTK\ =\ GTEK\ |\ Tx\ |\ Rx$$

- 128-bit GTEK (Group Temporal Encryption Key) to encrypt multicast and broadcast data packets
- 64-bit Tx key, used in TKIP for MIC of multicast and broadcast data packets sent by the AP
- 64-bit Rx key, unused since STAs do not send multicast and broadcast data packets

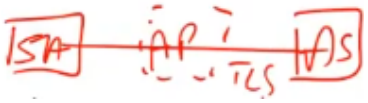
It is again the concatenation of the real key, the *Group Temporal Encryption* Key, which encrypts multicast and broadcast, and again two keys used for TKIP in the two directions.

The GTK is updated due to **time limits** or when a **STA is leaving the BSS**, because when a station performs de association the group key must be updated for all the others, otherwise the station which has disconnected could the steal/read the group messages. It is done by 2-way handshake where the AP sends the new GTK to each STA and BSS encrypted using the KEK of that STA and protected with the MIC which is important for authentication and not only for integrity. Each station acknowledges the new GTK and replies to the AP.

802.11i and 802.1x/EAP

To use all of this with 802.1x, there are various way:

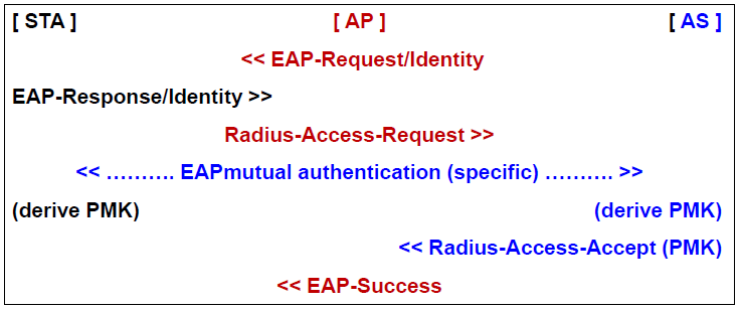
- **EAP-TLS**: it uses TLS for mutual authentication between the authentication server (AS) and the STA. This implies that the STA must have X.509 certificates (so, a key pair), which is not so common.
- **EAP-TTLS (Tunneled TLS):** in this case the TLS channel is created without client authentication, but only with server authentication and note that the **AP is not authenticated**. The RADIUS server is the authenticated one, because that is the one that we're talking to. The AP is a pass-through device. There is a direct TLS channel between the STA and the AS. The AS will decide which specific method to use (EAP methods or PAP, CHAP, MS-CHAP_v2 the most used one). The MS-CHAP_v2 is used by PoliTo over Eduroam connections. Tunneled TLS authenticates the RADIUS server (no fake servers possible) but then inside that protected channel twill be run a specific algorithm for that station. That can be based on username and password (eduroam uses it).



- **PEAP** (**Protected EAP**), it creates again a TLS tunnel with AS authentication only, but in this case only EAP methods are permitted in STA authentication (it is possible to use PAP, CHAP or MS-CHAP-V2 because EAP accepts only secure methods).

EAP authentication and key derivation

The authentication that is performed with 802.1x is end-to-end between the STA and the AS (Access server). Here there is a problem: the PMK must be known also to the AP, because the encryption, the protection, will be between the STAs and the AP. So, the AS besides talking with the STA will also talk with the AP to provide the required key.



There is first the *EAP-request/identity*, because when STAs try to connect to the network, the AP will ask for the identity. The STA sends its identity in an *EAP-response/Identity*, which is passed to the AS inside the radius packet. Now the AS will run an appropriate *EAP mutual authentication*, depending on what is selected: TLS, tunneled TLS or PEAP. At the end of that phase, the STA and the AS will have the PMK based on 802.1x, so, the key will be the result of 802.1x key management (remember that 802.1x protocol is not only used to authentication but also for key management). Since the PMK is needed in the AP and not in the AS, the *Radius-access-accept* that is sent to the AP to inform that STA is recognized and authorized to enter in the network, **will also contain the PMK**. Finally, consequently, the AP will send the EAP-Succes packet to the STA and at the same time the AP will have the PMK information.

WPA

WPA (*Wi-Fi Protected Access*) and that is a **reserved word**, in the sense that it is the used term to identify the supposedly short-lived version to overcome the WEP weaknesses, with the use of TKIP instead of WEP, with the use of 802.1x authentication and dynamic key management. Wi-Fi manufacturers, to get Wi-Fi certifications for any equipment (the AP or clients), the compatibility to WPA is compulsory since 2003.

WPA2

WPA2 is the *Wi-Fi Protected Access v2* (defined in June 2004). It completely replaces WPA, and it also replace the use of TKIP with AES-128 in CCMP mode. All the rest is equal to WPA.

WPA3

WPA is the *Wi-Fi Protected Access v3* (defined in January 2018) and it is now starting to be available in some products. It comes in two different modes:

- **Enterprise mode**
  - It has got a cryptographic strength of 192-bit because it uses *AES-256-GCM* (the strength of that would be 256) but that is paired with *HMAC-SHA-384*, remembering that the strength of HMAC is half of the length of the digest, and since the complete strength is the minimum one, then it obtains a strength of 192 bits.
- **Personal mode**
  - It has got a 128-bit cryptographic strength, because it uses *AES* in *CCMP-128* mode.

The difference between the two modes is that the enterprise mode assumes that you are also using 802.1x authentication, so it is a bigger, larger, and complex infrastructure, while personal mode is typically used when only one AP is managed (as usually done in home networks). Another big advantage of WPA3 is that it **completely replaces PSK** (pre-shared keys) **authentication** with **SAE** (*Simultaneous Authentication of Equals*) which provides much better security and forward secrecy. Moreover, WPA3 mandates the use of another protocol, which is 802.11w to protect management frames, otherwise could be possible to associate/deassociate without proper authentication.

Some attacks

Weak password

If a weak password is used, the PSK of WPA and WPA2 are vulnerable to password cracking, because the WPA passphrase hashes are seeded from the SSID name and its length. So, since the SSID name is known and, in some cases, it is a standard one, rainbow tables have been created for the top 1000 network SSID and a multitude of common passwords. It does require only a quick look up to speed up cracking WPA-PSK. If rainbow table doesn't work, **brute forcing** can be attempted using the *Aircrack Suite*.

WPS (Wi-Fi Protected Setup)

WPS is a simplified way to perform an association to an AP, to avoid the use of a password by simply pushing a button instead, or if it is not possible to push the button is possible to enter an 8-digit PIN, often written on the label attached to the AP, which is very bad. There are various flaws that permit to recover the PIN and WPA/WPA2 PSK, the best solution is to **completely disable WPS**.

WPS weaknesses

WPS PIN is composed by 7 digits, +1 for checksum. It means that only 10^7 attempts are need at most, which is not so secure. The Registrar, that is the AP, when receiving a PIN, it reports if the PIN is valid or not, separately for the first and the second half. It means that the AP will tell if the first four digits are good and if the following 3 digits are good or not. It is possible to start working only on the first 4 digits (only 10^4 attempts) and once the correct 4 digits are found, it is possible to start working on the other 3 digits with another 10^3 attempts. It is a total of 11000 attempts, even worse than the previous evaluation. This technique is used in the **Reaper** tool and sold for 1.5 million of US dollars.

Pixie Dust attack

It is a brute force offline attack. Once we get the MAC values used for mutual authentication, since that is using a key, it is possible to try and brute force it to recover the key. The easiest way is to just ask to the system for the Wi-Fi passphrase. If WPS is enabled, then it is possible to connect with windows client through WPS. If we have admin privileges on Windows, it is possible to display the properties of the adapter and then click on show characters. In this way the passphrase is shown, even if we didn't enter it, but using only WPS. There is something similar if using the Intel PROset wireless client. This client asks if you want to reconfigure the AP, in this way it is possible to decide what will be password to access the AP.

Electronic identity: delegated and federated authentication, policy-based access control

Electronic Identity is very important nowadays because we base most of the applications on Internet, but the identity is difficult to demonstrate on Internet. Demonstrate in a secure way what is your identity in an electronic world is very important and we will address various modes, in particular **federated** and **delegated** authentication and policy-based access control that heavily exploits electronic identity to decide which actions you can perform in a system.

Delegated authentication

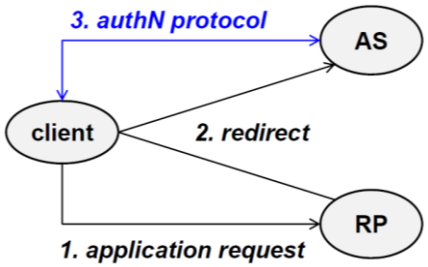
When talking about authentication there is a Relying Party that needs someone to authenticate the users. The RP (Relying Party) can validate the users by itself or may decide to **delegate** authentication to a separate entity which is the **Authentication Server** (AS). This is typically done when you have several applications that want to share the identities of their users e.g., in the network of PoliTO where you have one AS and you can use your username and password for all the services offered by PoliTO.

The AS is performing the authentication on behalf of the RP. But the problem is that the AS is a different entity from the RP. So, the AS will interact with the users that will use an authentication client with one among a set of authentication protocols. For example, in the PoliTO AS there are three options:

- authenticate with digital certificate that is an asymmetric challenge response
- authenticate with username and password
- authenticate with SPID which is the Italian electronic identity infrastructure

Once the authentication protocol has completed successfully the problem is how the authentication server can provide to the RP the authentication result in a secure way. Typically, what is returned is a **ticket** or **assertion**, that needs to be secure.

The RP is typically the application service to which the client is trying to access, and the AS is the one that this RP is willing to use for performing authentication. The client performs an **application request**, the RP will ask for an authentication with the AS (**redirect**). The AS will run the **authentication protocol** with the client. The problem that needs to be solved is how the AS will be able to confirm the identity of the client to the RP.



Transmission of authentication result

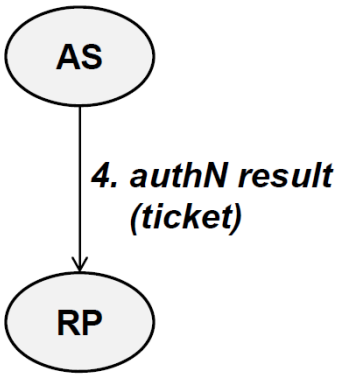
This is the problem in delegated authentication. The AS can transmit the authentication result in various ways and when evaluating the various solutions, various things should be checked:

- **speed**: for the RP it is important, since it can't wait five minutes to get the authentication result
- **security and trust**: if someone can create a fake response then the authentication fails
- **implications on services** offered**, interfaces** that must be available to implement that exchange, **network filters** like firewalls, because the RP, the client and the AS can be protected with it, so deciding in which direction the traffic flows is important because some firewalls could deny that kind of traffic.

There is no single correct or best solution, it must be selected based on the application scenario. This is a typical design question, if you have this kind of environment what the best solution for it is.

Push ticket

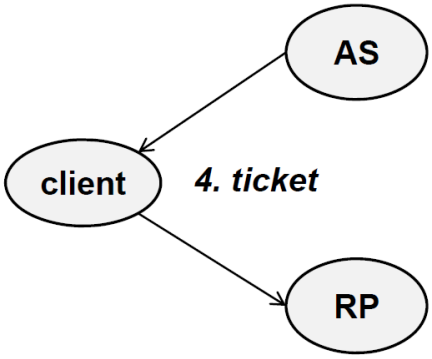
Let's analyze the first solution. The first three steps are the same as in the previous picture on previous page. Now there will be a fourth step which typically involve the AS and the RP. In case of transmission of the authentication result in the form of a push ticket the fourth step is the authentication server sending the authentication result to the RP that is providing a ticket that attests the identity of the authenticated user.



The ticket is sent directly from the AS to the RP. For example, SPID here seems good because as soon as the authentication protocol has been completed, immediately the result is returned to the RP. From the point of view of network filters the **client is not involved**, the authentication server performs an **outgoing connection**, which is normally permitted by firewalls, you can eventually whitelist the RPs if you want to limit the communications. On the contrary you could have problems at the RP, because to receive this ticket the RP **must expose a service**, an endpoint to which the AS can send the result. Secondly **it must accept ingoing traffic**, which is normally not very good from the point of view of firewalls. That could be accepted by whitelisting the AS since typically the RP will have only one AS. As there is direct communication between AS and RP, if the ticket is protected, then we can think that we have good security and trust. In this solution the biggest problem is the **correct configuration of the network filters** as well as the fact that the RP must create and manage public end point for receiving the ticket.

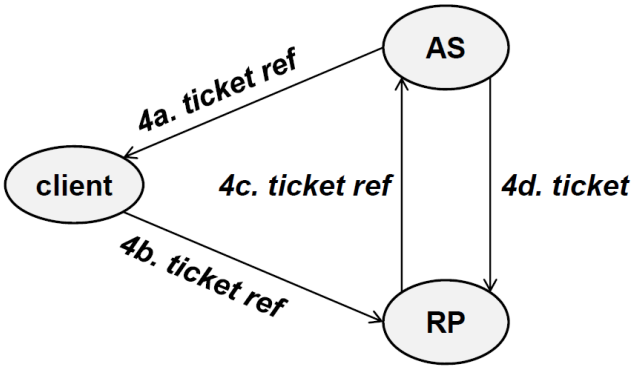
Indirect push ticket

Imagine that the AS and RP have a strong boundary protection and their firewalls do not permit the kind of traffic needed by a push ticket. It is possible to decide to involve the client in the exchange. This is an **indirect push ticket**: once the authentication protocol has been completed the **ticket in sent to the client** and then the **client has the duty to send it to the RP**. That simplifies the problem of the filters as well as clears the need to offer an additional end point, because the AS is returning the ticket to the client, maybe as part as last step of the authentication protocol and the client already has got the communication channel with the RP because it requests the application service and so the ticket can be sent as part of that exchange. In this case there are no problems with network filters and end points, but the big problem is how the ticket is protected. Since there is no direct communication between the AS and RP, we must either trust completely the client which is normally bad, or adequately protect the ticket.



Push reference + pull ticket

Another solution is to push a reference and to pull a ticket. The ticket is not directly sent to the client, but it sent a reference to the ticket and then the ticket is pulled from the relying party to the authentication server. Looking the picture, the authentication server is sending to the client a reference and this reference is passed to the relying party. Now, the RP knows where the statement about the result of the authentication is. The relying party will ask to the authentication server to give the result of the authentication for this client and finally the ticket will be returned. Here, just for the number of steps involved, it is the **slowest solution**. The problem of security and trust of the client is mitigated, because we are not sending the real ticket, but a reference. We change who needs to have some public endpoints and what is the direction of traffic between relying party and authentication server. Now



the authentication server **must provide a public endpoint** and **must accept incoming traffic** while the relying party has just to have the ability to open the channel towards the authentication server. This is visible because normally for the authentication server to manage users on behalf of the relying party, the RP should be known. So, you should have the "*4c.*" authenticated and verify that it's a valid relying party.

Problems with tickets

- **Binding with the client**
  - One general problem is how the **binding with the client** is performed. You could, for example, put the IP address of the requestor but if this is a shared machine there could be some problems, it would be better to identify the real user.
- **Ticket authentication**
  - If I get a ticket, am I sure that the ticket was generated by the authentication server? If you're sending the ticket through the client, then you must consider if the client can change the content of the ticket or not, or if the ticket can be manipulated while in transit by a man in the middle (in any place). **Authentication** and **integrity** of the ticket is not the only security property needed.
- **Ticket manipulation (at client, by MITM)**
- **Ticket sniffing (in the network / at client) – privacy!**
  - The **ticket could be sniffed** in the network or when it's stored temporarily in the client because the ticket may contain relevant information about the requestor, so privacy is at risk.
- **Listening services at RP**
- **Incoming firewall at RP**
- **Ticket replay (by same client)**
  - A client that receives a ticket, next time will provide the same ticket to the RP unless there is an anti-replay system.
- **Ticket reuse (at different clients)**
  - If the ticket is not bound to the specific client, maybe it could be possible to use a different client and even if the ticket is containing, for example, the IP address, by using IP spoofing it could be possible to reuse the ticket at a different client by using the same IP address.

Ticket protection

Definitely, we need ticket protection.

- **Direct transmission**
  - In case of direct transmission between authentication server and relying party, it is possible to protect the ticket intrinsically, for example, it can be **digitally signed** by the authentication server and then encrypted for the relying party so that only it can read the content.
  - It could be possible to **rely also on a secure channel** which means that we need to authenticate the authentication server, to protect packet integrity and authentication during the transmission, to have packet encryption and then no replay. All these properties can be achieved with TLS. If we want to filter out invalid relying parties, we could have a TLS channel also with client authentication.
- **Indirect transmission**
  - If we use indirect transmission via the client, then we must protect the ticket intrinsically because there's no direct connection between the authentication server and the relying party. So, in that case the only security that we can implement is **digital signature** by the authentication server and the encryption for the relying party.

For protection for the replay or reuse it could be possible to insert in the ticket a **timestamp**, so that it tells when it has been generated and then it must be decided also a **time limit** for reuse of the same ticket. This is tricky: we would like to make the time limit as short as possible, like 30 seconds or 1 minute at most, so that even if the ticket is replayed or reused that would be for a very short time. Remember tough, that once the

ticket is expired, the authentication protocol needs to run again so we should use *cookies* or *sessions* for maintaining the state but those have typically insecurities as well. Secondly, to avoid replay or reuse, we should implement the **binding of the ticket** either with a *user ID* and *a network address* or, at least, *a network address*.

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.

In this case we use federated authentication. The problem here is 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 can 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.

XACML (eXtensible Access Control Markup Language)

XACML it's a specific language to give instructions about access control, since in multi-domain environment a common way to specify the common rules for the different parts and different domains**.** It's a **markup language**, so it is derived from XML, but it's focused on access control. It is in general a language to describe **authorization policies**: you know that authorization assumes that you have first authentication of the peers that are interacting. These policies are defined in terms of:

- **Subject**, that wants to perform a certain operation (can be users, computers or network services).
- **Resource**, to be accessed (documents, files, data or network ports). In general, since we are more oriented to internet, everything is identified through URI.

This language helps you to manage the access to resources protected by authorization. So, XACML is a **data format** to represent the request (for access) and the response (access permit/deny). XACML does not specify a protocol to carry these requests, so the client-server protocol for transmission must be chosen.

It's an OASIS standard (which is standardizing everything regarding XML).

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 service and 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.

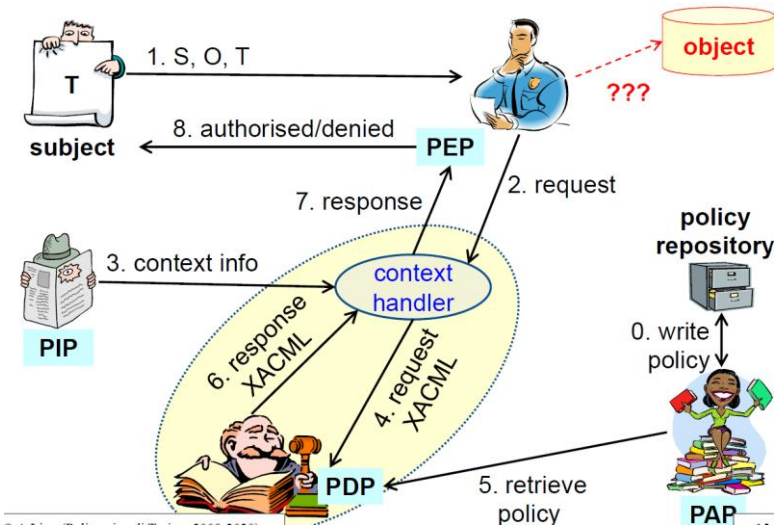
COPS was not very successful but it leaded to the foundation of what came subsequently. After this initial attempt, there was a generalization, an extension, to the **management of information systems** (by DMTF: *Distributed Management Task Force*) and to the **access control in distributed environments**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

Components policy-based access control

These are the components of a policy-based access control system, it is not compulsory to have all of them in place but this is the most complete list:

- **PEP = Policy Enforcement Point**: it is needed for sure, since it is something, typically a *security control*, that protects a resource and allows access only after verification of compatibility with the policy. So, this is the one implementing the *access control*. But, it needs to know if the request should be accepted or not. In that case, it must work with PDP.
- **PDP = Policy Decision Point**: it's the one that receives all the information related to the current request (what is *the applicable policy, the subject, the resource, the access type, the general context*) and decides whether to permit or deny the access.
- **PIP = Policy Information Point**: it's the one providing the additional information about the access requested.
- **PAP = Policy Access Point**: this is the entity in charge of managing the policies that can be applied to the different requests.

Especially for PEP and PDP, a lot of people talk about them, even if they don't use XACML, since it is a generalized concept.



Somewhere there is the *policy repository*, where all the access policies are stored. Then there is the *PAP*, that, among the other tasks, is also in charge of writing the policies and, later, to retrieve the applicable ones. Then there is a subject (*user/router/network service*) that wants to access an 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. 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. So, finally, the response is provided to the PEP that will implement the response: you are authorized and PEP will allow the connection to take place, or you are denied.

Note that XACML is limited to the part in yellow in the picture, 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.

There are not so many implementations of pure XACML access control policies, but the concept is widely used. When you have multidomain/multiservice access control system, you typically talk to one or more PEP, because one PEP, can then feed other PEPs as well, e.g., $PEP1 \rightarrow PEP2 \rightarrow PEP3 \rightarrow \cdots$, that is typically something in which you centralize the decision and then you distribute the decision to all the points that will implement the access control policy. This is very good because when you go to large information systems, one of the more important things is to have a uniform view of the system. Anytime you can centralize some decisions, that's good. Implementing the same thing inside the various servers would be prone to errors. In this way there is just **one policy** to be applied to all the applications in my area, any application which is PEP will need to query my PDP and this will take the decision. This is important also because it is possible to **write logs**, which is important in order to detect if *inappropriate actions* have been performed or *inappropriate users* were connected. We must not think that our system will always be secure, we must always be ready for errors/attacks.

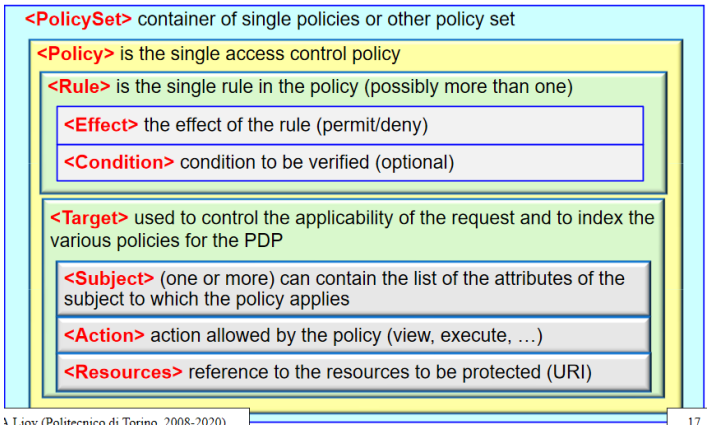
Another important thing is that after the attack has taken place, we should be able to understand what's happened. Having a centralized point means that even if hacker is attacking here or is attacking directly the object, it is possible to have at least a trace of what's happened (in another place). Of course, it is possible to have a double log: it is possible to have the PDP and the PEP performing a log. Let's imagine that some bad action happened on an object: the owner of the object will go to the PEP saying "Oh, you have permitted that action" but then the PEP says "You told me to permit it, because I only implement your decision". So, it is very much important to have **distributed logs**. It seems unneeded, since the information (log) is already present on the PEP, but let's have the same information also in another place, especially if there is a company A and the other is a company B, which would lead to some discussions. In general, remember that logs should be append-only (another problem is that companies implement log in read-write mode which is wrong).

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.

XACML: policy format

The top-most element is the *PolicySet* which is a container of single policies or other policy sets, so it is possible to have recursive format. As a minimum there is one *policy*, which is the **single access control policy** and it contains:



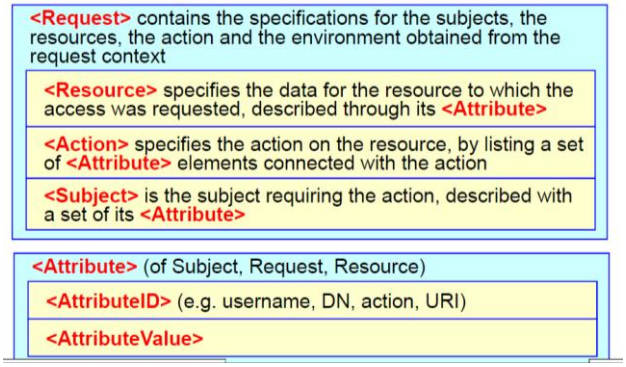
- **Rules**: each rule is one rule in the policy (of course it is possible to have more than one rule in the policy) and each rule contains two sub-elements:
  - The *Effect* of the rule (permit or deny)
  - The *Condition* to be verified (optional)
- **Target**: it says "this is a policy to be applied when there is a request for this target (object in previous slide)". The target is used to control the applicability of the request and to index the various policies for the PDP, so when a request for a certain object is received, a lookup is performed on all the policies that have as target the specified object. Then, for this target the rules are expressed in the following form:

- **Subject(s)** (one or more): contain the *list of the attributes of the subject* to which the policy applies. The subject is identified in various ways, for example: "any subject that has got an address in the subnet x.x.10.1" or "that subject that corresponds to the username Lioy" and so on. There is not a single way to identify a subject, and this field accommodates different ways to specify it. For example, this can also be used for **RBAC** (*Role Based Access Control*), and the subject could be anybody who has got the Role: "director", or "professor", or "student". It is possible to express policies in terms of single users or anybody who has got a certain role in the organization. That is RBAC, which is much better than identity-based access control.
- **Action**: the action allowed by the policy (view, execute, delete, create, …).
- **Resources**: reference to the resources to be protected (specified via URI).

XACML: request format

In a request there is the specification for the subject, resource, actions, and the environment obtained from the request context:

- **Resource**: specifies the data for the resource to which the access was requested, described through its *Attributes*.
- **Action**: specifies the action on the resource, by listing a set of Attributes connected with the action.
- **Subject**: it is the subject requiring the action, described with a set of its Attributes.



Notice that Subject, Resource (first called object) and Action are specified through their attributes. An attribute is, in general, made by:

- An **identifier** *AttributeID*. There are several identifiers pre-defined in the standard (e.g., username, Distinguished Name (DN) of the certificate presented by the peer, the action performed, the URI, etc.).
- The **AttributeValue**: which value must have that Attribute in order to permit/deny the access.

XACML: response format

The Response **encapsulates** the decision of the PDP is made by:

- **Result**: it is the taken decision, which contains:
  - **Decision**: contains the result of the application of the policy on the request (Permit/Deny/Indeterminate/NotApplicable).
    - *Indeterminate*: if there is an inconsistency in the policy, for some reasons it could be permitted, for others it would be denied, so it cannot take a decision.
    - *NotApplicable*: if there is no policy that can be applied to this specific request.
  - **Status**: represents the status of the result of the authorization decision (contains a status code, a message status and the status details).

SAML (Security Assertion Markup Language)

We mentioned that when we enrich with the context the request, it may be needed that the information is conveyed in the form of a specific statement, or assertion. For this reason, there is a specific language **SAML** which is another XML based language, specifically developed for security assertions, used to state something which is related to security. It is a data format, which is used to: **represent various types of assertions, construct requests for assertions** and **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.

An assertion is a **decision** (the assertion is the base object of SAML). SAML has the scope to simplify and to make standard the interactions aimed to establish permissions in a multi-domain distributed system, and it is

again an OASIS standard (based on XML syntax). For example, SPID is using SAML. The European Network for Identity (EIDAS) is also using SAML. Although XACML is not so much used, SAML is widely used for national and international applications.

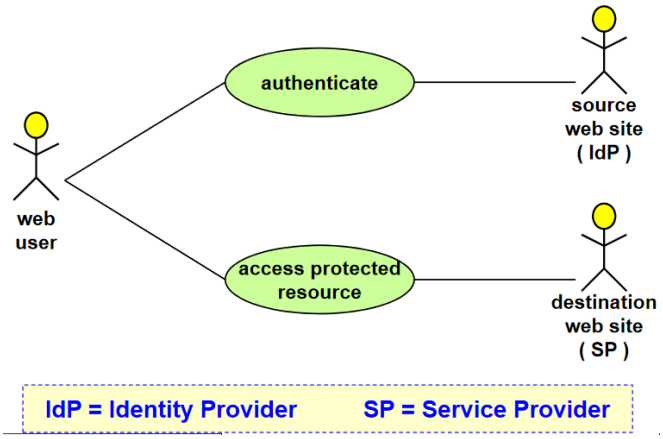
- *SAML 1.0*
  - November 2002
  - The original version
- *SAML 1.1*: improved 1.0 version about the security of the messages itself
  - September 2003
  - Now messages can be protected with **XML-dsig** (**REMIND**: this does not mean "digital signature", since with this method it is possible to use a MAC as well as a real digital signature)
  - It defined profiles for the usage of SAML with the web browser to implement **SSO** (*Single Sign On*), and there are these two profiles (profiles are ways to apply the standard):
    - **Browser/artifact** profile: the assertion (SAML token) is passed by reference.
    - **Browser/POST** profile: the assertion is passed 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.

SAML use cases

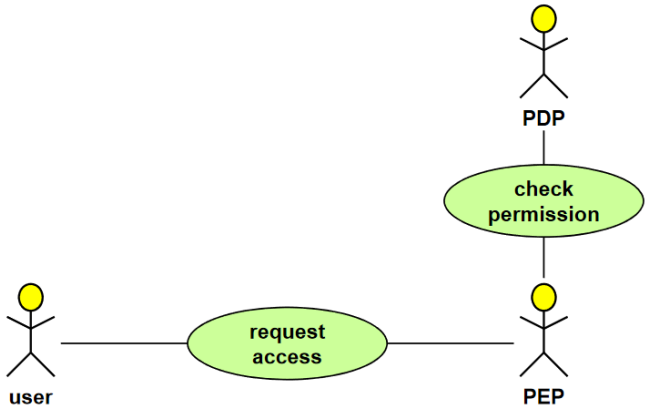
Web browser SSO use case

There is a web user on the left that would like to access a certain resource of the Service Provider. It is an *access protected resource*. The website does not want to implement by itself authentication and authorization, so it is using the *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, 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.



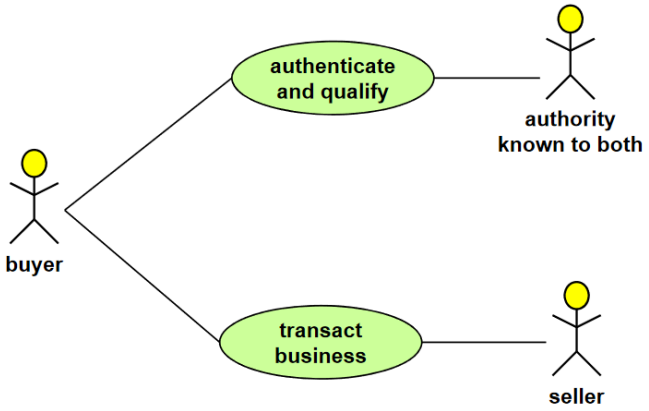
Authorization service use case

This is a XACML, but where is SAML here? The user is requesting the access, then there is the PEP, which means that on the right of the PEP (not present in the picture) there is the Service Provider protected by the PEP. The PEP needs to talk to the PDP to check permissions. This permission (or deny) can be returned in a SAML object (the SAML response). So, the SAML part is in the way from PDP to PEP. Maybe that is then encapsulated in XACML or not, it is not compulsory to use both. In practice XACML is rarely used (it is more needed in general as a model) and most of the implementations use SAML.



Back-office transaction use case

There is a buyer, that goes to the SP of a seller (e.g., I am in PoliTo, I want to buy something outside it, but I want to buy in the name of PoliTo). So, I must first *authenticate and qualify* (e.g., give me an assertion that states that I am Lioy and I have Politecnico financing me for 10'000€). This token, again, is returned in some way to the seller, as a proof that the user belongs to that organization, and it is authorized to perform that action in the name of the organization.



Note that the authority is known to both: for the buyer is important because it must qualify (demonstrate identity, role, etc.) and the seller must accept that kind of statement.

The presented use cases correspond to the 3 basic types of SAML assertions.

SAML assertion

It's a **declaration of a fact regarding a specific subject**. For example, in the assertion it is possible to declare that prof. Lioy is a professor (has got a role). This declaration is made by a certain issuer, like what there is in the PKC with the difference that these are not certificates but statements. Of course, the **issuer is trusted**, because by asking to it for a statement, it will be paid to state anything, in some sense it must be a trusted third part.

There are 3 types of assertions that correspond to the 3 previous cases just discussed. Assertion about:

- **authentication**: this is the one returned by the Identity Provider (IDF – SP)
- **attributes**: this is for the third case (back office)
- **authorization decision**: which is for the PEP-PDP

Those are the basics, but SAML is extensible, so it is possible to add other types of assertions. Assertion can be "digitally signed": it uses XML signature, so it can be a MAC, or it can be a real digital signature. You should check the implementation details, to detect the kind of risks.

Info common to all assertions

Whatever is the type of assertion there are certain elements that are in common to all assertions:

- **Issuer** (who created the assertion) and the **issuance timestamp** (date and time when it was created)
- **Assertion ID**, that uniquely identifies this assertion
- **Subject:** be aware that this is used in a multi-domain environment, so saying that "The Subject is Lioy" is meaningless, because by looking around there are at least 30 professors in the world with same name. So, which Lioy? The **name must always be qualified with a security domain**! (e.g., "Lioy in this specific environment".
- The assertion may contain "**conditions**":
    - if the conditions are not understood by the receiver (SAML client), then the assertion **MUST** be rejected.
    - There is one important condition, which **is assertion validity period**, which stands for *how much time this assertion is valid*. Because, for example, in ten years Lioy could be no more a professor and yet still have the assertion, which won't be valid anymore for the future.
- **Other useful information**: maybe there is an explanation/proof of the basis on which the assertion was constructed (e.g., you are asserting this thing, based on which evidence? On which basis?)

Given the general information, will now be introduced the assertions.

Authentication assertion

This is the assertion related to the authentication performed by an identity provider. An issuer (which is typically the IdP) declares that: ***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. Be aware: **SAML does not perform authentication**, it is not part of the protocol requesting the password, or performing challenge-response, …. SAML is only providing a mechanism to create a link with the **result** of an authentication performed by an authentication agent. The IdP, using a specific protocol will interact with the requestor (e.g., challenge-response, OTP, etc.), then once the authentication exchange will be completed, it will create an assertion that will say something like "Yes, authentication completed, and the result is *Lioy is authenticated with this username and this password*".

Example of authentication assertion

The opening tag for the data of SAML is *<saml:* which is in the picture specifically an assertion. There will be the version *major* and *minor* fields and then 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 which will specify the date and time of the creation (the Z at the end stands for Greenwhich Time).

```
<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>
```

The second part contains *Conditions*: in this case it was not valid before the specified time and will expire in a few minutes. This means that it is needed to use the token for accessing the SP before the token expires.

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.

Attribute assertion

An issuer is declaring that: ***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", …***

Most often this is obtained form an LDAP query, a directory service. For example: the user "*alioy*", in the specific security domain "*polito.it*", is associated with the attribute "*Department*" and the value is "*DAUIN*".

Example of attribute assertion

The Initial tag and Conditions are the same as in the previous example.

```
<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>
```

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.

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.).

Example of authorization decision assertion

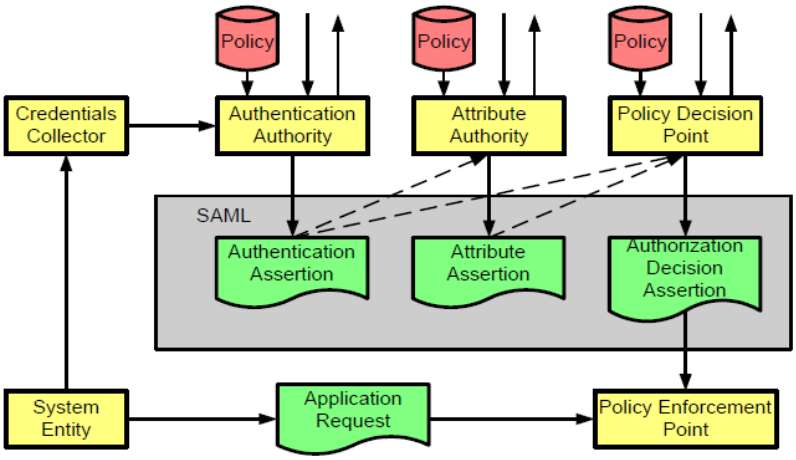
The initial tags are the same as before, but in this case there will be the *AuthorizationStatement*, in which there will be the *decision* field to specify if it is permitted to access the specified *resource* (*http://did.polito.it/m2170.php*) from the person specified in the Subject (the one corresponding to name *alioy* in the domain *polito.it*).

```
<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>
```

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: 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.

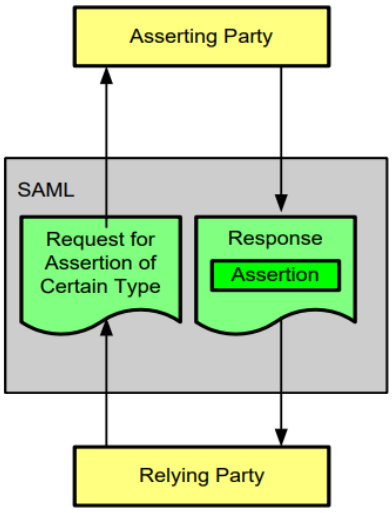


The assertions can be used individually, but they are quite often used together, which leads to the general schema presented in the picture, which is called **producer-consumer model,** in which there are several entities that at the same time are **producing an assertion** but also **consuming (receiving) 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) have got a specific policy which tells for example "*ok in order to identify the user you have to use password, digital signature, OTP and so on*", 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.

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 (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.

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*".

In this example of authentication assertion request we can see **samlp**, where p stand for protocol, then there are the version and the requestID.

Example authentication assertion request

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>
```

Trust relation

The assertion is part of a triangle due 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.

Binding SAML

SAML defines what to transport, while the binding defines **how** to transport it (e.g., a network protocol for SAML requests and responses), and there are quite a lot:

1. *SAML/SOAP-over-HTTP* is the original one. SOAP was *Service Oriented Application Protocol*, which was used long ago and nowadays it is neglected.
2. SAML 2.0 defined other bindings:
    a. *SAML SOAP binding* (based on SOAP 1.1) – again discouraged
    b. Reverse SOAP (PAOS) binding – don't care
    c. HTTP redirect (GET) binding
    d. HTTP POST binding
    e. HTTP artifact binding
    f. SAML URI binding

The most interesting are the ones which are **direct bindings with HTTP**, since nowadays everything is transported via HTTP.

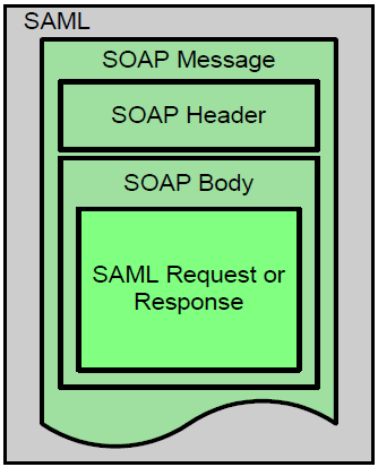
SAML Profiles

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

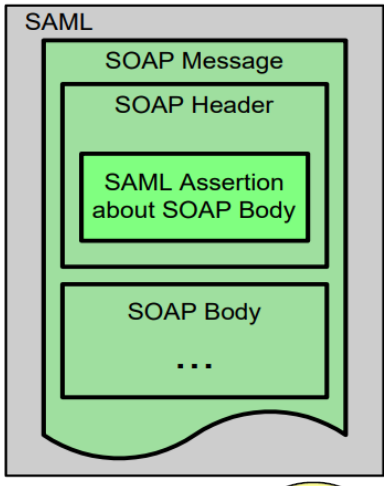
SAML: the SOAP-over-HTTP binding

This is the original one (nowadays no more used). The *SAML request or response* was put inside this message in the body. Then the header specified what was transported and everything was in SAML. This means that there were **three layers**: this thing is transported inside HTTP, but inside HTTP you are transporting SOAP and inside SOAP you are transporting SAML. Since nowadays SOAP is no more used, it is now used SAML directly inside HTTP.



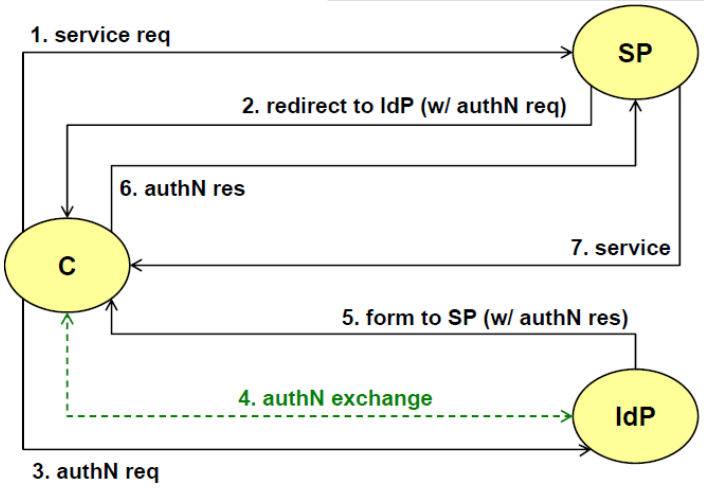
SAML: the SOAP profile

On the contrary, if HTTP is not used there is the SOAP body and an assertion about the SOAP body, but it is not used.



SSO push use case

In the picture the client (browser) is connecting to a specific page of a web server (which is the SP) and it want to get the page. The page 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.

In the number 5 there is a **form** to take the client back to the service provider and 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.

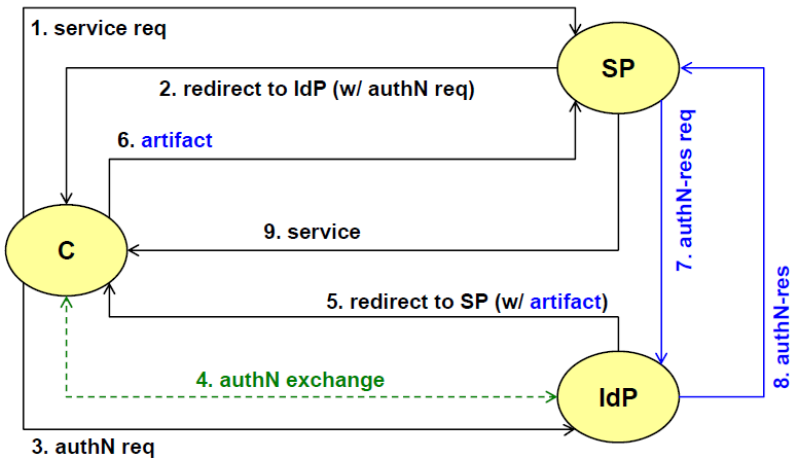
This use case is also named **HTTP redirect binding** which is composed of:

- C > SP – GET service URI
- SP > C – REDIRECT to IdP with SAML-authN-req
- C > IdP – GET with SAML-authN-req. Notice: since it is a GET there will be a URL with a *?req=XX*
- C : IdP – authentication exchange
- IdP > C – HTML form: POST to SP, with hidden field containing SAML-authN-resp
- C > SP – POST with SAML-authN-resp (it must use *XMLsig*, otherwise it is not possible to verify)
- SP > C – **verifies** SAML-authN-resp and eventually provides the requested service

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

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 and with the respect to the previous case there are:

- C > SP – GET service URI
- SP > C – REDIRECT to IdP with SAML-authN-req
- C > IdP – GET with SAML-authN-req. Notice: since it is a GET there will be a URL with a *?req=XX*
- IDP > C – HTML form: POST to SP with an **artifact (=pointer to SAML-authN-resp on the IdP)**
- C > SP – **POST with artifact**
- SP > IdP – **GET with artifact** (on a separate channel, typically a secure channel but not compulsory)
- IdP > SP – **SAML-authN-resp** (on a separate channel, typically secure but not compulsory)
- SP > C – verifies SAML-authN-resp and eventually provides the requested service

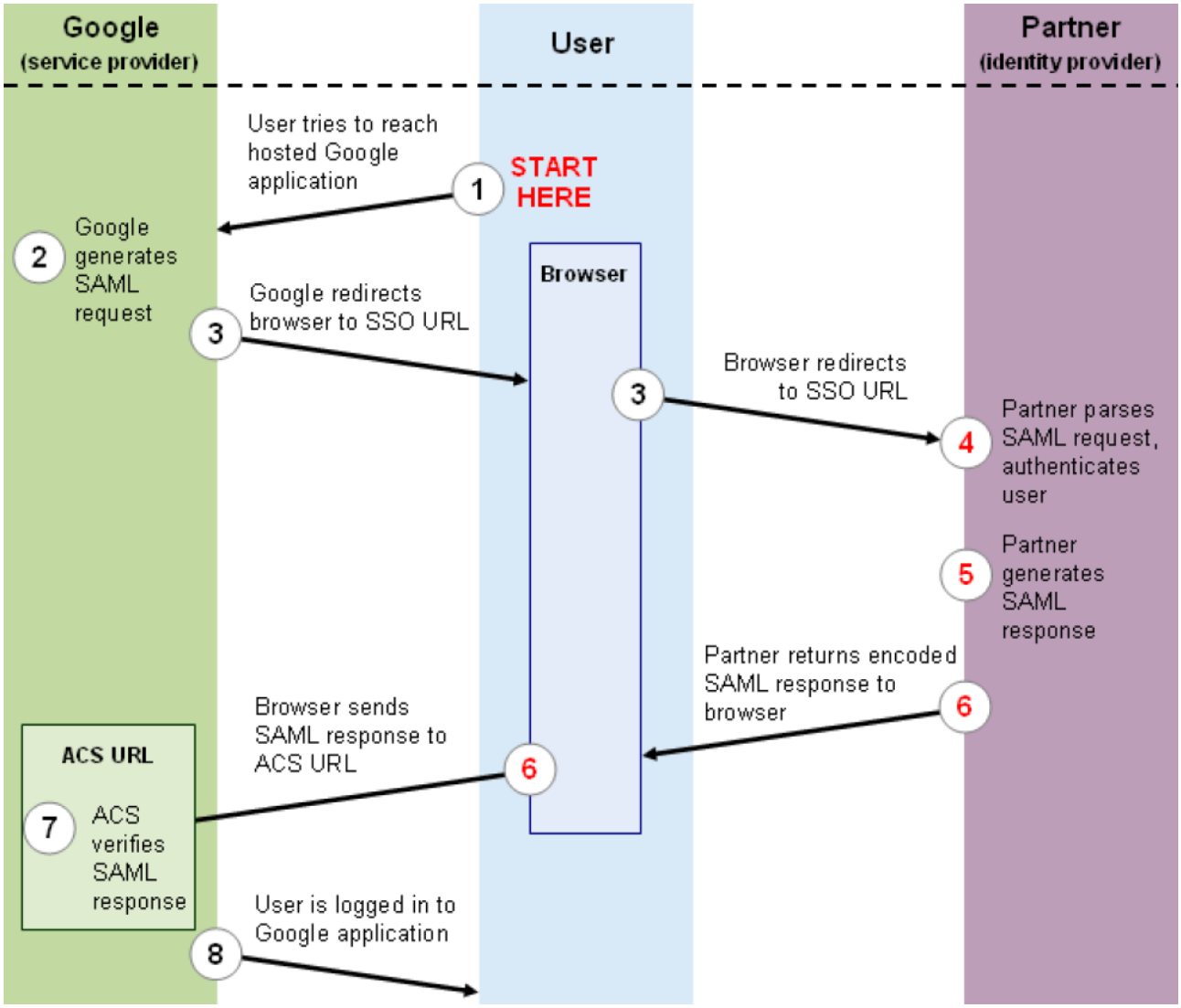
SAML SSO for Google Apps

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.



In the schema the user tries to reach the hosted Google application, for example, a webpage which is protected. There will be then a redirect to a specific URL hosted by the Identity Provider. Once the user is on the IdP the authentications protocol starts. Then the partner generates a SAML response, and it is returned to the browser (the client) which will send it back to Google but not at the original point but to an *ACS* (*Assertion Consumer Service*) which can accept the assertion, verify it and, if it is good, then the user is logged in the application and can perform his tasks.

Some details:

- The partner **must provide** to Google:
  - The **URL of its own SSO service**, because google will redirect the user there (a.k.a. IdP or AS)
  - The **X.509 certificate** to verify its signature
- The step 3 (the redirect) contains (in *opaque mode*, so the user is not seeing it):
  - The **URL of Google service** that requested the authentication
  - The **SAML authentication request**
  - The **URL of the ACS** (Assertion Consumer Service) so the point where, after authentication, the response must be returned
- The step 6 contains (in opaque mode):
  - The **URL of the Google service** requested by the user
  - The **SAML authentication response** with XMLsig (only the real digital signature)
  - The **URL of the ACS**

Federated identity

In so far, we have seen Delegated Identity, maybe inside the same security domain (PoliTo, 100 servers, all the servers delegate authentication to the idp.polito.it) but it's single-domain, or the Google example, in which it's delegated authentication, so Google is delegating the authentication for that specific application to that specific IdP, but there is always a match: 1 application, 1 IdP. The same identity provider can serve different applications but there is a 1:1 mapping.

On the contrary, **Federated Identity** is when there are multiple IdPs that can talk to the same application service. It is quite a common case: for example, in the creation of an account to a specific website, there are some websites that permit to choose to create a new username and password or to use Google/Facebook authentication. That's a **federation:** it is automatically recognized the authentication performed by the other service although they are outside the security domain. It is particularly interesting to avoid **duplicate authentication**, which is quite annoying.

Federate Identity **augments federated authentication with identity-related attributes**. SAML is often used to create federated identity systems because it supports both authentication **and** attribute assertion. Identity is **more than just authentication**, identity is **authentication + attributes** (e.g., name, surname, student id, residence and so on).

SAML is XML-based: XML is **simple but quite heavy**, so SAML is typically used in PC or server web-based environments. This means that it is difficult to support in light/mobile environments.

Now there is a juxtaposition: some people use SAML, which is not user-friendly for the mobile environment; other people use **OpenID-connect,** which makes things very similar to SAML (same architecture composed by client, SP and IdP), but this focuses on **native web application** as it uses **JSON** instead of XML, and **REST protocol** (because they are both native on smartphone and tablet).

Beware: *OpenID-1.0* and *OpenID-2.0* are **not** OpenID-connect, that is a completely different protocol that is based on *OAUTH 2.0* (an IETF *authorization framework*). So, OpenID-connect is for authentication but it uses authorization framework!? In fact, from the conceptual point of view OpenID-connect is confusing.

OpenID-connect (OIDC)

This is a **delegated authentication** system, we will get **federated** by the fact that we support many IdPs, but basically when you interact with a single IdP, that is delegated authentication.

It uses JSON data and REST protocol, 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**.

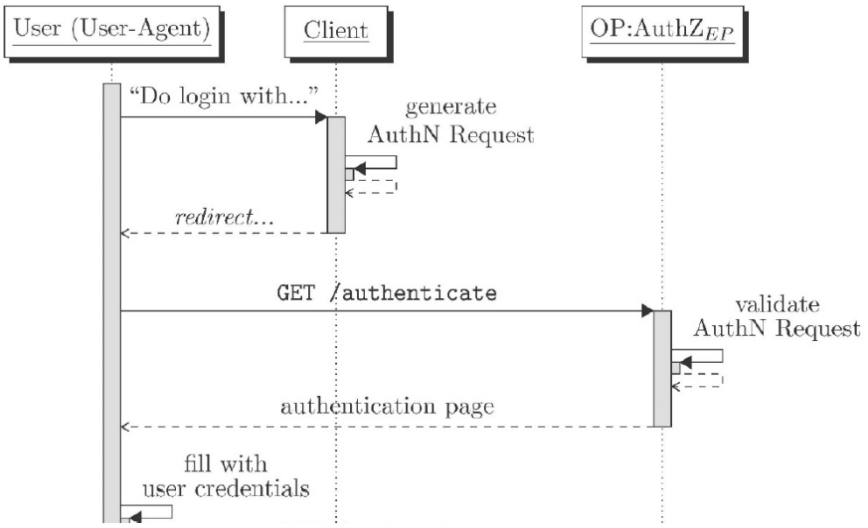
In this schema the user has its mobile phone, and it is named *User-Agent*, which is connecting to an application server, which is the client, because it is the client of OpenID-connect, connecting to a server that will provide authentication, authorization, and attributes. That's why it is named client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

The server (S in the picture) 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:

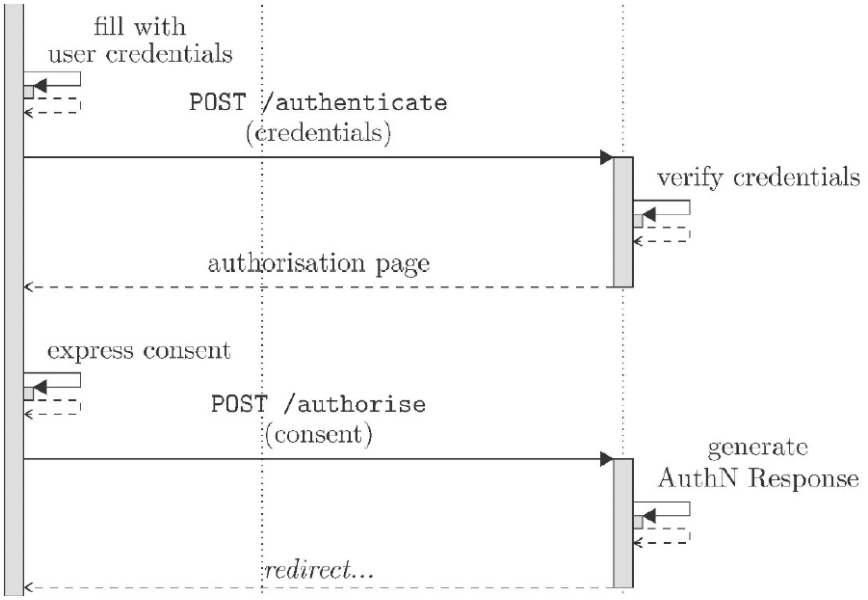
- **Authorization endpoint ($AuthZ_{EP}$)**: it is called authorization, but it performs **authentication** (confusing).
- **Token endpoint** ($Token_{EP}$), something that verifies if a certain token generated during the protocol is valid or not.
- **UserInfo endpoint ($UserInfo_{EP}$)**, if the user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

OIDC: user authentication



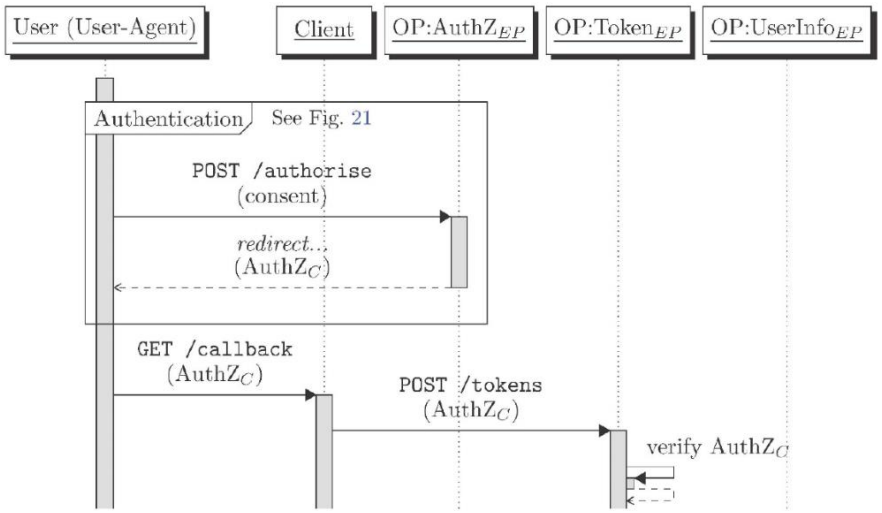
In the picture there are: the User-Agent, the client (the OIDC client, basically the server), the OIDC server (the OpenID Provider).

When the user is going to the application server, it will offer some alternatives such as Google, Facebook, etc. The user will select one of them and the client will generate an authentication request and it is returned in redirect, since it cannot be the client to perform the authentication but the user. The redirect will send the user to one OpenID-connect **authorization endpoint** (*/authenticate*, before that there is of course the complete URL). This is transferring the authentication request on the EP and the OpenID-connect will perform a check: is this authN request valid? It means: "*this application server has paid me to provide authentication? Have we got an agreement?*", if it is valid then the authentication protocol is run between the OpenID provider and the user.



The user will provide his credentials in the authentication page and will perform a POST to the same page to provide the credentials. The credentials are then verified and then there will be an **optional authorization page**. In the original request (the authentication request) is not only an authentication request, but it is a request the **authN of the user + additional information** that the client wants (*email address, list of friends, picture of the user and so on*). If, beyond authentication, other personal information was requested, then the OpenID provider is creating **authorization page** that permits the user to choose to transmit that information. If the user accepts, the authorization page will contain another URL, redirect (*POST /authorize*). Now, the authentication response will be generated saying that the user was authenticated and that it provided the consent to transfer additional information (if any) and there will be another redirect.

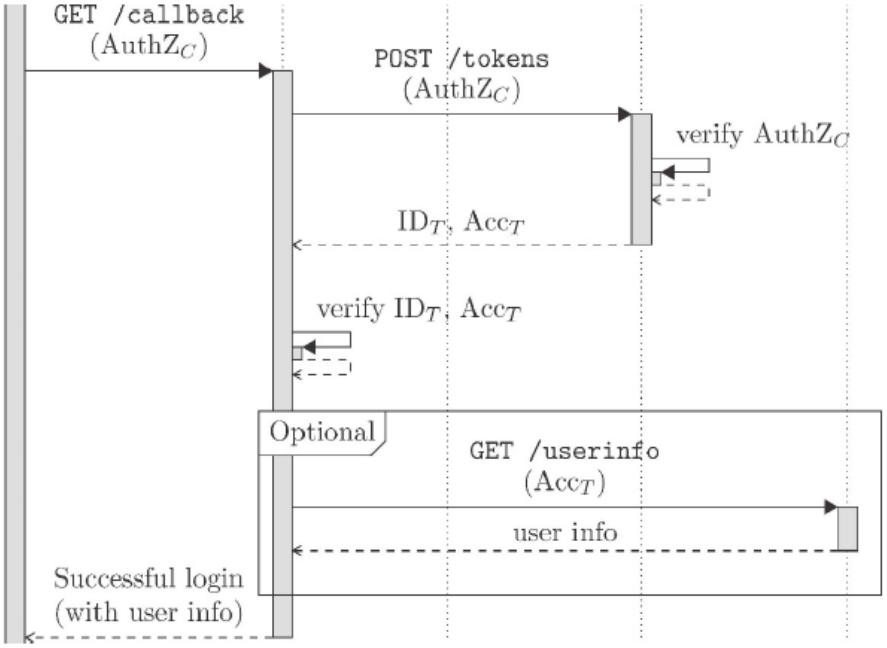
OIDC: login with token



Looking the picture, the part in the box is the one of the previous images. In this case the other end point takes place.

The first end point was named "Authorization" because it first performed authentication and then it performed authorization (giving consent) to transfer additional personal information.

After the redirect of the *AuthZ$_{EP}$* it is sent a GET to a */callback* point, passing the received information. The authorization end point provided a token in response. This token is now given to the client which can put it to the *Token$_{EP}$* to validate it, since it is demonstrating that the user accepted to transfer its data to the client. The token end point will verify if everything is ok, and if so, it will return ID_T + ACC_T which is verified and then if in the identity there was also the accessory information (ACC_T), then optionally the *UserInfo$_{EP}$* will be accessed to retrieve the user information. Finally, there will be a successful login providing to the client also additional information.



OIDC: 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)
  - If for example, someone accesses Google and pretend to use a gmail account, Google will agree by saying that the email is verified. If, otherwise someone says that the address is @polito.it it will transfer the information but with *email_verified* = *false* (not possible for Google to verify it).
- **Phone category**
  - *Phone_number*, *phone_number_verified* (Boolean)
  - If the number is used for verification, then the number is verified.
- **Address category**
  - Address
  - Cannot be verified.

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.

OIDC: JWT for claims – example of ID token

```
{
  "sub"       : "alice",
  "iss"       : "https://openid.polito.it",
  "aud"       : "client-12345",
  "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)
}
```

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).

eIDAS

It is the *identity system of the European Union* which is based on one specific regulation: *European Union Regulation no. 910/2014*. eIDAS stands for **electronic Identification, Authentication, and trust Services**. The focus will be only for the identity and authentication part.

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.

"*People and business can use their own national electronic identification schemes (eIDs) to access public services in other EU countries where eIDs are available.*"

The only limitation is the *public services*, in the sense that European Union can force only public services to adopt it. For private sector it is possible only to encourage the use.

It was adopted on 23rd July 2014 and initially the eIDAS eID infrastructure was initially voluntary but now compulsory for public sector since September 2018.

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 Decision (EU) such as:

- 2015/296 (24th February 2015): *eID procedural arrangement for MS (Member States) cooperation*
- 2015/1501 (8th September 2015): *interoperability framework*
- 2015/1502 (8th September 2015): *technical specifications for assurance levels for electronic identification means*
- 2015/1984 (3rd November 2015): *formats and procedures for notification* (in a country there may be multiple electronic identity systems, maybe it is wanted to limit citizens to use on abroad only one specific way, so other countries are notified about the way on how the citizens are identified. When solution is notified to the countries, each one will examine it performing a review if secure/trusted).

Pan-European eID

Notice once again that identity is not just authentication, it is **authentication + 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).

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.

eIDAS terminology

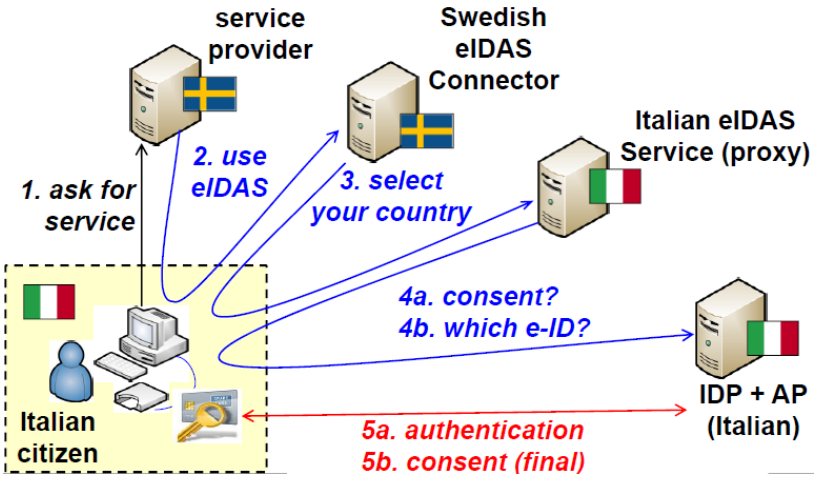
The **MS** is the **Member State**. It can be:

- **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.

The 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.).

Some services eIDAS-enabled

- **L1 (LoA substantial)**
  - (EU) https://europa.eu/europass/en
  - Authentication to europass, to create a European CV. It is possible to use SPID or whatever else, even at the lowest level (e.g., username and password)
- **At least L2 (LoA medium)**
  - (Belgium, tax) https://finance.belgium.be/en/about_fps/structure_and_services/general_administrations/taxation
  - (Sweden, tax) https://skatteverket.se
  - To access Belgium/Sweden tax service it is needed a LoA medium
- **At least L3 (LoA high)**
  - (Austria, something) https://www.oesterreich.gv.at/
  - This unknown service is accessible only through a LoA high

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**

Every country must support a minimal data-set. It must be supported by any eIDAS node for cross-border authentication:

- **8 attributes for natural persons**
  - (mandatory) *PersonIdentifier, FirstName, FamilyName, DateOfBirth*
  - (optional) *BirthName, PlaceOfBirth, CurrentAddress, Gender*
- **10 attributes for legal persons** *(OpenID Connect is not doing that)*
  - (mandatory) *LegalName, LegalPersonIdentifier*
  - (optional) *LegalAddress, VATRegistration, TaxReference, BusinessCodes, LEI, EORI, SEED, SIC*

The **security requirements** are:

- SAML request (no personal data) **MUST** be signed with digital signature, but no encryption required
- The request may be transmitted via:
  - *HTTP redirect* (i.e., 302 GET with request in a parameter): preferred if the dimension of the request does not exceed the max URI length (256 characters)
  - *HTTP post* (i.e., form with POST and request in a hidden field): if the request is much bigger.
  - Dimension can change since the request contains what is requested (authentication) and some attributes. If all attributes are requested, then the request will become huge.
- SAML response (containing personal data) **MUST** be signed with digital signature and will contain an *EncryptedAssertion* with one *AuthenticationStatement* and one *AttributeStatement*. Response is not encrypted, but the content is (remember that in SAML is possible to encrypt only parts).
- The response is transmitted via *POST binding* to the ACS (*Assertion Consumer Service*) of the connector
- The connector metadata contain (among others) the **encryption certificate** and **ACS URI.** If, e.g., looking the last picture, Italy needs send back something encrypted it must know what is that public certificate and must know where to send it, because at the end, after performing authentication, it must perform a redirect where it is required to know the ACS URI and the public certificate for that node. Those are metadata information.

All channels must be **TLS** version ≥ **1.2**, with qualified web certificates. The 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 are:

- For ECDH keys the minimum 256 bit, while for DH keys the minimum is 2048 bit.
- 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., unsupport that extension)

Requirements for SAML:

- Data encryption only via *AES-128/256-GCM*
    - MAY use *AES-192-GCM*
- Key encryption with *RSA-OAEP-MGF1P* or *RSA-OAEP* (3072 bits minimum key length)
- Key agreement via *ECDH-ES*
    - ES = **Ephemeral-Static mode** (i.e., recipient has static DH parameters while sender creates ephemeral ones). The destination publishes in the metadata what are its own DH parameters 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, BrainpoolP512r1, NIST Curve P-256, NIST Curve P-384, NIST Curve P-521*

Trusted computing and remote attestation

Baseline computer system protection

If possible, attackers try to attack our systems in the very base foundation, at the lowest possible level. This means that if they can modify the operating system, then everything that is layered on top of it such as anti-malware or firewall will become useless, since the OS is entirely compromised. If it is not possible to compromise the OS but there is physical access to the system, then it is possible to try boot an alternative OS.

Lot of systems nowadays have the possibility to perform network boot. If it is not possible to have physical access, but the attacker was able to provide to the victim a computer which has the double option (local and network boot), it will first try to boot via network and, if it fails, only then will boot from the local operating system. If the attacker can supply such a computer, and later it is able to act in the same network without physical access to your computer, then it will be able to set up an operating system to be loaded from the network, and the victim will not notice that.

Or even, given physical access, an attacker can also modify the boot sequence or the boot loader if it is able to supply to the victim a *firmware update* that is wrong.

Due to these problems, the **boot system** and then the **OS** must be protected.

In the past, there was the **BIOS** (*Basic Input Output System*), that was the standard firmware initially for personal computers but now available in most systems (also servers have inherited the BIOS). The BIOS is very **difficult to protect** since it is possible to just use a *password*, but most users don't know this possibility or don't want to use a password because every time there is a boot/reboot, it must be inserted.

In the last few years, BIOS has been replaced with UEFI, which is taking care of security, in the sense that it offers **native support for signature of the firmware**, so it checks if a firmware is good or not and will automatically perform the verification. After that UEFI has verified that the firmware is ok, then the boot loader (which is part of the firmware) can verify the OS before loading it.

Starting from here, it is needed to create a **chain of trust**, starting from these two points up to the upper layers.

Self-verification of firmware (example by HP Enterprise)

Most of systems nowadays have available some sort of self-verification of the firmware. It is reported here a specific example, because UEFI provides the generic support, but each company must implement it in a specific way.

The example is about HPE, which is particularly careful about security. Any system that HPE sells nowadays, inside the *flash ROM* that contains the firmware of the board, has a region labeled as "**signature**", and that is at a **fixed location** in the BIOS image (16 MB). It is important that this region is **fixed** location because firmware may change in size (as time passes) or between different machines. Since it is needed to minimize the complexity of the fixed part, it is much better anytime that you can have the fixed place where you need to look for the signature.

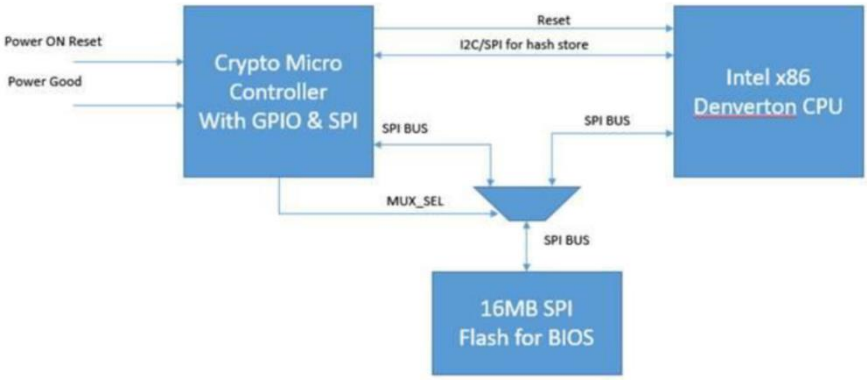
When a new image (e.g., update) is created, the **SHA256** hash of specific BIOS regions is computed (e.g., the signature itself must not be computed, so it must be excluded from the computation). Basically, the hash is computed over all the regions that include usually *static code*, the *BIOS version information* and *microcode*. Once the hash has been computed, it is **sent to a specific ultra-protected HPE signing server** which contains the private key protected with HSM of HP, that will return a **signed hash image** (*32 bytes + signature + certificate size*), and then it is **copied into the signature region**.

Note: HPE signing server has a critical role. The private key **must be** absolutely protected, because inside the firmware there will be the corresponding public key and if the private key is compromised, then it is needed to change all the firmwares in the world corresponding to HP, but physically (not logically, since the key is not trusted anymore).

The signed firmware with the signature region successfully filled is now powered on. At this point there is the **early BIOS**, which is a **fixed portion** of the BIOS, not in the flash ROM (so it cannot be changed), that is the **first firmware** being **executed** by physical arrangement. This portion cannot be changed unless the board is compromised (possible in theory but it is rather complex). It **cannot be overwritten** by an update because it is in a physical ROM (not in a flash). That portion of the code (the early BIOS) calculates the **combined hash** of each of the specified valid regions in the BIOS image. After verifying that the signature region contents are valid (verifying → using HP public key embedded in the BIOS early code) the BIOS compares the *stored* and the *calculated* hash: if the two values are equal, then the boot continues; otherwise, **halt the system**. This is typically named **secure boot**. Secure boot means a cryptographic verification of the boot portion and if it fails, the system **does not start**. In theory, it is possible to substitute the ROM containing the early BIOS.

To have additional security, it is possible to use a **hardware root of trust** which is external to the entire system.

In the picture there is the *Flash for BIOS*, the normal CPU (Intel x86) but the important part is the **cryptographic micro controller** which has got just basic interfaces (typically the serial peripheral interface) to talk with the flash ROM hosted on that kind of bus.



The Crypto receives the *Power ON Reset* and the information that *Power is Good*.

Self-verification is based on the firmware itself (static portion of the firmware verifies the parts that can be updated), but verification of the firmware **can be implemented by an external chip** as well (the Cryptographic Micro controller).

The external crypto chip validates the **whole** BIOS (not only the parts that can be updated) in SPI flash just after power on. It means that the computation is not performed by the normal CPU with the static portion of the BIOS, but it is computed by a program which is self-hosted inside a microcontroller.

Once the validation is successful, the CPU will be out of reset state, otherwise it will remain in the reset state and system will not start. This microcontroller has a **fusing option** because it is needed to fuse **one public key hash** (not the real Public Key but the hash), since it will be used to verify the signature of hash file stored in the signature region. It means that a generic microcontroller is bought and then it is customized with the hash of the public key. Again, if the corresponding private key is compromised then it is needed to be replaced (once it has been fused there is no other way to change the hash of the trusted public key).

Validation flow is like BIOS integrity check by BIOS (called self-verification), except for the fact that an external chip is doing the validation, which makes it the real **hardware root of trust**.

The point is that we're striving nowadays to minimize the part of the system that must be trusted. It is tried to make it as small as possible because it will be easier to design, verify, check. This can be attacked if someone is able to provide a fake microcontroller or substituting it.

Secure boot stops at this point. If the firmware has been manipulated, then it will not start. Then, inside the firmware, there is a **bootloader** which can *optionally* verify the signature of the OS. Nowadays, both Windows and Linux have options to sign the kernel of the OS. If kernel has been modified, it is not loaded and executed.

On the contrary, we would like to extend this chain of trust from the hardware up to the applications.

Trusted Computing

A trusted component or a trusted platform is one that **behaves as expected**, which does not mean that is absolutely good or secure. It only means there has been no modification with respect to what was programmed previously (if program contains a bug, it will only mean that it has not been injected by malicious actors but from the programmer itself).

To achieve that, an **attestation** is performed, which is evidence of what is the **current state of the platform** that can be verified by someone else. With *state* it is mean the **software state**: all the running applications + configurations.

Some foundations are needed, which is the **Root of Trust** (*RoT*). It is something that must be trusted (for example, it is provided by the secure boot). The starting point can be the microcontroller or the firmware itself when it is performing self-verification and we build up on that.

The Trusted Computing approach defines schemas for establishing trust in a platform based on identifying its hardware and software components.

More specifically, the **TPM (Trusted Platform Module)** provides methods for **collecting** and **reporting** these identities: a TPM is not something that will stop the system from operating, but it is something that cannot be faked, and which will be able to provide **undeniable evidence** that system is in a certain state. It is a **Trusted Reporter**. A TPM used in a computer system **reports** on the hardware and software in a way that allows determination of expected behavior and, from that expectation, establishment of trust.

Trusted Computing Base (TCB)

TCB is a *collection of system resources* (hardware and software) that is responsible for maintaining the security policy of the system. An important attribute is that it can prevent itself from being compromised by any hardware or software that is not part of the TCB. The TCB is self-protecting itself against other parts, for example, both self-verification and external hardware RoT are TCB, since they cannot be modified in anyway by any hardware or software (excluding physical manipulation, in that case attacks are feasible).

The TPM is **not** the TCB of a system. Rather, a TPM is a component that allows an independent entity (typically external to the system, called *external verifier*) to determine if the TCB has been compromised. In some uses, rarely adopted, the TPM can help prevent the system from starting if the TCB cannot be properly instantiated.

Rather than having that kind of secure boot displayed before, with TPM is possible to have all the BIOS and boot sequence that is verified by the TPM and then the TPM can provide a flag (good/bad) and it is needed some physical action so that if the result of the verification is bad, the system is halted (not so typical, only few systems work in that way). For halting a system usually secure boot is used, when TPM is used the result is given not to the local system itself but to an external verifier, which will then take appropriate actions.

Root of Trust

It is a component that **must** always behave in the expected manner because if it is misbehaving it cannot be detected: it is the building block for establishing trust in a platform. Typically, the RoT has some hardware component and then later it can also include some software parts.

There are different RoTs in a trusted computing environment:

- **Root of Trust for Measurement (RTM):** measurement means computing values that tell if system is good or not. It measures and sends its integrity measurements to the RTS (another RoT). Usually the CPU executes the CRTM (*Core Root of Trust for Measurement*) which is a software component.
- **Root of Trust for Storage (RTS)**: is a special portion of memory that is shielded/secured. Shielded means that no other entities but the CRTM can modify the value.
- **Root of Trust for Reporting (RTR)**: an entity that securely reports the content of RTS.

Basically, there is the RTM, which is computing the measure, then it is securely stored in the RTS and then when it is needed, the RTR will ask for the measure and they will provide it to an external verifier.

The TPM is typically the RTS+RTR: it is a secure storage, and it is also a trusted component for reporting. It means that it will not report anything else but what is written inside the secure storage. The CRTM is still needed and that's why the TPM is executed in combination with a secure boot, because it will ensure that the system has installed correctly the CRTM and from that it will continuously measure and send results to the hardware component.

Chain of trust

In general, there is a component *A* that measures component *B* and once these measures have been performed, *A* stores the result of them in the RTS. Then component *B* will do the same tasks with another component *C*, storing the results.

Then, with all those measures, it is possible to ask RTR the measurement stored by *B* and *C* from the RTS. If component *A* is trustworthy, then the verifier knows if B and C are good or not, because the expected hash value is known (if it is equal is good, it fails otherwise).

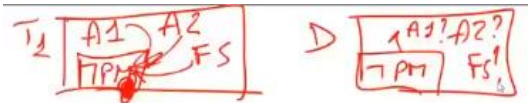
TPM overview

It is a cheap component, less than one dollar usually, and it is available on most servers, laptops, and PC. It's **tamper-resistant**, but not *tamper-proof*. Tamper proof means that it cannot be attacked, while tamper-resistant means that it tries to resist various kinds of tampering. Although TPM contains cryptographic modules, it is **not** a high-speed cryptographic engine (it is rather slow). The important point is that it is certified **Common Criteria EAL4+,** which is quite a high level.

It is a **passive component**, which means that it does not take the control itself of the computer, but it must be driven by the CPU. For this reason, it **cannot prevent boot** (because it is out of the boot sequence), but it can be used for data protection. So, CPU which is executing the core of RTM can ask to the TPM to store values.

TPM features

- It contains **hardware random number generator** (not a pseudo RNG)
- **Secure generation of cryptographic keys for limited uses**, so not generic keys for any use
- TPM can be used for **remote attestation**: it is used to store the **hash summary** of the hardware and software configuration and then a third party can verify that the software has not been changed
- TPM can perform **binding** (data encrypted using the TPM bind key, a specific key inside the component, cannot be decrypted outside that specific TPM, because it is a unique RSA key descending from a storage key). It is a good solution because even if data are stolen, there is no way to decrypt it. It is also a bad solution because if it is needed to export data to move to another machine, a complex procedure is required.
- **Sealing**: it is an additional level of security, in which not only the data are encrypted with a key that is internal to the TPM, but as part of the decryption operation, the operation requires the TPM state to be the **same as when data were encrypted**. Remembering that the state is the collections of all application running + configuration files. Looking the picture, there is a TPM and in a certain moment T_1 there are the applications A_1 and A_2 and the FS contained the configuration. All these things are known to the TPM and an element inside TPM is protected. In the moment in which it is tried a decryption, before allowing it the TPM will ask if A_1 and A_2 are running and if the system is in the same state. If the answer is no, then the TPM will refuse to perform the decrypt operation.

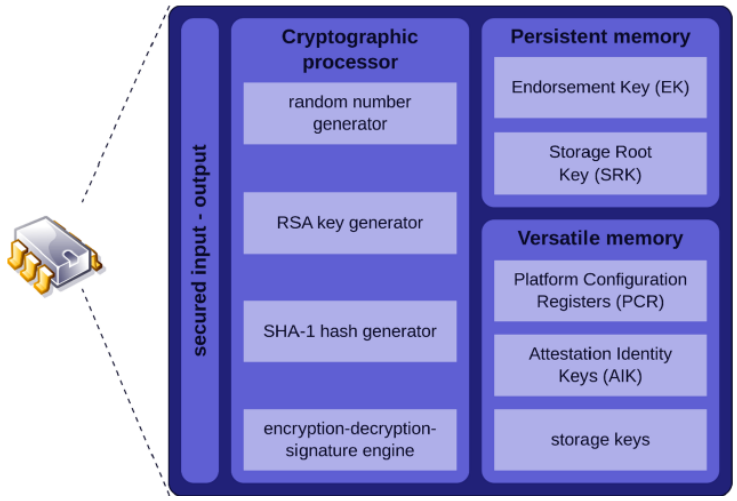


- Computers can use TPM to **authenticate hardware devices**, since each TPM chip has a unique and secret **Endorsement KEY (EK)** burned when it was produced. Again, it is good because it is possible to clearly identify that machine, but for the same reason there is no privacy, so it must be ensured that this feature is used by authorized people.

TPM-1.2

There are two important versions of the TPM, 1.2 and 2.0. The first one was the most widely used until few years ago and it is rather unflexible because it contains:

- **fixed set of algorithms** (*SHA-1* for computing the hash, *RSA* for signature, verification, and encryption of keys, optionally *AES*) to simplify the system.
- one storage hierarchy for platform user
- **one root key**, named *Storage Root Key* (SRK, RSA-2048)
- the hardware identity is built-in with **Endorsement Key** (EK)
- **sealing** (having data being able to be decrypted only with a certain state) was tied to PCR value, where PCR are special registers inside the TPM.



In the picture are shown the basic features: the *random number generator*, the *RSA key generator*, then the computation of *SHA-1* and the *encryption-decryption-signature engine* (typically with RSA, that is the minimum). Then it has a **secure memory**: the *Endorsement Key* and the *Storage Root Key* (RSA keys). In the Versatile memory, which means that can be used for other purposes, there are Platform Configuration Registers (PCRs) which record what is the current configuration of the system, Attestation Identity Keys (AIK) which is the key used for signature of the external reports to perform the remote attestation, and some storage for various keys that may be needed during the operations.

TPM-2.0

It is a big improvement because it provides **cryptographic agility**. For backward compatibility continues to have *SHA-1* but it also offers *SHA-256*. It has got RSA for backward compatibility, but it can also perform signature, verification, and encryption with ECC-256. It contains as native features HMAC based on SHA-1 or SHA-256 and as minimum AES-128 plus other things that are optional. Rather than having fixed keys it has got three key hierarchies:

- **Platform**: for data coming from outside, from the platform which is hosting the TPM
- **Storage**: for internal storage
- **Endorsement**: providing information to the outside

Each hierarchy may have **multiple keys** and **multiple algorithms**. When you need to change anything in the TPM there are **policy-based authorization**, it means that there is not only one password, maybe there is also two-factor authorization, or a fingerprint (in TPM 1.2 there was one password that can change anything in the TPM). Moreover, there are platform-specific specifications for **PC client** that was the original application, but also *mobile* and *automotive*, because in those environments trust is strongly needed.

Implementations of TPM-2.0

TPM-2.0 can be:

- **Discrete TPM (dedicated chip)** – which implements TPM functionality in its own tamper resistant semiconductor package (manufacturer STM, Infineon).
- **Integrated TPM (part of another chip)** – it means that the hardware of the TPM is embedded as part of another chip. In this case the TPM is **not** required to implement tamper resistant. Intel has integrated TPMs in some of its chipsets.

- **Firmware TPM (software-only solution)** – it is needed to run the firmware corresponding to the TPM in a CPU's **trusted execution environment** (*TEE*). AMD, Intel, and Qualcomm have implemented firmware TPM.
- **Hypervisor TPM (virtual TPM provided by a hypervisor)** - it runs in an isolated execution environment (its security can be compared to a firmware TPM).
- **Software TPM (software emulator of TPM)** – some software running in user space, software emulator of TPM useful for development purposes. There is not real security.

These different solutions lead to different *trusted computing base*, because in Discrete TPM there is separate hardware RoT (separate component), while in Integrated TPM it the RoT is Intel, it is trusted that it has correctly implemented and not altered the TPM, in Firmware TPM the software must be trusted itself (e.g., the secure boot) and in the Hypervisor TPM is even worse, because the user doesn't know anything of what is happening below the virtual machine. There is **no solution**, it depends on what is accepted as Trusted Computing Base (TCB).

TPM-2.0 three hierarchies

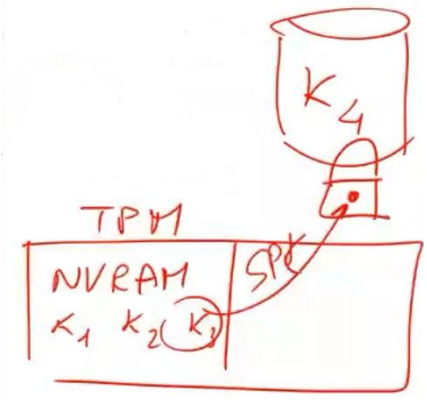
- **Platform Hierarchy**: platform means the board or the system, which is hosting the TPM, which is using the TPM as RTS and RTR. So, the platform hierarchy is to be used for the platform's firmware. It contains *non-volatile storage*, *keys* and *data* related to the platform.
- **Endorsement Hierarchy**: it is used by the privacy administrator for storing *keys* and *data* related to the privacy.
- **Storage Hierarchy**: used by the platform's owner which is usually also the privacy administrator and that contains *non-volatile storage*, *keys*, *and data*.

Each hierarchy has a **dedicated authorization** (*password* as a minimum) and a **policy** (can be very simple or complex). Each hierarchy has also different seed for generating the primary keys. So, keys of the various hierarchies are unrelated.

Using a TPM for securely storing data

We have the platform that can use the TPM as a secure storage with:

- **Physical isolation**: storage is in the TPM, particularly in the Non-Volatile RAM. Here, are typically placed *primary keys* and the *permanent keys*. There is very **limited space**, so it is important to choose what to store inside. Moreover, for those keys there is **Mandatory Access Control**, so they cannot be unprotected easily.
- **Cryptographic isolation**: in the Non-Volatile RAM can be placed a limited number of keys (so, the keys are placed inside the TPM) and this is physical isolation. In Cryptographic isolation, the key is outside, stored in a disk, but then the key is encrypted with one key which is inside the NV-Ram. The actual storage is outside the TPM. In this way can be protected external **keys** or even **data**. But, for the TPM that is a **blob,** a bunch of bytes, which are encrypted by the TPM, so encrypted with the key which is hosted inside the TPM. There is also Mandatory Access Control, but the main point is that while physical isolation provides limited space, since it is internal to the TPM, with Cryptographic isolation there is unlimited space (from the point of view of the TPM), the only limit is the external memory. Of course, if it wanted to decrypt the external data, you need this specific TPM (not another one), you it is not easily possible to migrate the data to another platform.



TPM objects

These are the objects that are managed by the TPM:

- **Primary keys**: in particular, *endorsement keys* and *storage keys*. They are derived from one of the primary seeds; the TPM does **never return the private value**, so the private keys are inside the TPM, and it is not possible to extract them. They can be **recreated** using the same parameters assuming that the primary seed has not been changed. So even if the private key is destroyed or lost, it can be recreated using the same seed and the same parameters.
- **Keys and sealed data objects (SDO)**: they are protected by a *Storage Parent Key* (SPK). The SPK is needed in the TPM to load or create a key or SDO. Randomness for these keys come from the TPM RNG which is internal to the TPM. The TPM returns the private part of these keys protected by the SPK, so the private part needs to be stored externally somewhere, but then it is needed the TPM, that SPK to decrypt it.

TPM object's area

- **Public area** – it is used to **uniquely identify an object**, even if there are no special permissions it is possible to list what are the objects stored inside.
- **Private area** – contains the object's secrets and exists only inside the TPM.
- **Sensitive area** – it is an encrypted private area used for storage outside the TPM. Not compulsory.

Resuming, an object has parts inside TPM: the public and private area. On the contrary, the sensitive one is external. You are not obliged to have the sensitive area, while the other two are mandatory.

TPM Platform Configuration Register (PCR)

The TPM can report the current state of the system. To store the current state and to report it, the TPM contains a special set of registers named **PCR**. They are registers that keep the history of the platform configuration. The set of PCRs is the core mechanism for recording platform integrity. These registers are reset only at **platform reset** (after a reboot or after a hardware signal). Reset means that all the registers will start with 0. This is important, because even if malicious code infected the system, it cannot have the registers back to some value.

These registers have only two operations: one is the reset, the second one is **extend**. The extend operation is:

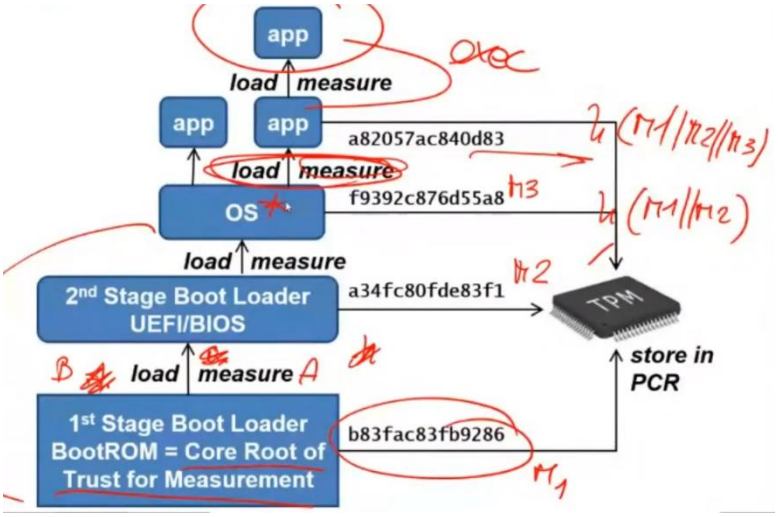
$$PCR_{new} = hash(PCR_{old} || digest\_of\_new\_data)$$

The old value contained in the PCR, concatenated to the digest of some new data, is hashed. The result becomes the new value of the PCR. This is the only operations that can be performed. This operation is designed for trusted computing.

The value of the PCR can be used to gate (to control) access to other TPM objects. For example, in Windows, BitLocker can be activated, that is disk encryption. To activate it, TPM must be enabled, because BitLocker is *sealing* the disk encryption key to PCR values. It means that the key needed to decrypt the disk is available only when Windows is booted, and it has not been modified. If BitLocker is activated, the key is stored inside the TPM and if the system is not in a specific state, and the state is not coincident with the STATE of values present in the PCR, the decryption will not work (binding + sealing).

Measured boot

To implement those capabilities, it is performed not secure boot but measured boot.



- The first stage of the boot loader, that it is assumed to be trusted, constitutes the **core root for measurement** and that value is stored in the PCR. It is verified (computed) what is the hash of the core root for measurement.
- This boot loader will first measure (compute the hash of the second stage boot loader UEFI/BIOS) and then load and execute it. The corresponding value will be stored in the PCR through the Extend Operation. The second stage will store in the PCR $h(M1 \,||\, M2)$, so the hash of measure1 concatenated with measure2.
- The second Stage Boot Loader first will measure the operating system and then will load it for execution and will extend. So, after this step there will be $h(M1 \,||\, M2 \,||\, M3)$. The PCR is working as an accumulator, but not as a simple sum, but with the hash of the hash of the hash and so on. The final values depend upon all the hash values that were inserted. Not only values but also the sequence.
- Finally, the operating system can be modified in such a way that also the OS does the same thing. Anytime it is decided to start any application, it is measured, loaded and the value is extended into the PCR. This is optional, e.g., Windows does not perform that, while in Linux there is a special module which is able to do that. Depending on the implementation inside the PCR there will be the whole history of what has been executed on the system starting from the boot. That is the **state**: when we say that the system is in this state, it means that the system has executed this sequence of applications. Then, an app could load and measure another app and so on.

We said that we would modify the OS to perform these operations. But then, given the picture, then it seems that is needed to also modify the application when it's the application which is starting a new application, that would be complex. This is not true, since it is executed with a specific system call of the kernel for executing a binary, so even if an app starts another app, it is starting through the exec system call, which is inside the OS. This means that it is enough that to modify the OS and not all the applications.

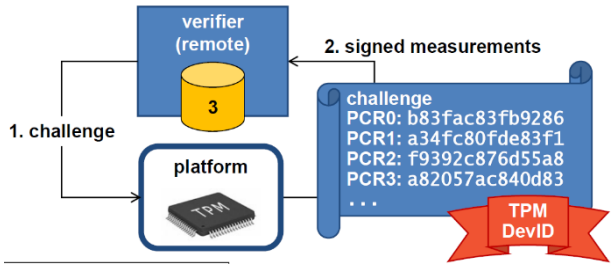
Remote attestation

Until now we have not performed any comparison between the values in the PCRs and some expected values, because that comparison would be executed locally. But if the system has been manipulated, then that comparison would be wrong. So, it's very difficult to use TPM for self-control of the system. It must be used some external party, and this is the purpose of remote attestation.

In remote attestation there is one platform with the TPM and an external verifier, typically remote (via network):

- When the verifier wants to know if the platform is good or not, it will send a **challenge**.
- As a response to the challenge, the TPM will read all the values present in the PCRs and this list of value is digitally **signed** with **TPM DevID** (*device identifier*). In this way, it's possible to know that these values have not been tampered. These are values created by this specific TPM and for this specific challenge. A replay attack cannot be performed (the response depends on the challenge). The TPM with its internal RSA/EC key is creating a signature over the challenge and the values present in this moment in all the PCRs.

- The remote verifier performs **validation** in two steps:
    - First it verifies the signature cryptographically: crypto + ID. There is also the ID because there must be somewhere a table that says that the node with identifier *ID1* has got the public key 1. This means that if a challenge is sent to a device with ID1 the answer will be signed with that key.
    - Then, it will **check measurements** against Reference Measurements also called *golden values*, that is what are possible values that are already known.

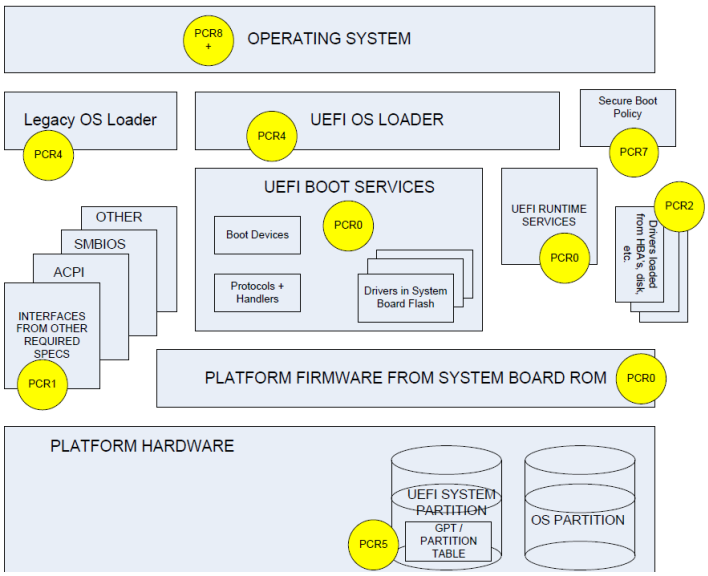


The question now is: "How do we know if the PCRs values are good or not?". These values depend not only on the software being executed but also upon the sequence, so how is it possible to know which are the good values, since we need to perform a comparison.

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.
- *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

Linux's IMA

The OS can be extended to perform the same kind of operations that are done with the boot sequence. In this way the Root of Trust can be extended up to the application, but that depends upon the OS. Windows does not have it, but Linux has got a module that can be added and activated.

IMA is **Integrity Measurement Architecture**, which is an additional component inside Linux that perform various operations:

- **Collect**: every time the kernel is invoked for an exec, before running the binary the IMA measures it.
- **Store** – after it measures, it stores inside one of the PCRs 8 to 15. So, It adds the measurement to a kernel resident list and extend the IMA PCR.
- **Appraise** (*optionally*): enforce local validation of a measurement against a "good" value stored in an extended attribute of the file. In certain file system of Linux, the files in addition to the standard attributes (mod, user, group), have extended attributes. If that version is activated, then it is possible to store what should be the good value. So, when the binary is executed, its hash should be that one, if it is not, do not execute it. Binary can be changed and so that value. Appraise is an option, but the fact that those data have not been modified must be trusted.
- **Protect**: protect a file's security extended attributes (including appraisal hash) against **off-line attacks**. If someone modified the extended attributes by starting an application that permits to modify it, it was successful, but a new application has been executed and it will be detected. Imagine unplugging the disk when the system is switched off, attach it to another computer, change the hash value and finally put the disk back: this is an *offline attack*. IMA computes an HMAC with the secret key, so that if someone tries to recompute the hash it will not have the secret key and the hash will not match.

The part related to Collect and Store are for sure correct because their values are collected by a component which has been verified and they are stored in the PCR. Appraise is a riskier action.

Linux's IMA details

IMA is extending the UEFI measured boot to the OS and the applications. Linux IMA uses PCR10 to store everything, and it initializes PCR10 with the first measurement of the *boot_aggregate*, which means that in the moment in which it starts, it computes the hash of all TPM's PCR from 0 to 7 which report if the boot was performed correctly or not. What IMA should measure is defined by the **IMA template**, mostly of times it is used the *ima-ng* but it can be customized. All the measurements performed by IMA are listed in the kernel's *securityfs* at */sys/kernel/security/ima/ascii_runtime_measurements* that contains a table like the following one.

PCR	template-hash	template	filedata-hash	filename-hint
10	91f34b5[…]ab1e127	ima-ng	sha1:1801e1b[…]4eaf6b3	boot_aggregate
10	8b16832[…]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

It shows the PCR number, that had *template-hash* original value, then it was updated following that *template* with the *filedata-hash* value and the name *filename-hint*. It also lists the various components that are being executed and extended. The table is very important because by looking only at the value in the *PCR10* there's no way to know if the value is correct or not, so the table is needed because it tells which commands and in which order are executed.

Dynamic Root of Trust for Measurement

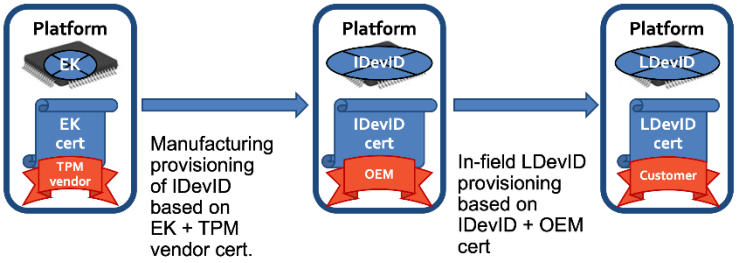
The DRTM provides a **special processor command**: *SINIT* for Intel CPU or *SKINIT* for AMD CPU. It stops all processing on the platform and then **hashes the content of a specific memory region** and then stores the measurements in a dynamic PCR. Then it transfers control to a specific location in memory which is also called **Late Launch**. Previously, the measures were all about the file present in the ROM/disk before loading it, but we never considered the fact that once the file is in memory, someone could change it. Using this technique,

it is possible to compute the hash of a memory zone, that is a RAM area, and then once the measure has been performed it is possible to activate a program and verify if that value is good or not. It can be used periodically to check what is happening inside a specific zone of the RAM (typically those that concern with the kernel).

Credentials chain of trust

There are several keys which are not completely independent one from the other, but they have some chain (consequence). That is starting **from the TPM vendor** that created the chip or the firmware **to the customer**. That is important to identify the devices present on a network and there's also a standard *IEEE 802.1AR – Secure Device Identity using TPM* that permits *Zero-Touch management* of a platform. That means that if it is needed to substitute a switch, it is possible to configure it automatically by connecting it to the infrastructure. Note that the new switch needs to be sent by the person in charge of managing this, not because the switch needs to be configured, but because its private key and public key are needed.

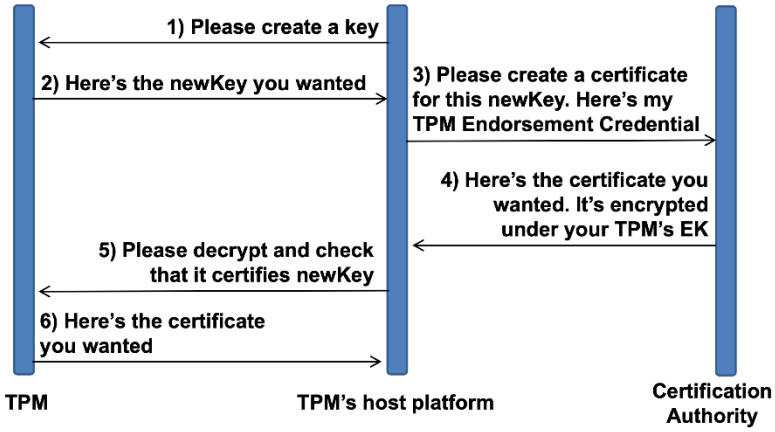
Everytime the equipment (that have a TPM) is bought, the first thing is to put in the inventory the device with its public key. Then, when the device is attached to the network in any place in the company it will be automatically recognized.



When a platform is bought (computers, mobile, automotive), it contains one key that was already generated, called the *endorsement key*, and to be sure that the key is a genuine TPM the platform contains a certificate for that key signed by the TPM vendor. When the platform is provided to the customer, it is possible to place an **Initial DevID,** *IDevID*, which is based on the endorsement key plus the certificate provided by the vendor. Now there will be a new key (*IDevID*) with a certificate signed by the OEM, *Other Equipment Manufacturer* (e.g., someone that has taken the board and inserted in its own PC). When the device is then used in a specific environment, we don't want to be traced so we need to change the identifier. For this reason, we will create a **local device identifier** (*LDevID*) built based on the initial device identification plus the OEM cert, this time signed by the customer. This is the **chain of trust**: the initial certification tells that it is a good TPM really created by that manufacturer, the second certification tells that it is a good device and finally the third certificate says that the device is a registered component of a certain infrastructure and avoid being traced the *LDevID* will be used in all network operations. There is no possibility to go from the LDevID to the IdevID and from the IDevID to the EK.

TPM-2.0 Make/Activate Credentials

It is now reported the procedure to make or active credentials inside the TPM, the ones needed to administer it. The TPM is placed on the host platform, so it requires the TPM to create a key. The TPM answers with the public part of the requested key. Now the platform goes to the CA giving it the new key and its *TPM Endorsement Credential*. The CA will create the certificate **encrypted** with the TPM's endorsement key. It is encrypted because it is needed a way to check if the TPM really possesses the EK. For that reason, the certificate is then sent to the TPM to check if it certifies the new key. Finally, the TPM returns to the host platform the certificate. This is a **POP** (*Proof of Possession*), which means that the certificate is given to someone that can demonstrate the ownership of the corresponding key (remember that here there's no signature). In PKCS#10 there's an actor which asks for a certificate that



performs a signature, but since here in the request phase there is no key, the operation with the private key is placed in the distribution of the certificate: if the user doesn't have the private key, then it will not be able to have the certificate (and to use that key).

TPM basic authorization mechanism

It is needed a way to protect the configuration of the TPM:

- It is possible to have **direct password-based authorization**, valid for single commands to be executed.
- Another way is **password-based** HMAC to authenticate commands and responses with *caller_nonce* and *TPM_nonce* to prevent replay, useful when a program is communicating to the TPM so the commands that are sent to the TPM can be protected with HMAC and nonce.

These methods are quite simple and trivial, but the TPM knows a lot about the platform states and can be configured to:

- *prevent object usage unless selected PCRs have specific values*: even if the password is known or if an HMAC command is received, if the PCRs have not the specific values the operation will not be performed
- *prevent object usage after a specific time*
- *prevent object usage unless authorized by multiple entities* (e.g., different passwords and users).

Which is your trust perimeter?

We said that we are measuring the binaries being executed and the configuration files, this is somehow deciding what is our **trust perimeter**. If we verify the hash at installation or downloading time, it is possible to check the signature, or it is possible to perform the verification every time a binary is loaded for execution.

What does "executable" mean? For example, if a python program is started, what will be measured/executed in that moment? The measurement can be done only when there's the **exec system call**, for a command like ***python x.py*** the exec system call is executed only to execute *python* but not for *x.py*. In the moment in which the command is executed, the interpreted is not cared. If the interpreter has been modified the system will perform badly, but the important is that if *x.py* has been modified, then the result will be different. This means that **load time** verification is good if we're running pure binaries, but if we're running an interpreter, then another IMA template must be used, that will measure not only the binaries but also the configurations and scripts passed to an interpreter.

But what if any browser is started? It is possible to measure the browser, but when it's executed and used for internet navigation it will get *Javascript* and it is not possible to measure something coming from the network, which is not stored on disk but directly executed. The solution is enhancing the security of the system but it's not the perfect solution.

In the python case we need to change the template and ask IMA to measure not only the binaries but also any script which can be executed by the binaries and compare that with the value that is expected. On the contrary, when it is started something like a browser that contains an interpreter that executes something from the network, the only possibility is to hope that the browser's sandbox is behaving correctly and there's no bug so that if there's a threat it can only be executed in the sandbox.

Run time (components that change their behavior while running) components can be measured by measuring the configuration files (because that affects the ways in which components are running) but should be measured the **in-memory configuration**, adopting the technique of **dynamic root of trust for measurement** but that is not trivial, it means that it is needed an appropriate firmware or host system because that's not normal. The normal thing is to measure when a binary is loaded, or when a configuration is read, but once that is being executed if there's an in-memory attack it's very difficult to be detected and requires something specific inside the platform.

What about virtualization?

All the things discussed in so far work well with **physical system** that has got TPM either physical or in firmware. Nowadays, it is used the **virtualization** such as *containers, virtual machines,* or *virtualization inside virtualization* like many companies that first start VMWare and then Kubernetes. This leads to problems in **performing integrity monitoring of virtualized components** because the hardware root of trust is in the hardware, but one TPM will may virtualize 100 virtual machines. There is a *virtual boot system* (and no more a physical boot system), furthermore it is difficult to inspect the actions inside the virtual machine.

Integrity monitoring in v-environments

There have been some attempts:

- In general, using **containers** is better than using virtual machines for monitoring because in containers the kernel is the same for every container. Container's operation can be easily monitored by the host because when a container wants to execute something it will ask the *baseline host* to execute it, and if there is trusted computing in the baseline, there's no need to install it inside the containers.
- **Remote attestation for containers** is possible: IMA has been extended to measure not only the original operating system but to measure and distinguish what is happening inside each container running on top of that operating system.
- Another solution is **CoreOS trusted container**: it stores *container state* and *configuration* in the event log of the base TPM but only the container as a whole. It means that CoreOS TC does not cover modules executed inside a container, so there's no control on what is happening after starting the container.
- Intel has a similar technology named **Clear Containers** which uses *Intel virtualization technology* to **run a container as a virtual machine** but with better performances (than VMs). Intel only performs **load-time integrity of the container image**. The container in the repository must have a digital signature and it will be checked. Again, there's no control after starting the container.

Audit and forensic analysis

TPM is also good to perform audit and forensic analysis because when something bad happens in systems it is not possible to be sure that it was an attack (it could have been a bug). Thinking about IoT, an electronic control unit inside a car, when there's an accident who is guilty? For example, if the brake was not working because of a bug is it possible to demonstrate that? There are various cases in which there's an accident in which it is wanted to know what the software was being executed in that moment and its configuration.

We'll see in the future an increasing number of incidents that are caused or may have been caused by bugs or wrong components. Having something that will demonstrate what the configuration was is important.

More intelligence or computation is moved into the edge nodes or the network itself (think about software defined networking, NFV). When there's a network attack, the manager would like to know how the network was configured.

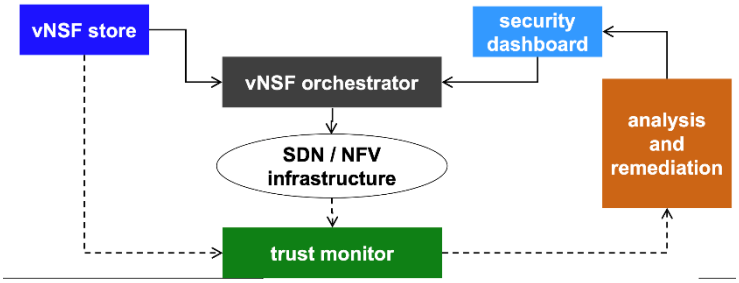
In general, the open questions are:

- *What was the system state at time T when attack happened?*
- *Considering a network, what was the network path that packets were following and what was the processing for user U at time T?* (Because given a MAC/IP address the packets can follow a path or another)

In the end, it is absolutely needed to be able to reconstruct an attack to rebuild the system in a way to be more resilient to other attacks. **Logs** are very important too, but it is needed to be careful if the hacker has modified them. A good solution is to have the TPM generating logs so that they are digitally signed and are given to an external entity, *the verifier*, so even if the machine is attacked, if the verifier has not been attacked there will be the evidence.

SHIELD project: trust monitor

This is a project where the TPM has been heavily used. SHIELD was a project related to an *SDN/NFV infrastructure* used to provide **SecAAS** (*Security as a service*). It means that since there is a network composed of SDN switches and NFV nodes (nodes able to host specific functions), it is possible to deploy inside the network specific security components as needed: firewalls, intrusion detection, VPN channels.



The point is that those things need security. There is the **virtual Network Security Function (vNSF) orchestrator** that decides which function must be placed and where. There is the **vNSF store** that says what component he wants and where. There's also a **security dashboard** for *monitoring* (because it can also be used an IDS or simply a network monitor to collect information) but here there is a trust monitor.

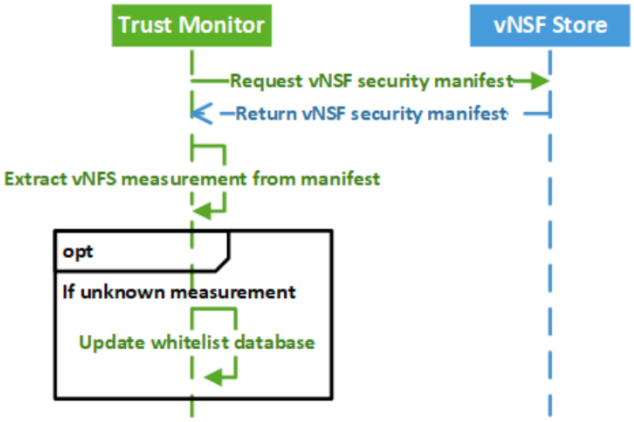
The **trust monitor** uses *remote attestation* to verify the integrity of the infrastructure, because since the infrastructure is software based if it is deployed a virtual firewall but the node hosting it is compromised, then that will not work. For this reason, it is periodically performed a **poll** and it is asked to each node to report its state and it is compared with the good measures. That is not only about the node itself, but also about the virtual functions deployed and their configurations. That is **measuring**: as soon as it is detected that a node is compromised, thanks to the SDN infrastructure that node is excluded from the network. In this way, that node becomes isolated, and someone will go and perform the investigation.

In this case the system needs to keep working, so a new virtual network function will be deployed in another node, that hopefully is not compromised.

Trust monitor performs **analysis and remediation**: when the trust monitor is detecting that a node is misbehaving, that information is provided to a human operator, but it is suggested also the default operation (disconnect the node and move that function to another node). Here monitoring through TPM and remote attestation is playing a vital part.

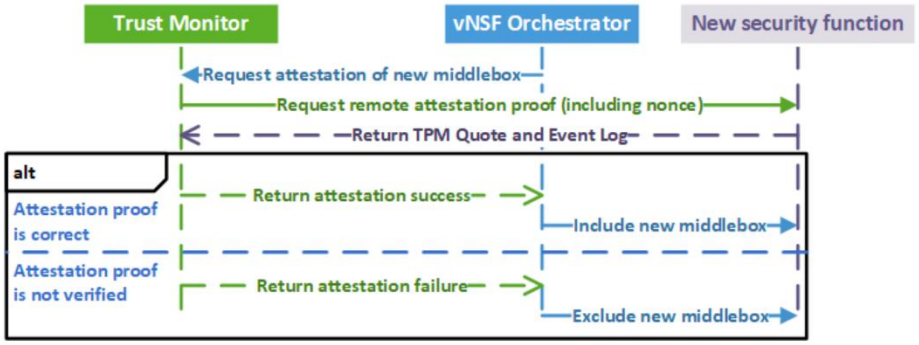
Golden value creation

We said that the TPM is reporting values that needs to be compared to good (golden) values. One important thing (in SHIELD but also in general) is the **golden value creation**. The trust monitor is initially going to the *vNSF store* containing all the virtual network security function, **requesting the manifest** (the list of the elements inside that virtual function) and then the measurements are extracted from the manifest and the whitelist database is **updated** (or created).

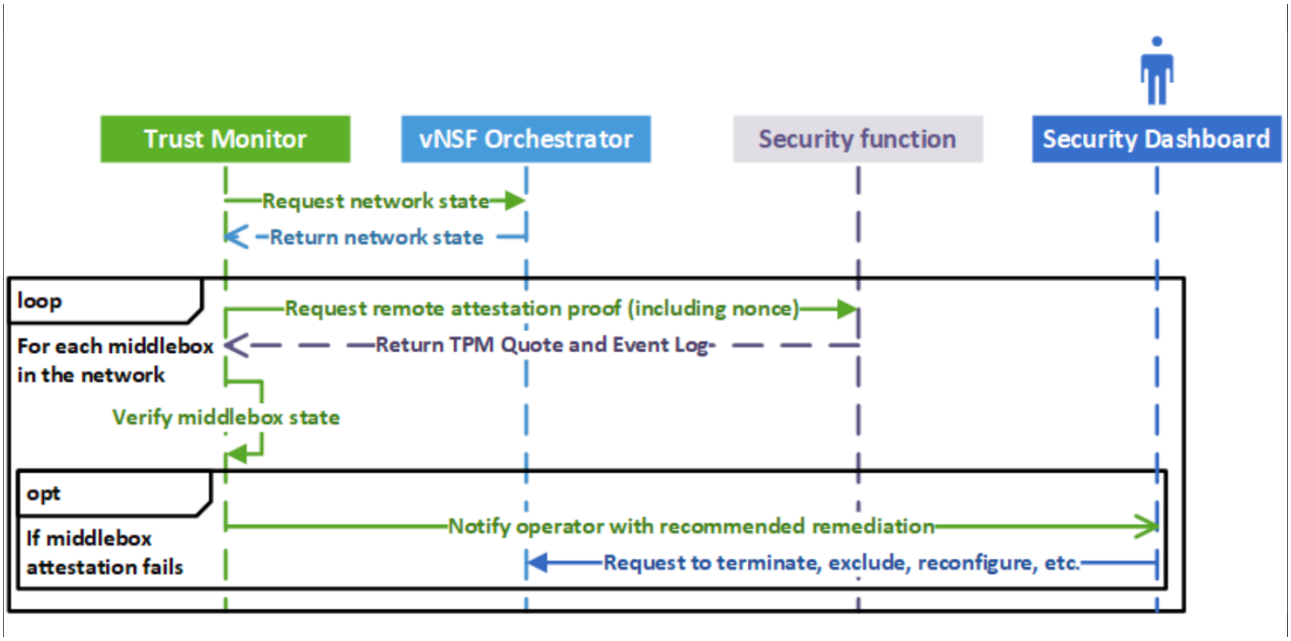


Initial deployment of a security function

The orchestrator, before deploying a new virtual function, will ask to the trust monitor: "Please, I want to deploy a virtual firewall on node number three, tell me if that node is in a good state". The trust monitor will request remote attestation, will receive the answer (the *TPM Quote* means the list of the PCRs signed by the TPM) and the *event log*. If the attestation proof is correct compared to the whitelist, then it returns *attestation success* and this new element is included. If attestation proof is not verified, then it is returned attestation fail and exclude the new middlebox.



Periodic attestation of security functions



The attack could happen after the element is adopted, so periodic attestation of all the deployed security functions needs to be performed. The trust monitor is **periodically querying the orchestrator** asking what the correct configuration of the network in such moment is. The orchestrator will answer with the state and then the trust monitor goes to the various security functions (there will be many on many nodes) and for each of them will be get the TPM Quote and the Event Log and will verify the middlebox state. Again, if the middlebox attestation fails will notify the operator with the recommended remediation like terminate, exclude, reconfigure and so on.

There is an initial phase with the creation of the whitelist, then there is a first check while deploying a new function, but there's also a periodic check. This part of periodic checking is the trickiest one, because given the fact that the TPM is slow it is not possible to perform an attestation every 10ms. Typically, it is possible to perform an attestation every 10s. This is the needed time to request the attestation, get the signature, transfer the data, compare it to the whitelist and so on.

This means that there is a window of exposure of 10s, so if the attack is fast enough to attack the node and then put it back in the original state in 10s it is not possible to detect or stop it.

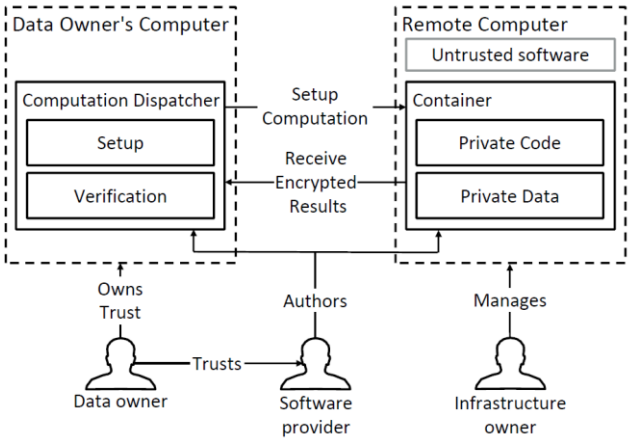
The application of this kind of things has some limitations and the time of window of exposure is one of them. Note that most of that time is due to the TPM signature because it is slow.

Intel SGX (Secure Guard Extensions)

We are going to talk about the secure remote computation problem and the proposed solutions to this problem by using the trusted computing. Then, there will be an overview of *Intel SGX* and just a bit of its internal workings. It's a **complicated technology** and a lot of parts of it are not well documented. Then, we'll also see how to attest software using Intel **SGX local and remote attestation**, some of the **attacks** that can be performed on the actual Intel SGX implementation, and then, a brief of SGX coding example.

Secure remote computation

It is wanted to execute some software on a remote computer. This computer is not under our control, and we don't trust it.

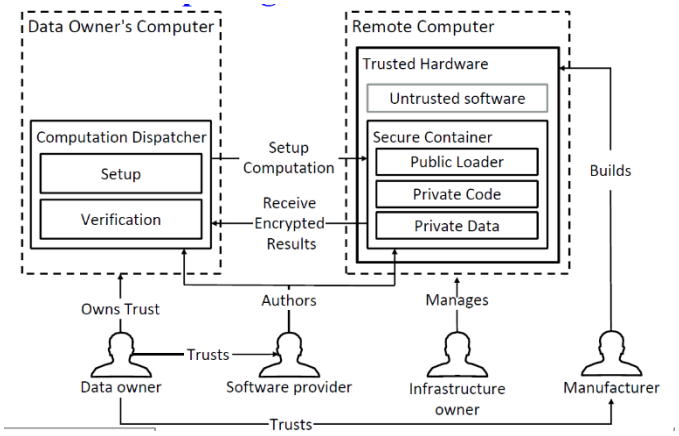


On the left of the picture there's the Data Owner's Computer, where there are some data and it is wanted to do some calculation/computation on this data on a remote computer, which is managed by some infrastructure owner (e.g., cloud computing, a cloud node such Amazon, or Alibaba, or Google), but we don't trust it.

So, what we want is some **guarantees of integrity** and **confidentiality** about this data and this computation. The infrastructure owner, that has complete control on this remote computer, should not be able to discover our code (so, for example, if we have some proprietary code and we want to execute it) and our data. It is a problem of confidentiality and integrity.

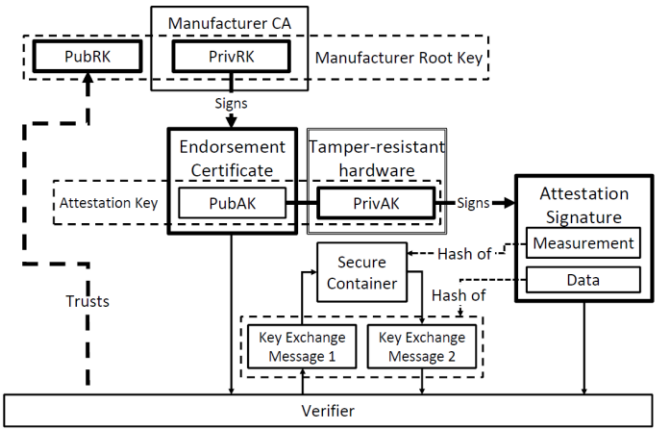
This is an unsolved problem for the moment: there's no complete assurances for the confidentiality and integrity point of view. The ideal solution would be that the data owner, which trusts his own computer and also trusts a software provider (the data owner trusts the software provider by using his software to handle his own data). The idea is that it is possible to setup a *computation dispatche*r, which is a software that runs on the owner's computer, that is used to establish a communication with the remote computer and to setup the computation on a container. The separate container, that runs inside the remote computer, in a way that it is possible to isolate the computation on the remote computer so that the infrastructure owner can't access, for example, the memory part of this container. Inside the container there will be both the *private code* and the *private* data of the data owner's computer, and the computation will be executed, which will produce at the end the *results*. The results are sent **encrypted** from the container to the computation dispatcher to avoid that these results can't be read by someone that is in the middle of this channel (no MITM attacks). In the end, there should be a *verification* from the computation dispatcher, to be sure that the container is still isolated from the rest of the remote computer, that the code has not been tampered, etc. So, this is the problem and the ideal solution.

Trusted computing



The aim of the trusted computing is to resolve the problem mentioned before, by introducing a secure container in a trusted hardware. In the picture there is the actual implementation of how to make this ideal container completely separated by the rest of the remote computer. The trusted hardware is made by a manufacturer (e.g., Intel). The data owner trusts the manufacturer, so it will automatically also trust the hardware. From the trusted hardware should be possible to establish the logical secure container and to give user enough assurances for the confidentiality and integrity of the code of the software provider and the data of the data owner. The rest of the picture is equal as before, it has just been introduced the actual implementation of the secure container. The secure container is obtained by putting some trusted hardware (e.g., an external chip, logical stack inside the actual CPU, etc.…). And this is built inside the remote computer, and it gives the assurances that the container is secure, and so it is possible to trust it.

Software attestation: chain of trust



To be sure that the container is secure (so that the trusted hardware is mounted and working inside the remote computer) the *software attestation* is used. The data owner's computer, before setting up the computation, it first attests the remote computer: it will assess its identity and the actual implementation of the trusted hardware that is running by using certificates. The client software asks to the server counterpart to attest itself and the remote computer answers with a proof (by a cryptographic signature). In particular, this cryptographic signature signs a hash of the contents of the remote content. In this way, it is sure that: we are actually talking to the remote computer we intended to; the computer is running this kind of trusted hardware; the contents of the remote container are actually what we expect.

The software provider authors the computation dispatcher, which in the end is going to be a module of the software of the gui of the client's side running on the data owner's computer, but it also authors the secure container, so the private code that is inside the container. All this verification is done by the client and the server side of the software, relying on the trusted hardware, but we are not going in any way to rely on the rest of the contents of the remote computer, because the rest of the remote computer is **untrusted**.

Again, the software is started on the data owner's computer, and the software of the data owner's computer starts to setting up the computation on the remote computer, but before doing it is going to attest the identity of the remote container (the server-side part of the actual software). In this way, by attesting the contents, we know that the actual code has not been tampered with and we know that the rest of the remote computer haven't been able to modify the contents of the container. The hash of the container content is certified and then this sign hashed contents of the secure container are sent back to the computation dispatcher on the data owner's computer that is going to verify it by using the verifier. To do this, the software attestation schema relies on a **chain of trust**.

On the top of the chain of trust there is the **manufacturer root key** (root key by Intel, AMD, …) which is an *asymmetric key* (public and private key). Then, there is an **attestation key**, which is on the actual CPU or on

the external chip, and also in this case it is an asymmetric key: the private part of the key is put inside a tamper-resistant part of the CPU, so something that does not permit to read the content by extracting the part from the CPU. Then there is the **endorsement certificate**, which is the public part of the attestation key, and it is signed by using the private key of the manufacturer. The **verifier trusts the public manufacturer root key**, it knows that this is going to be secure. The verifier sends a message to the secure container: "please, attest yourself". The secure container is going to make an **attestation signature**: to do so, the trusted hardware takes the content of the container, hashes it, takes the private attestation key inside the tamper-resistant hardware (it is the only that can do that), uses the private attestation key to sign the hash of the contents (code+data) of the container, and this is called the **measurement of the container**, and also it takes this measurement and signs it. Inside the data part, there will be also the signed key exchange messages. In this way we are sure that we are effectively talking to the secure container. We have: the attestation of the identity of the secure container and the integrity of the code and data inside this secure container. It sends back all this data and then by using the public root key the verifier will verify the attestation signature.

Trusted computing by Intel

Let's look to the implementation of the previous scheme made by Intel. The first one was the **TPM** (*Trusted Platform Module*): the TPM is an **auxiliary tamper-resistant chip**, so it can be mounted inside the CPU. It has been developed by the Trusted Computing Group, which is a big group of big farms in the IT scene. Intel is one of the participants of the trusted computing group (also HP, and others).

The idea is that TPM has to be **cheap**: it costs around 20$ and it is also cheap to install it, since it does not require any hardware modification to the CPU. The problem of that TPM is that *the attestation covers all the software running on the machine*! It is a problem because the attestation of the computer is made first on an initial state and everything works, but then if there are device drivers, e.g., by attaching a USB key the computer is going to stop because we are loading a device driver and we are attesting the memory, so this solution does not work. All computers made in the last 10 years have a TPM inside it. This is because the TPM is actually used, and it is used every time the PC is powered-on by the secure boot: UEFI.

UEFI uses the TPM to attest the content of the BIOS because the BIOS never changes (only if you update it, and when this happens the measurements inside the TPM are rewritten) so it is possible to be sure that no malware can modify the content of BIOS and this was a big problem before the secure boot.

Then there is the **TXT** (*Trusted eXecution Technology*) there is only made by Intel. It is based on the TPM and it reduces the amount of data that is actually attested. The attestation covers only a virtual machine inside the container. So, this is useful e.g., in a cloud computing scenario when it is wanted to run a secure virtual machine. There is a problem here: first of all, the active container has full control of the machine, so the problem is not solved, since even if we are sure that the active container is secure from all the rest of the contents of the remote machine, we are not sure of the vice versa. Because when there is an active container, it can read and modify all the contents of the remote machine. Also, the secure container is stored unencrypted in DRAM. So, when the virtual machine is offloaded, it is possible to reach the DRAM and look at all the content. So, there is no confidentiality assurances for the remote computation.

Intel SGX overview

The answers to previous problems by Intel are in the **SGX** (*Secure Guard eXtension*). An architectural extension is an **extension of the assembly language**, that is going to introduce new instructions.

It has been introduced starting from Intel Core 6-th generation processors (2015) (e.g. Intel® Core™ i7-**6**700 Processor, the '6' after the dash indicates the generation number).

The big functionality introduced with SGX is that the **attestation covers only the protected software**. SGX is able to obtain the closely-ideal secure contain (completely separated with all the confidentiality and integrity assurances).

This secure container is called "**enclave**". It contains only:

- *User private data*
- The *code operating on the user private data*

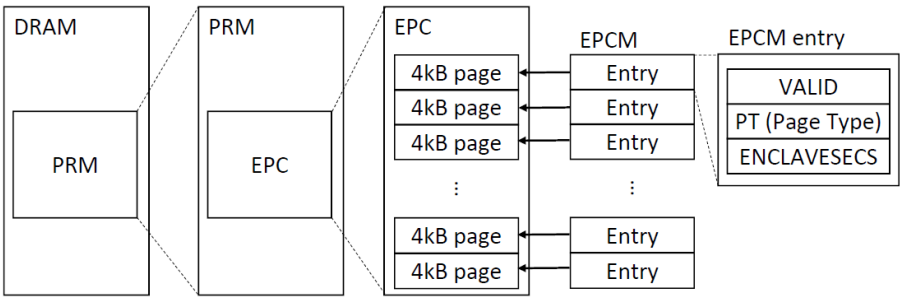
It is possible to have also **multiple enclaves** running in parallel on the same CPU, on the same server. This is perfect for the cloud computing scenario with secure remote computation.

One problem is that the **OS must support SGX**, so the OS must expose to the user space's applications the primitives, the system calls in this case, to create an enclave and to manage it.

Intel also distributes a free of charge SDK (*Set Developer Key*) that contains the C/C++ libraries to run SGX. There is an SDK for Windows that works well with VScode, and there is another one for Linux too.

Physical memory organization

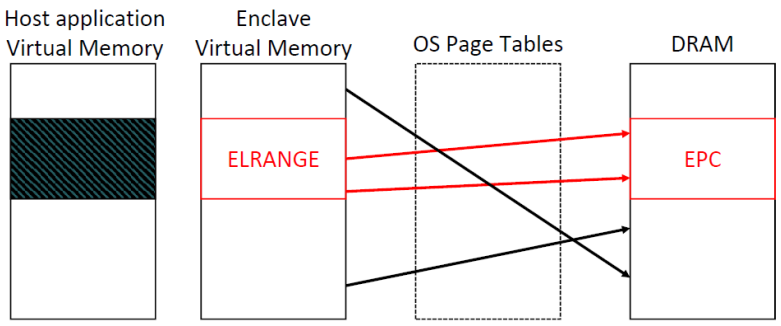
The first thing to see is how the physical memory is organized: the first big problem is that to have confidentiality assurances on the private data of the remote party we must be sure that no other processes/software/OS can go inside the memory and see the content of the enclave. To be sure of this, the only way is by using a hardware solution. The enclaves' code and data are stored in the hardware protected **Processor Reserved Memory** (**PRM**). In particular, the enclave's code and data are part of the *Enclave Page Cache* (*EPC*), which is inside the PRM. Intel has modified the hardware memory controller: since the only way to read/write on RAM is to go through the controller, then this modified controller can block the accesses from the outside. Encrypting everything is not a solution because if the actual enclave's code must be executed, and everything is encrypted in the RAM, what happens is that you need to decrypt the code, which is done by the OS, before starting the scheduling. By doing this, the code is in clear in the RAM and there is no assurance. So, this is why we need to have the enclave's code and data unencrypted in RAM but in a way where only the enclave's code can access the enclave's data in RAM. And this is why the memory controller actually deny everyone to access to this memory.



In the picture there is just a sketch of what there is inside the processor reserved memory. Inside the PRM there is the EPC. Inside the EPC there will be the OS paging. This is compatible with the existing paging scheme, since the idea is to have retrocompatibilty. There will be pages of 4 KB that will contain the enclave's code and data. Then, in another part of the PRM, there will be the *EPCM* (EPC Map) where there will be the actual indexes to all these pages: there will be an entry in the map for each page. Each entry of the EPCM (so for each page) there will be 3 sub-entries: one is a *VALID* bit saying if the page is valid or not, where valid means if the page contains enclave's code and data of an active enclave or if it is free; then the *page type* used to distinguish if page is assigned to a normal software enclave or to an enclave used by the GSX implementation; then the *ENCLAVESECS* which is a structure that identifies the enclave that owns the page, so it is a reference to another structure that identifies an enclave.

Enclave memory layout

The virtual addresses of the application virtual memory are the same inside and outside the enclave execution. The *ELRANGE (Enclave Linear Range)* is the range of the protected memory addresses mapping to the EPC pages. "Linear", for Intel, is a synonym of "*virtual*". The **address translation is done by the OS**.



In the picture there is the memory. From the point of view of the OS there is the virtual memory, the range of virtual addresses assigned to the host application. In this portion, it is placed the ELRANGE: the sub-range of addresses inside the virtual memory addresses assigned to the host application by the OS, in the normal way as always happens. In this linear range there are the *protected memory addresses*, so these virtual addresses actually map to the enclave's pages, to the actual physical enclave's pages in the processor reserved memory. This is basically the address translation, from virtual to physical addresses which is modified by SGX. The address translation is done by the OS, so the OS knows about the addresses of the PRM. The normal structure is maintained, so there will be

the mappings from the enclave linear range (the virtual addresses) through all the page tables, to the actual physical addresses of the DRAM, to the EPC pages. There will be also the translation for the addresses inside the enclave and for the addresses outside the enclave.

The address translation is done by the OS, so if an untrusted malicious OS tries to map an address inside the ELRANGE to another site, e.g., to a location in DRAM outside the EPC, it happens that the memory controller, when the OS tries to make this translation, blocks it. By leaving the translating operation to the OS it is possible to have complete compatibility from the previous OS, but this operation is controlled by the memory controller. So, this is how you introduce security while maintaining retro compatibility.

For every enclave there is one **SECS** (*SGX Enclave Control Structure*). The SECS, so the control structure, is stored in *dedicated EPC pages*, (this is why before there was the page type). This kind of pages are used by the SGX implementation. The SECS contain some attributes (Debug, XFRM, 32/64) for the specific enclave.

- **Debug** bit: it is a bit that says if the application is going into debug. Suppose to be the application developer that is introducing SGX in the application: the enclave is correctly ran, we want to perform debug on it, and I want to see the content of the memory in the debug → it is not possible (memory controller blocks the operation). By enabling Debug mode, there is no security, but it allows you to partially develop application.
- **XFRM** (*eXtended Feature Request Mask*): this is a way to tell to the SGX implementation all the others extensions used by the application.
- The **32/64** bit: tells if it is running a 32- or 64-bit app.

The security enclave control structure can be accessed only by the SGX implementation. It means that it can't be accessed by the OS, because addresses of the OS-based address translation could lead to memory mapping attacks (as mentioned before).

The EPC pages can be only be mapped to a specific virtual address. And this is how the memory controller blocks the accesses. The EPCM also contains the range of allowed address, and also the access permission bits, in order to have a correct use of the memory without security problems.

SGX supports **multi-threading**: a way to have concurrent threads of the same application in parallel execution on the CPU. It is an important point, since nowadays all CPUs are multi-core processors. It works essentially by **treating each thread separately**: there is a *thread control structure (TCS)* for each logical CPU executing the enclave code and this structure is stored in EPC pages. The TCS can be accesses only from SGX implementation (but can be partially read in debug mode). The TCS must be **allocated by the application before entering in the enclave**. For example, if 4 threads need to be executed concurrently, then to execute the enclave code we have to allocate 4 TCS.

When there is a **hardware interrupt** (when there is a call to a subroutine to handle that) it triggers a **context switch.** It is a problem because there is no idea of where to save the enclave code execution context (since the normal RAM is untrusted). To solve this, every TCS references a sequence of **State Save Area**, which are *secure areas of memory* (they are in the processor reserved memory, so same assurance). Each SSA is saved in **contiguous EPC pages**. The TCS, for each separate thread, needs to save the separate registers of the different logical CPUs, since logical CPUs have different registers.

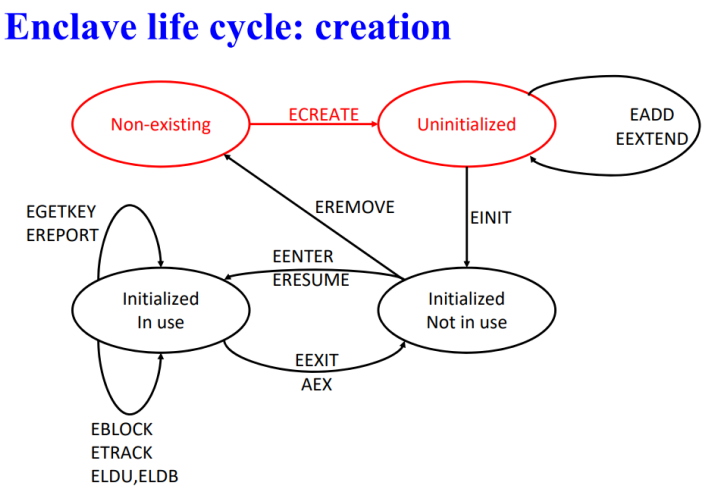
Enclave life cycle

In the cloud computing scenario (the user wants to execute the code on the remote PC), the secure container (enclave) is on the remote computer and it talks with the computation dispatcher. This also works in reverse.

To explain this, think about one of the biggest issues in software security is the *man at the end attack*. When a cracker has a game, he can execute it on his PC and can perform reverse engineering of the license key and release a crack for the game.

It works also in reverse: since the software provider does not trust the computer owner, it is possible to have that the software and the secure container both on the user PC. In this way the sensitive code (license check) works on the owner's PC and it is another scenario where SGX can be used, but nowadays it is not used because the software houses want to have retro compatibility with old CPUs (and with non-Intel CPUs).

Back to the enclave life cycle schema, the starting point is the **non-existing** enclave. The unsecure part of the software wants to create the enclave, so it calls the function from the SDK that calls the system call of the OS that in the end tells tells the CPU to execute the *ECREATE* instruction. With this instruction it is possible to create an empty enclave (an *uninitialized enclave*). Basically, the ECRATE instruction turns an existing EPC page into the enclave control structure (the ECCS) for the new enclave.



Inside the enclave page cache there are all the pages, inside one page there is the ECCS of the new enclave, and inside the ECCS there is the *Enclave ID*, the *Process ID* and other stuff, for example, the *EL Range*, that says "here in the RAM, inside the processor reserved memory this actual stuff is for this enclave".

Then, we must load the enclave code and data inside the enclave, so we use the *EADD* and *EEXTEND* instructions of SGX (all of these are instructions added by the extended instruction set of SGX). The *EADD* **takes the enclave and puts inside it the code and data** (this is part of the EPC, some of the EPC pages will contain the code and data). There will also be another page for the TCS, since it is needed in order to start up the computation and then save it.

The enclave code is saved on the hard disk normally, it is packaged with the software, so there is usually an encrypted DLL which is distributed with the rest of the software. The *EEXTEND* instruction is used to **create and update the enclave measurement**, that is the actual hash of the contents of the code that must be signed when we there is a remote attestation request.

After those instructions (the EEXTEND is called a lot of times, one time to update the measurement, one time for each EPC page that is used by the EADD) the enclave is ready to be executed. However, it must be executed one last instruction to get from the uninitialized state to the initialized, but not used, state. The instruction is *EINIT* (actually it is not needed but Intel forces to do it). If Intel does not want to launch your enclave, it can do that, since Intel has the key needed by the EINIT instruction. If EINIT instruction is not called with the correct key, the enclave will not go into the initialize state and so it won't be possible to execute it. So, the EINIT instruction is needed by Intel to have the control on which enclaves are launched.

After the EINIT instruction there are two possible cases: the enclave is initialized but not used, and the enclave is used.

In the first case there are the *EENTER* and the *EEXIT* instructions to go from the "not in use" to the "in use" state. With the *EENTER* instruction the code is running. If at a certain point there is an interrupt, the *AEX* instruction (*Asynchronous Enclave Exit*), that trigger a context switching, is executed. We will be now outside of the secure execution. When we want to resume, after the interrupt execution, we call the *ERESUME* instruction. At the end of the execution, there will be the *EEXIT* instruction and the enclave will go back again to the "not in use" state and the execution can go back to the insecure part of the software.

There are also other instructions that can be called during the "initialized" and "in use" state:

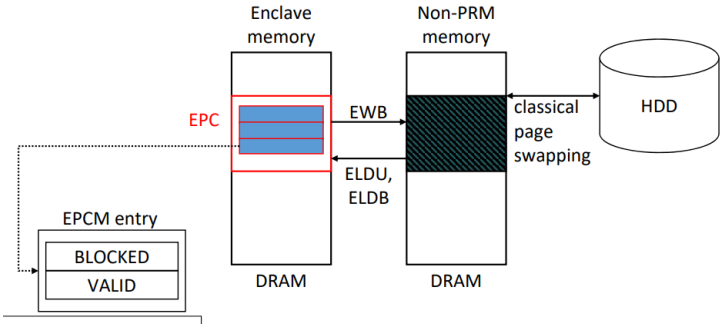
- *EGETKEY* and *EREPORT* instructions: these instructions are used during the software attestation. If there is an attestation request, they are called during the enclave execution (even if we are not in the enclave execution but if we have a request attestation the enclave must be started to answer to it).
- *EBLOCK*, *ETRACK*, *ELDU* and *ELDB* instructions: are used by the OS to handle the EPC pages (to handle pagination).

Finally, when I know that I don't need any more a specific enclave, it is possible to remove it from the memory by using the *EREMOVE* instruction. The OS calls the EREMOVE instruction, but it can also be called by the untrusted part of the application. This instruction **deallocates all the EPC pages associated to the enclave**.

Enclave page eviction

When a page is evicted, it is encrypted in hardware by Intel *Memory Encryption Engine (MEE)*. If the OS needs to evict a page in the EPC (a page of an enclave), this page must be saved encrypted, in the CPU there is a cryptographic chip that uses *128-bit AES-CTR + 128-bit Carter-Wegman MAC* to **swap the memory** (*to save the memory encrypted*).

In this schema the OS needs to swap a page. For retro compatibility it is not possible to simply use the MME to encrypt the page and then save it in the HDD, because it is not possible to modify the classical page swapping system used by the OS.



So first the contents of the page to be evicted are taken to the HDD by saving it into the non-PRM memory so that the OS can perform the classical page swapping.

The *EWB* (*Enclave Write Back*) instruction is used to save the page to the non-PRM memory (remember: the content of the page is not encrypted; the access is controlled by the MC). The EWB instruction also marks the *BLOCKED* value (if 1 no one can access the page) to 1 and *VALID* value of the EPCM entry as 0. By marking BLOCKED as 1 the CPU knows that the page is going to be evicted and no one can access it (in this way there will be no inconsistency). Then, after the page is evicted both BLOCKED and VALID values will be set to 0 (page is free and not blocked).

If the page needs to be loaded back from the HDD to the EPC, the VALID flag will be set to 1 and for the BLOCKED bit there will be two different instructions, since the ELD (*Enclave Load Data*) instruction is given in two types: ELDU (Unblocked, BLOCKED=0) and ELDB (Blocked, BLOCKED=1). This is not documented so the reason is not known.

Enclave measurement

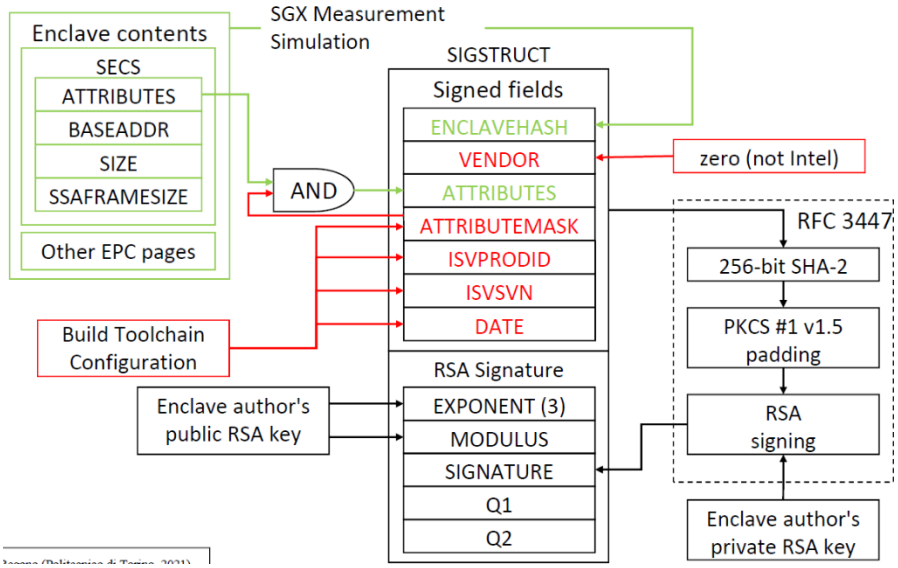
The enclave measurement is the base process for the remote attestation scheme: the measurement identifies the software running in the enclave. It is computed with *256-bit SHA-2 secure hash* function, this hash is stored in the enclave's *SECS* (*MRENCLAVE* field).

It is computed during the enclave initialization (done by the OS):

- *ECREATE*: executes the SHA-2 initialization algorithm, so goes into the SECS, in MRENCLAVE, and inserts the results of the initialization algorithm
- *EADD*: hashes the enclave virtual address + SECINFO, which is a structure that contains the security attributes relative to the page, in fact this is what goes inside the EPCM (metadata of the page in the enclave page cache map)
- *EEXTEND*: hashes 256 bytes of the enclave data (must be called multiple time)
- *EINIT*: SHA-2 finalization algorithm (that computes the valid final hash) inserted in MRENCLAVE (valid enclave measurement)

Enclave certificates

The SGX design requires each enclave to have a certificate, which is issued by the author and this requirement is enforced by the EINIT instruction. If a certificate is not provided, the enclave cannot be launched, since the EINIT checks the content of the certificate of the enclave.



The implementation of SGX requires certificates formatted with the **SIGSTRUCT** structure. The certificate is generated while building the shared libraries (the DLL libraries). This structure contains the metadata fields of the enclave and the RSA signature of these fields. The fields are:

- **ENCLAVEHASH**: the content is measured and hashed and put in this field. It is hashed also the data that is going to be loaded while executing the enclave in the SECS (attributes, base address for linear range, size of the enclave, etc…). One enclave page for the SECS and then all the other EPC pages needed to load the code. These are taken and signed. When the enclave is going to be loaded there will be the same structure inside the untrusted run on the untrusted computer. So, the measurement can be performed and compared. Everything goes in the field enclavehash.
- **VENDOR**: it is always 0 except for Intel (which will assume a value that is not 0).
- **ATTRIBUTES**: the attributes are those inside the SECS.
- **Build Toolchain Configuration**: *ATTRIBUTEMASK* (Masks some attributes of the enclave contents), *ISVPRODID* (product ID), *ISVSVN* (SGX Version Number), *DATE*.

All the previous fields are signed with the RFC-3447 scheme. It is computed the *256-bit SHA-2* hash which is padded with *PKCS #1 v1.5* and then it signed with RSA using the enclave author's private RSA key.

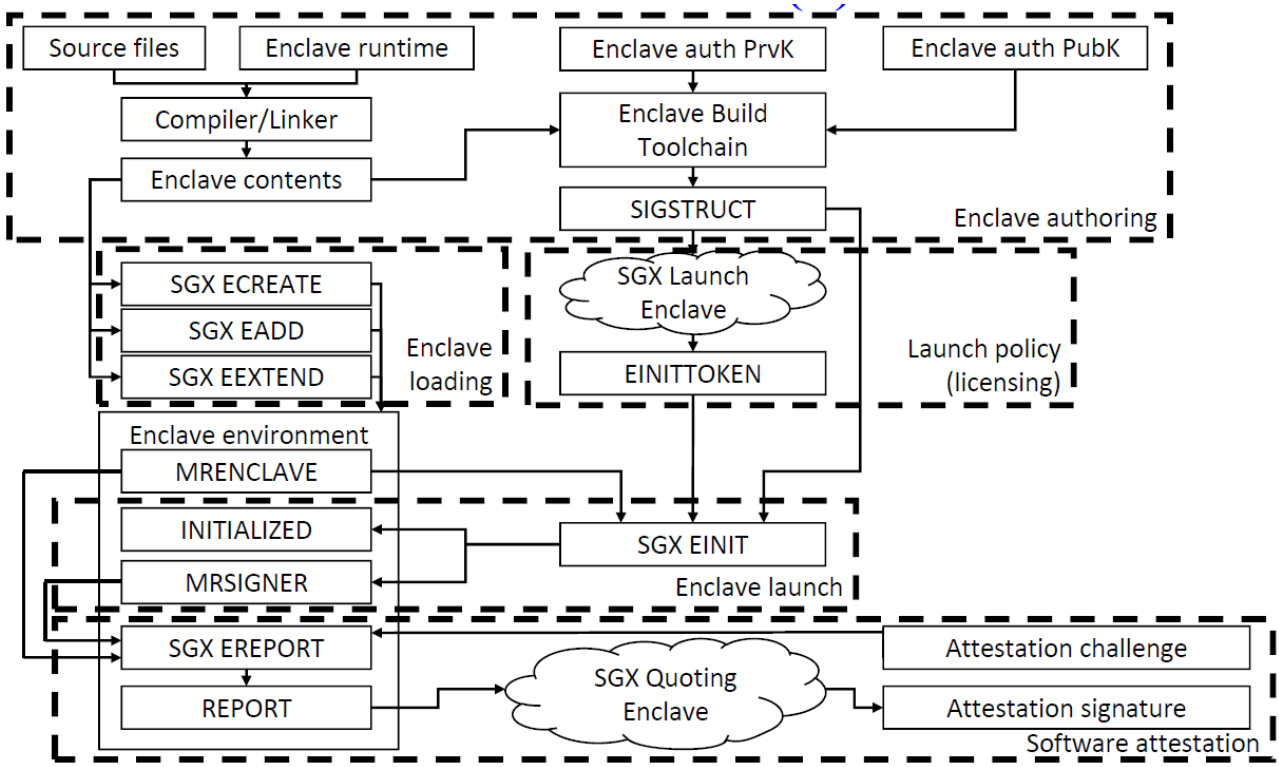
In the RSA signature there is the enclave author's public RSA key with the exponent and the modulus. There are also Q1 and Q2 due to CRT, which provides speeded-up verification of RSA signature. The previous signature goes in the signature field.

Enclave sealing

The enclave's private data can be saved on disk before the teardown of the enclave with "sealing", which uses *128-bit AES-GCM*. Data can then be retrieved when the enclave is reloaded in memory, so the enclave knows (it is not actually the enclave that knows the key since key is bound the certificate of the enclave) the key to reload the data. There are two sealing modes:

- *Enclave-specific*: only an enclave with the same MRENCLAVE can decrypt the data.
- *Author-specific*: only an enclave with the same author (i.e., certificate signed with the same public key) can decrypt the data.

Software remote attestation



The first dashed box is the **enclave authoring**. It is the part in where the DDL library is built. There are the *source files* and the *enclave runtime* together with compiler/linker made by Intel. The *enclave contents* are the real DLL. Then, apart from the DLL it is needed to build the certificate. The *Enclave Build Toolchain* takes the *author private key*, the *author public key* and the *content of the enclave* to build the *SIGSTRUCT*. The results of this phase are the encrypted DLL and the certificate.

To load the enclave, the **enclave loading** (dashed box on the middle-left) phase runs, which uses the *ECREATE*, *EADD* and *EEXTEND* instructions. Those instruction use the contents of the enclave.

Then there is the **launch policy (licensing)** which is a check of the license made by Intel. The *SGX launch enclave* is another enclave running on the remote computer that is made by Intel and it takes the certificate, goes to the remote server of Intel to verify the validity of the signature, and gives back an *EINITTOKEN* which is a one-time token that is used in the next phase to launch the enclave by the EINIT instruction.

The enclave loading phase generates the **enclave environment** which contains:

- MRENCLAVE: measurement of the enclave.
- INITIALIZED bit that goes to 1 (checked every time by all the instructions that are used to execute the enclave).
- MRSIGNER: measurement signer, the author of the enclave.

The INITIALIZED bit and the MRSIGNER field are set in the **enclave launch** phase when the EINIT instruction runs.

Then, the untrusted part wants the trusted part to attest itself, so here starts the **software attestation** phase. After the *attestation challenge* made by the user, the *EREPORT* instructions is called, and it takes the measurement of the enclave and the signer to create a report. This report is signed with a key that is unique to the key (a key inside the CPU). This report is not sent directly back to user, but it goes to the *SGX Quoting Enclave* which checks the signature and reauthors the report. Then the *attestation signature* is sent to the user. When the user checks the signature, it will use the public key of Intel.

To make everything work, there is a shared secret between the *CPU e-fuses* (one-time programmable part of the CPU that is tamper resistant) and the *Intel Provisioning Service* (which checks the signature). The provisioning Enclave, which runs inside the untrusted computer, obtains an attestation private key from the Intel Provisioning Service (IPS). This private key is stored encrypted in DRAM. Then, the remote party challenges the application enclave, the application enclave call EREPORT, the quoting Enclave verifies the report locally and then, if the verification is valid, it sends the report to the remote party signed with the attestation private key. In this way the remote party knows that the remote computer is running a correct SGX implementation, because the remote party asks to the IPS the attestation public key and verifies the report signature.

Intel EPID (Enhanced Privacy ID)

All this complication is made because there is the **Intel EPID (Enhanced Privacy ID)** which is a *custom cryptographic scheme*, and its implementation is undocumented. This schema **enables *Direct Anonymous Attestation***, which is an attestation that preserves the privacy of the user.

Every CPU is **part of a group** (CPUs of the same family or with the same SGX version). The enclave reports are **signed with the attestation private key** which are *unique key* for each CPU. In this way there is a 1:1 map from the requests of the same machine, so the IPS could know each machine which application is running. The enclave reports are then verified with the attestation public key, which is **one unique key for each CPU group**. So, the public key is the **same** for each CPU inside the group (no documentation about how it works).

In this way is **not possible** to bind a specific CPU to an attestation request, so it is not possible, i.e., knowing the enclaves/application a user is running.

Enclave launch control

The Intel Launch Enclave must authorize every enclave initialization. The SGX patents reveal a possible licensing scheme. The problem of this solution is that it is a way for Intel to decide which software vendor can use SGX. Since the EINIT is launched by the OS all these problems apply also to the authors of the OS (Microsoft, Apple).

Physical attacks

The SGX hardware implementation is largely unknown. It is not documented. What is known is that the MEE (Memory Encryption Engine) encrypts EPC data stored in DRAM, but it *does not encrypt the addresses used while loading/evicting lines from the cache*. It is a problem because it could be possible to tap (look) inside the *uncore's ring bus* and make an attack via GDXC (Generic Debug eXternal Connection, to look data passing through), so it could be possible to put pressure on the L3 cache by loading/unloading a lot of processes since we know that the CPU is going to evict the enclave data from the cache, which is going to be saved by the MC.

The *CPU e-fuses* are easy to read with high-resolution microscopes and the secret in e-fuses are shared by millions of chips. It is addressed in new SGX patents with PUF (*Physically Unclonable Functions*) which is a fancy name to say that the physical objects that contains the key is going to be unique due to physical properties (some gates can be open/closed) to have a unique identifier of the CPU. In this way, if the CPU e-fuses is read the possible attack is limited to that specific CPU.

Privileged software attacks

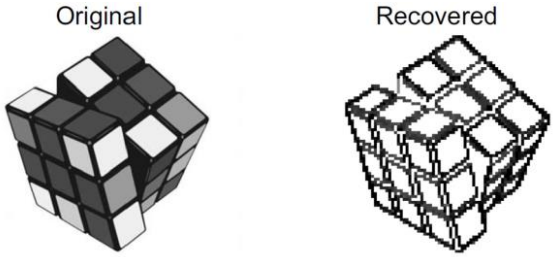
These are attacks that can be mounted by malicious OS. The SGX threat model consider the OS untrusted, and the SGX implementation runs in CPU microcode, which has a higher privileged level than the OS. Since the SGX allows the hyper-threading, a malicious OS can schedule on the same logical processor the *victim enclave thread* and a *snooping thread*. It is theoretically possible to discover the victim enclave instructions and memory access patterns. In practice, it means that if the enclave thread and the snooping thread are swapped a lot of time there will be a lot of contexts switching. By looking to the contexts switching due to hyper-threading there may be different times to do that, and this time may be bounded to the instruction that is executed. It is a **difficult attack**.

Memory mapping attacks

The active memory mapping attacks, where the OS tries to map an EPC page to a non-protected area of the memory, are defeated by the *EPC page eviction system*. But a malicious OS can infer **partial information** on enclave application memory access patterns. Given the instruction that is going to be executed, it is going to access memory in some way. If some patterns of access to memory are known, it may be possible to understand the instruction being executed.

A real attack performed in this way is a *JPEG* image partially recovered while decompressed in a SGX enclave. The problem is that the decompression algorithm of JPEG has specific patterns due to data on which it works.

The performers of this attack were able to infer, with a modification to the OS kernel, to infer some data on the JPEG. The original JPEG was a Rubik cube, and, on the right, there is what has been recovered.



The impact of this attack is **limited** because this problem is present only on a cloud computing scenario. Usually, even if the SGX implementation does not trust the remote owner, the end user should be sure that providers such as Amazon, Google are not going to do this kind of attack.

Cache timing attacks

The hit/miss latency of cache is available to ring-3 software for performance purposes (it may be needed to obtain a performance boost on the application by adapting the algorithm). The possible attack arises if the attacker and the victim process share the same cache (same CPU). The attacker process can **measure the access latency** to memory locations in his address space (for performance needs). If there are memory locations that map to same cache lines of victim process memory locations, the attacker process (when the 2 processes are swapped) can use collected data to learn victim process memory access pattern, since the victim process may use data-dependent memory fetches to process sensitive information.

SGX does not protect against cache timings attacks, since Intel says that these are difficult physical attacks (but they are not physical).

Another attack performed using a malicious OS that has complete control over cache placement is the **Prime+Probe attack**. The attacker *primes the cache*, which means that the entire cache is filled with data of an attacker process. Then, the victim executes code with sensitive data-dependent memory access (such as the JPEG decompressing algorithm) so that the attacker can *probe* (check) which of his cache lines got evicted. In this way there will be some knowledge of the memory access pattern of the victim process. Using this pattern it has been possible to recover an RSA-2048 key during decryption from RSA implementation in SGX SDK crypto library.

SGX in real-world applications

- *Ultra HD Blu-ray playback on PC:* It was first implemented by CyberLink PowerDVD 17 in order to decrypt the video stream inside an enclave (to make impossible the ripping of data). The system is now requested by the Blu-ray consortium.
- *NeuLion Digital Platform*: it handles the Ultra HD live streaming of American sports leagues.
- *Google Asylo*: it is a confidential computing open-source framework/SDK.
- *DashLane Password Manager*: encrypts in the enclave the master password, since it is a single point of failure of password managers.
- *PokitDok DokChain healthcare API*: author all accesses to medical data
- *Microsoft Azure Confidential Computing*
- *Alibaba Cloud Elastic Compute Service*
- *Numecent Cloudpaging*

How to use SGX into existing applications

Suppose that we want to secure the *myfunc* function in the following code (in a Linux environment):

```
#include <stdio.h>
#include <string.h>
#define MAX_BUFFER_LENGTH 100
void myfunc(char *buffer, size_t length) {
    const char *enc_str = «from the enclave!";
    if (length > strlen(enc_str))
        memcpy(buf, enc_str, strlen(enc_str)+1);
}
int main() {
    char buffer[MAX_BUFFER_LENGTH] = "Hello World";
    myfunc(buffer, MAX_BUFFER_LENGTH);
    printf("%s", buffer);
    return 0;
}
```

It is first needed to write the EDL (Enclave Description Language) file which identifies the functions in the enclave. Then, the *Edger8r tool* from Intel SGX SDK is executed, which generates the prototype declarations and function definitions for both the trusted and untrusted part, as follow:

```
// my_enclave.edl
enclave {
    trusted {
        public void myfunc([out, size= length] char* buf,
                                        size_t length);
    };
};
```

Then we manually fill the generated function definitions for both the trusted and untrusted part. By building this file will be generated an encrypted library (.so in Linux, .dll in Windows).

In the untrusted part it is imported the SGX library to manage enclaves, then it is imported the untrusted prototype declarations and it is set the name for the resulting enclave library.

```
#include <stdio.h>
#include <string.h>
#include "sgx_urts.h"
#include "my_enclave_u.h"
#define ENCLAVE_FILE_T(«my_enclave.signed.so")
#define MAX_BUFFER_LENGTH 100
```

Then, the enclave is created by loading the encrypted library

```
int main() {
    sgx_enclave_id_t enclave_id;
    sgx_status_t ret_code = SGX_SUCCESS;
    sgx_launch_token_t token = {0}; //Enclave launch control
    char buffer[MAX_BUFFER_LENGTH] = "Hello World!";
    // Create the Enclave with above launch token.
    ret_code = sgx_create_enclave(ENCLAVE_FILE, SGX_DEBUG_FLAG,
                                  &token, 0, &enclave_id, NULL);
    if (ret_code != SGX_SUCCESS) {
        printf("Error %#x failed to create enclave\n", ret_code);
        return -1;
    }
```

Then the functions are called inside the enclave

```
    myfunc(enclave_id, buffer, MAX_BUFFER_LENGTH);
    printf("%s", buffer);
    // Destroy the enclave before exiting program
    if(sgx_destroy_enclave(eid) != SGX_SUCCESS)
        return -1;
    return 0;
}
```

Finally, it is possible to build the application. There are two separate builds for the trusted and untrusted part. The .so library must be signed with the developer key provided by Intel using the sign tool provided in the Intel SGX SDK.

Research on Intel SGX

The group of PoliTo is working on easing the deployment of SGX on existing application, so instead of writing all the manual operation needed, the idea is to have a *comprehensive framework* to automatize the SGX adoption in existing software. There is an initial proof of concept, which was a master thesis work of a student. Another student is now working on *automatic implementation for RA scheme* (a verifier that calls the attestation anytime a specific enclave is run). There are still many limitations to overcome.

Conclusions

- SGX successfully (in theory) reduces attestation to the critical section of the application
- The libraries provide an easy (in theory) implementation of SGX in existing applications.
- The OS is untrusted, feasible application in IaaS systems
- Enclave launch control system is troubling
- Many feasible attacks (especially cache timings attacks)

Computer forensics

Computer forensics (CF) derive from *forensic science* (*forensic analysis*) which is the application of scientific knowledge and methodology to legal problems and criminal investigations. It is possible to distinguish:

- **Original forensics** which are related to autopsy, detection of fingerprints or footprints in a crime scene.
- **Digital forensics** applies the same concepts of the original one in the fields of
    - computer devices (and hardware components)
    - network forensics
    - forensic data analysis
    - mobile device forensics

Moreover, the investigation must be performed in a legally admissible way, respecting all the policies and laws e.g., the sniffing of a communication in a not authorized way cannot be provided to the court in a trial.

Some possible definitions of Computer forensics are:

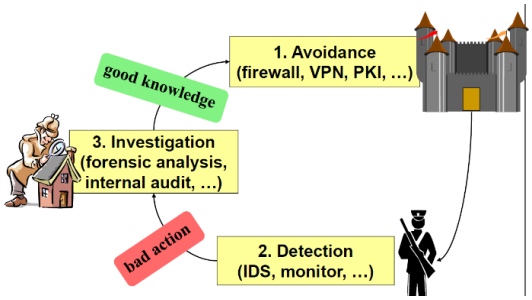
- *US_CERT*: the discipline that combines elements of law and computer science to collect and analyse data from computer systems, networks, wireless communications, and storage devices in a way that is admissible as evidence in a court of law.
- *A. Ghirardini - Computer Forensics*: the discipline whose goal is preservation, identification, analysis of information system to the aim of identification of evidences during investigation activities.
- *NIST glossary*: the application of computer science and investigative procedures involving the examination of digital evidence - following proper search authority, chain of custody, validation with mathematics, use of validated tools, repeatability, reporting, and possibly expert testimony.

We are interested in the discipline of better understand what is going on in a computer science field, to understand the crime. There is a set of questions that allow to identify who and in which way perform any kind of detection. The key questions are:

- *What happened?* –sometimes it cannot be immediately obvious, e.g., the date of a file can be changed through the filesystem. If there is not something based on cryptographic techniques it cannot prove the integrity of data.
- *Who was involved?* – it is important to discover who was involved in that specific set of actions (malicious or not). If malicious to understand who is guilty, if not malicious to understand who is not guilty.
- *When did it take place?*
- *Where did it take place?* – it can be the device, but if the attack happened also on the cloud, it can be the region, state, continent. If the system interacted with systems in different countries in each of them there are different laws, so the proceeding way is different and more difficult.
- *Why did take place?* – in this way it is possible to understand, ideally, the whole set of micro-actions that allowed the attack and how to prevent a similar attack in the future.
    - *How did an incident occur?* – e.g., through which device it has been performed. If in cloud it is trickier to understand.

The answers of those questions lead to: confirm or refute allegations of an incident, support the mitigation of damage, mature future prevention approaches.

Security is based on a continuous cycle of avoidance. When it is not enough and something bad happens, it must be detected. After the detection of a bad action (e.g., using detection systems), it is investigated through *computer forensics*, so the crime scene is studied and understood to discover what was the issue. After the issue is resolved, the new state will be a better knowledge in order to avoid the same problem next time and then the cycle restarts.



Terminology

- **Legal**: requires adhering to law, rules, and procedures, which are different on a per-country basis.
- **Evidence:** requires finding concrete (computer science) "*facts*". Something that is the proof of something else. An evidence can be a *file log*, so any possible fact related to computer science. In order to be classified as evidence it has to be acquired in a very careful way. It has also to be maintained in a proper way to avoid any unwanted modification.
- **Preservation:** protect the integrity of the proofs for current and future analysis.
- **Reconstruction and explanation:** use detailed knowledge of mechanisms to understand and explain events and actions. Events has to be explained to not expert people (judge, experts, relatives). So, the analysis should be always ready to be performed multiple times with the same results.

Typical computer forensics scenario

- *Internet abuse from employee*: a possible scenario is when a former employee abuses the internet usage inside an organization. E.g., when a company fires an employee but it does not accept the decision and so it tries to damage the internal systems.
- *Computer-aided frauds*: when receiving a malicious message with a malware stealing credentials and that performs a bank transfer using the name of the victim. Also, this regards data theft or data disclosure.
- *Computer/network damage assessment*: e.g., in case of a computer infected with a ransomware. It is better to understand if it infected other devices or the network and discover why the anti-malware didn't detect it. All this is done to improve defences.

In general, computer forensics can be applied any time digital evidences may be involved in an incident.

Characteristics of evidence

Data **can be organized at different levels of abstraction**. Data can be a signal (electric/optic) but then it acquires a logical form in terms of *bits*, which are quite meaningless, but aggregating them in bytes and files is easier to associate meanings to data. The key aspect is that **data requires interpretation**, so there must be the creativity to interpret the data to give them a meaning, in the sense that e.g., it can be an attack because it is possible to see a trace path of the network perimeter, then a strange action on the gateway, etc...

Data is also **fragile**, meaning that while performing the investigation it is possible to alter data and so it is possible to destroy the way to the guilty. Especially today, data can be **voluminous**, data exchanged are in the order of GBs and for a computer forensics analyst in any of them can be hide traces of meaningful data. Moreover, data is **difficult to be associated with reality**.

Investigation process

Forensic analysis can be divided in a set of main phases:

- **Acquisition**: physically or remotely taking possession of the computer and its network / physical connections (e.g., network or USB disk). It is acquired the target of investigation.
- **Identification**: identifying what data could be recovered and electronically retrieving it by running various CF tools and software suites. During investigation has to be identified if there is any hidden data and what kind of data can be recovered. Sometimes it is needed to analyse the OS of different devices, or the firmware of sensors (if it is the case) and it is a very complex. About the tools there are

many that interact at low level with the OS, e.g., *FDISK* that allow to interact with partition tables in a logical way, or hexadecimal editor or reader etc.

- **Evaluation**: evaluating the recovered data to determine if and how they could be used against the suspect (e.g., for prosecution in court).
- **Presentation:** presenting the evidence discovered in a manner which is understood by lawyers, non-technical staff/management, and suitable as evidence (according to the law or internal rules). The investigation has to be finalized providing a report that explain what is the result and conclusion.

Tool requirements

These kinds of forensic tools must have these characteristics:

- **Usability**: present data at a layer of abstraction useful to an investigator. Must have an adequate level of abstraction to be understood also to not very skilful people in this field. In most of cases this is not true since many tools provide raw data hard to be read.
- **Comprehensiveness**: present all data to investigator so that both inculpatory and exculpatory evidence can be identified. All the different pieces of the analysis (data, network, hardware, devices) must be put together and properly stored.
- **Accuracy**: tool output must be verifiable with a (small) margin of error. If errors are introduced during the analysis, it cannot be tolerated. For example, a tool that verifies the integrity of evidence creates message digests for the evidence, and as we know the message digest is not theoretically error-free, meaning that collision can happen (even if with small probability). This kind of small margin of error is tolerable in a computer forensics analysis. In a computer forensics analysis, it is often possible to see that many digests with different algorithms are performed (md5, sha-1) to even reduce the small error margin and that an attack can be performed on a not robust algorithm (md5 is not robust but still used because it does not use lot of memory).
- **Deterministic**: produce the same output given the same rules and input data. Having the same environment with the same values of the variables involved, the output that a computer forensics tool produces must be the same. E.g., performing a digest, regardless how many times it is performed, the output must be always the same.
- **Verifiable**: ability to verify the output by accessing the layer inputs and outputs (verification can be done by hand or via a second tool). There must not be implementation errors, e.g. if calculate a message digest with an md5 tool, the same digest must be calculated with another tool that perform md5, because the implementation shall not introduce any error. In the general case other party in a trial can use different tools in order to check my conclusions that must be robust, I should not have used any broken tool that introduced error in my analysis.

Challenges

The main difficulties are:

- The **size of the storage devices** used to collect data
- The **embedded flash devices**. It is the fact that this data can be stored in different shapes, different physical containers like normal hard drive, magnetic tape, flash memory or embedded in devices leading to an impossibility to extract a hard disk from a laptop or a device due to construction requirements.
- Files can be in different shapes because of **different operating systems and file formats**. It's not unlikely that a normal computer analyst doesn't know any possible file format or operating system, e.g., related to sensors.
- We have to put together data from different devices and sources at the same time (inside the same criminal investigation). So, there is a **multi-device analysis**.
- **Pervasive encryption**: today most of the communication on the internet is encrypted, almost every site uses HTTPS. This can be very challenging from the inspection point of view because it's difficult to find what's going on in the network that you are inspecting like your home or company. That's why

we are looking for ways to **detect malwares without looking at the payload of packets** but looking at their behaviour and pattern.

- **Cloud computing**: data can be now spread in many different premises (not under the control of an organization) or places which may introduce legal problems because of different states jurisdictions or technical problems because there can be many replicas of data which can be different (some can prove innocence, some not). There must also be the managing to acquire the evidence accordingly to how data are managed in a single cloud provider (interact and improve knowledge about these mechanisms).
- **RAM-only malwares** can leave a very small amount on traces. This malware do not interact at all with the storage media, meaning that if the analyst acquires a laptop in order to understand if there's a malware in there, it is switched off and took to the laboratory, it is not possible to find nothing when examination starts. Of course, it is possible to avoid the switching off the machine and try to collect data as soon as possible, e.g., when the laptop is collected.
- Legal challenges decreasing the scope of forensic investigations. Everything of course must be legal.

After these practical challenges, it is possible to examine methodology challenges:

- **The forensics uncertainty**: anything done to a system disturbs it. It is like the principle of uncertainty, like the Heisenberg principle, which lead to collect **copies** to avoid to "disturb" a system and leave the original artifact as it is. This can be more subtle than how it appears. A computer's analyst tries to save the computer as it is in the moment in which it is acquired (so also caches, RAM, and everything else that may be volatile). There are tools that try to take a snapshot of a system to replicate its state in the moment in which the replica has been created (not after the switch off, because it changes a lot the state of the system). Other methods can be used to avoid loss of volatile data, for example it is possible to low the RAM temperature to slow the data loss of and giving time to arrive to the laboratory without losing data (with the same state).
- **Trust no one**: it is not possible to use a not trusted system because this can activate some countermeasure that a malicious actor has made available on its device. For example, on a login screen (username and password) if it is inserted a specific couple of username and password this can trigger the activation of some tools that delete some files on the computer, and this is very easy to do on a computer. A malicious guy can, for example, give the wrong credentials to make you trigger this action.
- **The history is written by the winners**: going through an analysis it is not possible to trust any user information. Looking into a specific device, so by looking through data associated to files and applications, any file can be fake and forged just to avoid the identification of a crime, so it is not possible to rely only on the regular system information. It is not possible to take as trusted the information inside a device even if they are obtained through the analyst tools. It is possible to get all the possible information and check for any inconsistency between them. For example, if there's a mismatch between the dimension of a file as stored in the file size table and the dimension of the file itself there could be something that could have been erased not through the proper tool of the operating system.
- **Everything is impermanent**: it must be remembered the so-called **order of volatility**. Some stuff can last less than other. This could be the case where a criminal switch off the machine, so what is going to disappear first? It is important to collect the most-volatile data first.

Order of Volatility

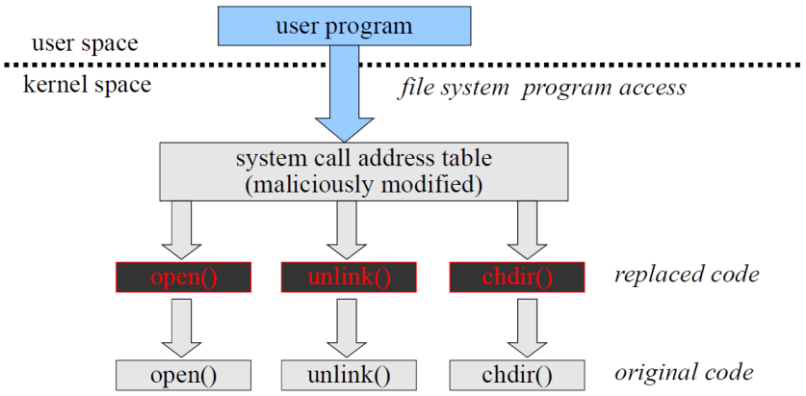
The collection of data must give priority to the most-volatile first. In the following list there are in decreasing order the elements that are first erased:

- Registers, cache
- Routing table, ARP cache, process table, kernel statistics, memory
- Temporary file systems
- Disk
- Remote logging and monitoring data
- Physical configuration, network topology
- Archival media

Trusted Environment

The analysis must be setup in a trusted environment because there are plenty of possible stuff that can go wrong in an unknown/enemy environment. For example, very easily tools like *rootkits* can change the usual behaviour of an OS and can change the behaviour of command line tools (ls, cp, mv, …) or hide information on some specific files. Even worse, they can change not only command line tools but also the system calls that the application uses to interact at a lower level to the OS. For example, there can be the intercept of open(), chdir(), unlink(), … to not show or act on specific files.

System Call Interception



When a user program runs, even tools that may be started from a USB Drive of the analyst can act in a non-proper way if they rely on dynamic libraries. This is because the OS conceptually like shown in the picture can have replaced all the authorized system calls with a fake version that is a sort of wrapper that intercept the call, maybe perform a filter, or check in advance, for example, to deny the examination of a specific file. Then, after that, call the original one for the rest of data. In this way the system can apparently behave correctly most of the time, but in fact it is not.

Portable OS

It may be not always possible to have a trusted environment, Portable OS comes for help to create one. Kali is an example of this, it can run from a "live CD" **fully in RAM memory** and also has some nice options that avoid the compromission of the evidence, for example the "*automount*" (option that automatically mount the devices present in a device) option is disabled. This means that to mount any partition present on the disk of that device it needs to be mounted in a **read only** way to avoid compromising the data in the hard disk.

There can be various kind of OS like security general-purpose (GRML, Kali, …) or forensic special purpose (DEFT, CAINE, Helix, SIFT, …).

There are several specifically developed distribution for forensic analysis. Some of them come from Italian teams. When the analysts are on the environment to be analysed, if there's no time or it is not wanted to take it, remove from the environment, and take it to the environment, it is possible to run a different operating system in order to explore the target of forensics analysis.

Another possible way is to **do not switch off the device** (and switch on with a different OS) but instead the analyst can try to connect the device with another one, make a full copy of it as much as it is possible. If

possible, there will be a copy also of the RAM and then the copy is put in a **virtual environment** (trusted environment created for the analysis of the media/memory/storage/device).

An example is shown at this point, in which Professor opens VirtualBox on which there is its trusted environment (Kali OS). This OS is started in "*Live (forensic mode)*". From the command line, it is used the $sudo\ fdisk - l$ command, which gives the whole list of devices (professor has only available one virtual device). To add a virtual disk to Kali, Professor switches off the OS and opens the settings of the virtual machine. In the *Storage* option Professor adds a new hard disk (an empty one in this case). When Professor runs again the VM it will pop-up giving the same command as before. Now there will be an environment that can be exploited for forensics analysis. What is missing now is the image to be analysed.

It is now supposed to have acquired an image to be investigated. The professor, after he has fetched from a remote host the image to be analysed, he performs a **bit per bit copy** of the image that he has got from the suspect device to the new virtual hard disk using the command $dd\ if = [suspectImage]\ of = [virtualHardDiskPath]$. In this specific case professor used $dd\ if = image.dd\ of = /dev/sda$. As a final step Professor mounts the device to a new directory he has just created, using command $mount\ /dev/sda\ newimagedir/$. Using the directory $newimagedir/$ it will be possible to analyse all the files and start the investigation. Anyway, this is the content of the fifth laboratory.

Linux distro interaction with devices

When interacting with devices there must be attention to **avoid any modification**, especially if it is not possible to perform a copy and the work runs on the original one. A good choice is to **mount block devices as read-only** which can be done with the command $mount - o\ ro\ /dev/sdb1\ /mnt/evidence1$. This could be not enough, in the sense that the mounting point is like a metaphor to access a virtual device. It should be also set the device itself as read-only. To do this, the following command is used: $blockdev - setro\ /dev/sdb1$ which is available almost in every Linux distro.

Professor showed then an example of *blockdev* command. By using $blockdev - -report$ it is listed the status of every mounted device in the system. It can be noticed that the virtual hard disk $/dev/sda$ is in a read-write mode, and this is not related to the mount option. For this reason, the following command is used: $blockdev - -setro\ /dev/sda$ to make it read-only. By using again the report command it will pop up in read only mode.

It is possible to also use GUI tools such as *UnBlock* tool.

Disk image mounting

As showed in the example before, it is needed to perform a bit per bit copy of the image in a new hard disk because the usual copy changes the metadata of the files that can be very useful in an investigation. Of course, it is possible to perform a copy of a disk into an image using the same command as before but inverting the arguments like: *dd if=[virtualHardDiskPath] of=[newImage]*

File System

A file is the smallest logical unit from a user perspective that can be stored: it can be stored in bytes, lines, records, etc… Logical files are mapped into "physical" entities (computer RAM, HD, the Cloud, …) by the Operating System: memory addresses, disk sectors, remote resources, etc… File systems define the organization and the structure of files on a computing device. Through the OS, a file system defines the rules to read, write and maintain the data.

The fact that OS provide this kind of metaphor is fine for almost any end user, but not fully fine from a computer forensics point of view, because inside the organization of the physical structure of a device into a logical one there are some substitutions in which information can disappear.

The file systems also provide **file attributes** such as:

- **Name**
  - A mnemonic (human) id for reference
  - DOS legacy: 8 chars + 3 chars for the extension (no modern OS still have this limitation, but many names still have)
- **Type**
  - Categorize the file to indicate how should be manipulated
  - "Magic number" (few bytes) usually at the file start, but…
  - Windows rely on extension to associate programs
- **Protection**
  - Access control information
  - Differ depending on OS/FS combination
    - E.g., owner and group (unix-like)
    - Read, write, execute (unix-like)
- **Location**
- **Size**
- **And some other**

FAT example

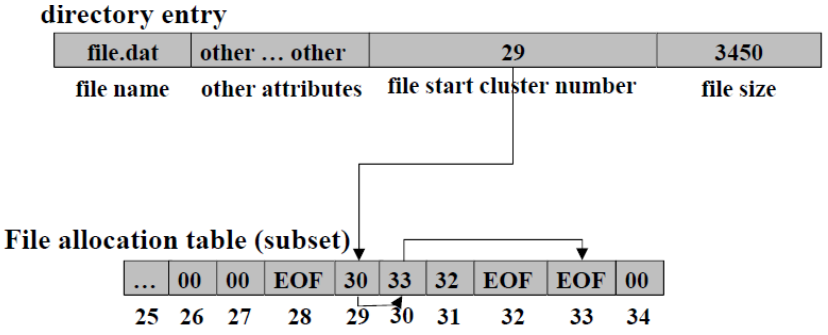
File organization is an abstraction created while performing the **formatting** operation, which is a preliminary operation needed to be performed in order to allow OS to understand how to organize/read/interact with the physical memory. While formatting the HD, the following items are created:

- The *Boot Record* which details the name and version of the OS plus the disk physical characteristics
- The *Master File Table* (2 times, for reliability reasons) which identifies the starting point of files inside the storage media and contains info on clusters (available, allocated, damaged, containing OS files)
- The *Directory Table* which contains the top-level folder file and directory information.

The **FAT** (*File Allocation Table*) is where the OS records, e.g., the files' position. It is close to the begin of the volume and it is specified in the boot sector. It is again made of 2 copies for redundancy, but it is optional.

Boot sector	Reserved	FAT 1	FAT 2	Root Folder	Other folders and files

To analyse a storage media is possible to use a hexadecimal editor/reader on the storage. Professor showed here an example from the VM by looking inside the image with the command $hexedit\ /dev/sda$.



In the FAT there is the information that allow to reconstruct a file. It is needed to understand how a file system is organized because this allows to understand what something needs more exploration.

The structure of a directory entry contains the *file name*, *some attributes*, file size, and the *file start cluster number*. The interesting point is that the file can be reconstructed because inside the FAT there is the sequence of the position of the pieces of the file. File in the picture is composed by 29, 30, 33.

It is interesting from a forensics point of view because when a file is deleted by the OS it is cleaver to improve performance of any kind of operation. The most efficient thing to do while deleting a file is to just say that file has been deleted exactly by deleting the first point (or the chain in the FAT). It is a quick operation because a very small amount of data has been changed (instead of deleting e.g., 1GB of a movie). This means that file is

still on the storage media (sectors are now marked as free and maybe in the future someone will write on them and delete part or entirely the file). Until that time, files will remain on the storage media.

If someone developed a device (a CF tool) that do not rely on information that the usual system call get back, but instead it searches inside the storage media (e.g., using hexedit tool) it is possible to find all the not-deleted history of that machine. Most of the time this is a enormous resource of information.

File copy

Normal commands preserve the file content but alter the file attributes (meta-data) for example in the creation data. It is required a bit-per-bit copy to avoid any modification:

- "Data dump" (dd) copy/convert bit-per-bit
  - $dd\ if =< inputfile > of =< outputfile >$
- Variants (e.g., *dcfldd*, *dc3dd*) with CF added features
  - E.g., on the fly hasing (md5, sha-1, sha-256, and sha-512), pattern wiping (when OS delete a file it just deletes the structure saying sectors are available, but programs such as data dump file allow to securely erase a file, writing on the specific positions of the file some patterns such as a sequence of 0 or some bytes), progress report

Digest are performed to be sure that no errors occurred in the operation. Some examples:

- Clone one hard drive onto another
  - $dd\ if =/dev/sda\ of =/dev/sdb$
- Clone a hard drive to an image file
  - $dd\ if =/dev/sda\ of =/image.img$
- Clone a hard drive to a zip image in 100Mb blocks
  - $dd\ if =/dev/hda\ bs = 100M \mid gzip - c\ > /image.img$
- Wipe a hard drive with binary 0s
  - $dcfldd\ pattern = 00\ vf =/dev/hdb$
- Write a binary image and calculate hash
  - $dc3dd\ if =/var/log/messages\ of =/tmp/dc3dd\ hash = sha512$

File Analysis - Metadata

Many other sources of information exist in respect of what is immediately apparent. The concept of **metadata** (like file attributes) can be applied in almost all kind of different files. This is because it is a useful concept, since it is useful to know what the main characteristics of a file are, because in that way it is possible to operate efficiently while manipulating that file. Metadata are information about a file. For example, they can be information about the structure of a word document, many characteristics of a taken picture, and so on… If from one side they are useful to operate, from the other they are a larger set of information and they are the basic source that a computer forensics analyst looks for. Since they are not trusted, they can be as well manipulated and can carry fake information.

Metadata example: ODF

It is the *Open Document Format*. It is a metadata XML-based style used by OpenOffice suite to categorize a document (e.g., assign a title, give description of the subject of the text inside a document, who is the creator, creation date). It is a format close to the one used in the *docx* format.

Metadata example: JPEG

The most popular is the *Exchangeable Image File (EXIF)* format allow to characterize a picture, like detailing the resolution, which camera performed the operation, the flash, etc…

Starting from EXIF there is a powerful tool *exiftool* developed in PERL that allow to query and see metadata (first only EXIF but now supports many kinds of files). EXIF tool is a common tool for a computer forensics analyst.

Metadata are useful also to identify a file. Extension is not a reliable source: literally anyone can change them. It is useful to check the metadata where available. Checking the first bytes of the file can act as a signature (there is a list of file signatures on Wikipedia), so just compare the signature with hex dump of the file.

Some tools for this purpose are:

- **hexdump**: command line utility to display a file in a specified format (default hex)
  - $hexdump < filename >$
- **hexedit:** hexadecimal command line file viewer and editor
  - $hexedit < filename >$
- **ghex**: graphic (GNOME) hexadecimal editor. It allows user to load data from any file, view and edit it either in hex or ASCII format
  - $ghex < filename >$

Metadata example: file system

The file system maintains several information about file contents. Depending on file system it is possible to have the **journal**, which is information about operation going on the operating system that allows to find inconsistences. Only modern file systems have this feature (such as ext3, ext4).

Slack space

The slack space is the leftover space when a file does not fill exactly a sector multiple size. It is basically the difference between the logical (bytes) and physical (sectors) file size. Since sectors have fixed dimension (e.g., 512 bytes) but files do not have such fixed dimension (e.g., 392 bytes), the file will result in 120 bytes of *slack space*. If the first file is erased, and a second smaller file (e.g., 192 bytes) is allocated in that space, 200 bytes of the old file will remain available in the slack. This means that even after new data has been placed, some of the old information can still be there and live there for a huge time.

The tool **foremost** is a command line tool that "curves" data from disk images, e.g., inspect content of an image looking for erased files. Identify file types on the base of file signature and metadata

- $foremost - t\ jpg, gif - I\ image.dd - o\ outdir$
  - Look for *.jpg* and *.gif* data
  - In the *image.dd* disk image
  - And put the recovered files in *outdir*

Data sanitization tools

To avoid recovering data there are some data sanitization tools. Its purpose is to make unrecoverable a piece of data by overriding with a specific pattern the storage to dismiss.

- File Shredder Programs
  - Permanently delete selected files
  - Overwrite using a specified data sanitization method
  - Ensures they cannot be undeleted
- Data Destruction Software
  - Completely erase (delete) all data on a HDD
  - One or more data sanitization methods to permanently overwrite all the information
  - Suitable for virus removal, HDD disposal or recycling

There are also standards that specify how to sanitize a storage media. For example, in the USA Airforce there is the **AFSSI-5020** that prescribes three passes of overwrites: first pass with '0', second pass with '1', third pass with random character and verifies the write. This procedure exists because in theory there may be an hardware tool capable of recovering the previous memorized value (just in theory, never seen in practise).

Steganography

It is another technique to hide information, but differently from cryptography it led the observer to think that the information does not exist. It is suitable only for some specific formats (e.g., JPG). In a JPG format it is possible to change some not important bits of the image and still, from the perceptual point of view, the image is still the same (maybe just some changes on the colour) and through some steganographic tool it is possible to break a secret message in single bits and put the value of the bit as the least significant part of the bit of the image. At the end of the communication the steganographic tool can recover the text, see all the least significant bits of the image and recover the secret message. The secret message **must** be lower than the vector data.

File format for evidences

Many time it is needed to manage evidences, for example a whole hard disk, in a very careful way, meaning that it is needed to acquire, maintain, make secure and assure the integrity of that and furthermore it could also be possible to aggregate (merge different sources, hard disks). In real cases the investigation is usually performed by different people, each devoted to a specific aspect, so it is needed to exchange our evidences and allow many people to securely investigate and acquire an image, for example.

Furthermore, even one of this evidences may be large (also TB), so the point is that it is needed to efficiently aggregate this set of sources.

There are many different file formats in order to securely manage evidences:

- **Raw**: bit-per-bit copy of evidences. The result is a collection of different files, one for any different storage and it is up to us to coherently manage and combine them during the analysis, maintaining different files (dd images for example, using dd command line tool).
- **Expert Witness Format (EWF)**: derives from the *EnCase suite* case which is a commercial set of tools specifically suited for computer forensic. A suite case is a set of tools just put together in order to be available for automatic analysis or available on command line just in one place, plus the appropriate file format to have together all the different evidences and the possibility to perform all the analysis (e.g., checking all file types that need to be analysed), and put the result in one place.
- **Advanced Forensic Format (AFF):** the purpose is the same of EWF but free.

Expert Witness Format

With EWF it is available something that allows to store integrity information of the images acquired, the content and some meta information about the evidences (when it was acquired, managed etc...), and the possibility to index the information in order to access them more efficiently, because there may be TBs of files and in forensic analysis it must be inspected every single bit, so this is a time-costly operation.

Advanced Forensic Format

It is an open-source standard for forensic-aware file storing, in which is possible to store many kinds of evidences, from simple files to disk image.

Furthermore, it can support very large file, in fact it can use an addressing schema of 64-bit and the possibility to compress these files or images on the fly, because even if it can store thousands of TB, still there may be some limitations due to physical size of our device.

It supports secure ways to assure the integrity, like *secure digest algorithms* and even *digital signature schema* adopting certificate and it is **self-consistent**, meaning that in case of error it can identify that an error has been performed and at least try to recover to avoid possible loss.

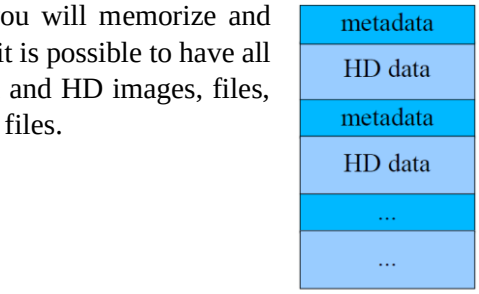
The AFF schema is organized in **type – value** pairs (where values can be very large). Examples:

- *Md5* = 17a5970a9baa83311f3e9d8a475c3c48
- *Imagesize* = "4290009873000"
- *Classification* = "Secret"
- *Gps* = "17.344523, 43.543567"

The type identifies what is the data associated to the value (e.g., result of a digest, metadata, some padding, some index, etc.) and if wanted it is possible to even extend the basic types in order to add possible information for a specific investigation.

AFF storage format

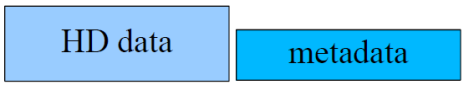
In general, there are many different possibilities regarding how you will memorize and what's the internal organization of the storage format. For example, it is possible to have all the metadata associated to the evidences in a sequence of metadata and HD images, files, and so on in a single place, or it is possible to have them in different files.



It is possible to have 4 different storage formats:

- AFF and AFD embed meta-data and raw data
- AFM and AFXML separate raw and metadata

The last one uses XML format, in case meta-data and images are put in specific tags.



AFM and AFXML

AFF4 imager

In the most common computer forensic distributions (Linux based) there will be also a set of tools to convert from or to images of files or set of files (directories and subdirectories) to a file in *AFF format*. Some examples are:

- **Acquire a disk image**
  - $aff4imager - i\ /dev/sda\ - o\ /tmp/output.aff4$
- **Acquire multiple logical files**
  - $aff4imager - i\ /bin/*\ - o\ /tmp/output.aff4$
- **See metadata**
  - $aff4imager - V\ /tmp/test.aff4$

Since the old versions of AFF are still in place and still useful, there may be, in some distribution, a previous version and slightly different tools. The most recent one is version 4, and it is full provided by the tool *aff4imager* (it is possible to query a file, convert, put a set of files in an output file etc…).

NIST NSRL

It is the *National Software Reference Library*, supported by NIST and some U.S Department of Homeland Security, FBI state and local law enforcement. It "*promote efficient and effective use of computer technology in the investigation of crimes involving computers*".

It has the purpose to improve analysis of what can be found in the storage media. It collects software profiles from various sources and builds a *Reference Data Set (RDS)* of information. For example, a collection of digital signatures of known software applications.

The purpose of this project is to target free and popular applications ("good" apps) tagged with application title, version, vendor, OS, etc… Anything useful to query and find it. The point is to **be able to identify something that is good** (the opposite of an antivirus) and to **categorize good applications by "type"**.

It is important to do this because in a real case it is usual to have an image with thousands of files and application that are just normal app and not interesting for a forensic point of view. This means that the purpose

is to have a hash that enables to identify an application that is good, so this is a kind of first step in a forensic analysis and it is performed by many suites case, it is used to filter out all popular and good application that are not interesting.

Autopsy

It is an open-source digital forensics platform, probably the most famous, that allows to perform a huge set of operations: for example, autopsy can connect to the NSRL, allow teamwork, co-operation between different platform. It is an **extensible** framework, so it collects many open-source tools and connect them through a Java-GUI.

Autopsy workflow

Autopsy provides the workflow to approach a case, composed by steps:

1. **Creation of a case**: choose a name and provide some metadata (e.g., authors of the analysis)
2. **Select data sources**: choose the evidences for example files, HD images and VM content, so the content in the virtual disk, the memory of the VM extract it and make it available for the analysis.
3. **Import data**: autopsy will automatically perform and create some integrity check (hash of images chosen). Autopsy has also a set of different modules that it is possible to select in order to perform some automatic operation like find some suspect stuff (like if a file has an extension but is of a different type).
4. **Data analysis:** a first visualization is shown, and it is possible find all the results categorized by type, suspects, something that has been recovered and this is in unallocated space (maybe if it has been deleted), and so on… It can even identify places of the image where there is some encrypted material (encrypted data have a very high set of entropy), and it is possible to set even more type of analytics and statistical analysis.
5. **Report generation:** finally, you can generate a report in HTML / XML.

Autopsy import

Autopsy can import many open sources file systems (NTFS, FAT, Ext* …) and commercial one (if they are known by a reasonable amount of time) such as windows and mac.

Autopsy provides also **interactive provision of the results**, since an analysis can take hours to reach the end, it is possible to have partial results that has been found even if the analysis is not finished.

Also, it is possible to start with default properties which give priority to specific parts of a file system, but it is possible to change this priority in case there is the suspect that some variable data can be found in a different place.

Here some operation that are permitted by default:

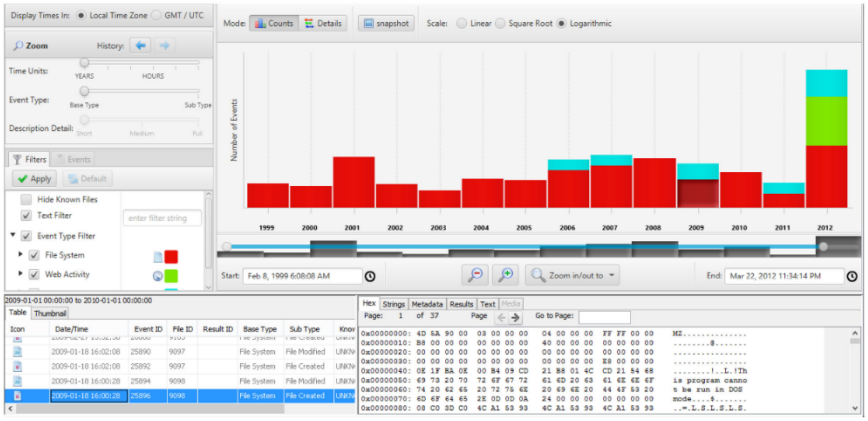
- **Recent activity extractor**: it can show, for example, an action saying, "*at this point in time the user accessed the email*". It is possible to find also, if it is still available, the email that has been sent by the user (and its content).
- **Hash calculation and lookup:** it is possible to calculate the hash of the different files and compare it to the value in the NIST NSRL database of known files.
- **File type identification**
- **Extension mismatch detector:** it finds mismatch between the extension of a file and the real format of a file.
- **Embedded file extractor**: extracts something which is embedded inside the storage and inside other files.
- **Email parser**: it can create a timeline of the email that are present in the storage.
- **Keyword search**: it can search (with a regexp) something between the string present in the storage.
- **Exif parser:** it can identify the metadata of different file
- **Encryption detection**

- **Interesting file identifier**: they are files that match specified criteria. Each set has a list of rules, which will match on their chosen file characteristics. File need only to match one rule to be found.
- **Correlation engine**: it correlates results between cases
- **PhotoRec carver**
- **Virtual Machine extractor**
- **Data source integrity**
- **Android analyzer**

Timeline

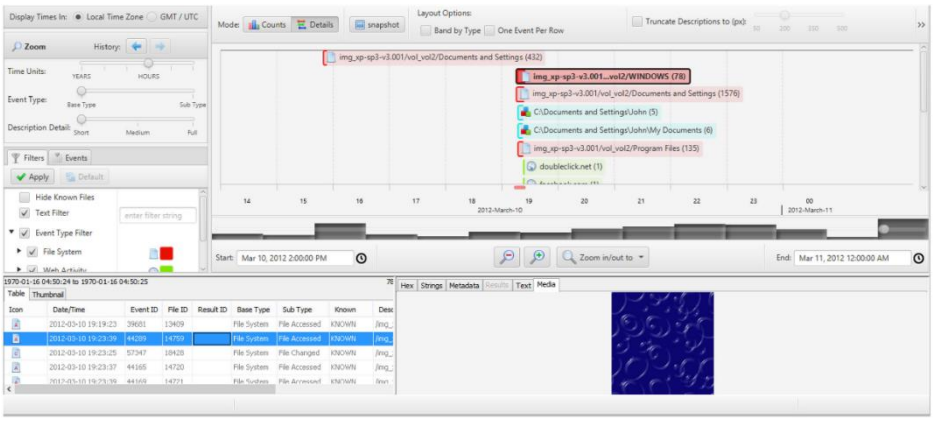
In the forensic analysis one of the most important parts is something that allows to reconstruct and see the timeline, meaning reconstruct when an action occurred and so understand the sequence of actions. It is important to figure out what a criminal performed.

Timeline is a **central visual tool** able to answer questions like "*when a computer is used? What events occurred before/after another one?"*. It gathers timestamps from different sources such as *files, web artifacts, meta-data* (EXIF, GPS, …) and they are used by the **correlation engine** inside autopsy.



The picture shows that the image was created at the end of the last century, and it was used until 2012, where most of the events occurred.

The different histograms are related to the number of events and different colours are associated to different action (red: file system and so on). For each of them it is possible to choose an event and through the help of the visualization tool it is possible to explore the text, the metadata related to the event, and any value inside the event with a hexadecimal editor. The first version of this timeline can be created automatically by you just running autopsy. I t will try to acquire all data and create the sequence of them.



It is possible to have another view of the timeline, with another granularity (even of hour of the different operation). It is possible to detail a specific day of the analysis, and inside there it is possible to see the hour in which different operations are performed. By clicking a specific event it is possible to check details about it. For example, if there is a picture it is possible to see a preview of it, still inside the different kind of visual tools that autopsy provide.

Computer forensics phases

Many different models were developed during last twenty years to define different *Computer Forensics phases*.

They are roughly grouped in:

- **Source identification and collection**: identification of your perimeter
- **Data acquisition**: in a secure way without introducing error and manipulation
- **Data analysis**
- **Report creation and result presentation**: very important to demonstrate the result of the analysis, if it is not clear for the target of the report (a judge, a project manager, the owner of an organization), then the work will have no real results.

There are also some **corollaries.** For example, get the approval from relevant authority of what is going to be performed (e.g., sniff communication between two devices).

Source Identification and Collection

It is the phase where the boundary of the data collection must be draw. It is difficult because from one side there is the acquirer of all the relevant sources of data, but if it is acquired more than what is needed there can be some trials. In this phase it is important to take right choices and to be able to demonstrate these choices them in front of a judge for example. Then, of course, the items must be sealed in order to preserve all the possible source of data. So, there are **two contrasting needs**: gather all necessary evidences, respect legal limitations.

Data acquisition

After sources are identified and possible secured, the evidences must be acquired carefully avoiding possible modification (willing or not willing). For example, just trying to understand the list of files present on a machine before having acquired the image of a machine might change some meta-data such as the last access of a file.

During this phase an adequate procedure must be provided, to:

- *Preserve*
- *Record the evidence life*
- *Record who is in charge to interact with the evidence*
- *Make the evidence available whenever required*

It may significantly differ on the context and device status (e.g., switched on or switched off)

Data acquisition: switched on example

Switched on case is trickier. The required steps are:

- *Take a photo of the monitor*: needed to get as much information as possible and understand what the state of the machine was when it was acquired, using a different device (not a screenshot).
- *Take note of time*
- *Gather data in proper order (volatile priority)*: already discussed, starting from registers, in order to acquire immediately data that might no longer exist after a few seconds/minutes.
- *Create a bit-for-bit copy (to capture "ambient data")*: trying to get all the bits of information using tools, but preferably to perform it as quick as possible using the version of tool that also compute the hash of the acquired data.
- *Check the cloned image*
- *Switch off (brutally) the computer*: unplug the cable (remove battery if there is one), in this way is possible to try maintaining the state of the device.
- *Take a photo of the crime place:* to reconstruct exactly the connection of the devices (if connected by cables)

- *Disconnect cables*
- *Transport safely the components*

Data acquisition: switched off example

In this case, data must be booted only in a **controlled environment** from a live CD or a floppy or a pen drive. Possibly, sub-parts of the system can be connected to safe devices (HD to a portable forensics station). A physical tool to acquire image is useful because it is possible to setup something even more robust such as physical write-block in order to be sure to avoid any kind of evading technique to corrupt data.

Data preparation

Often evidences are huge amount of data (terabytes). Data must be organized in useful chunks: **indexing operation** (to categorize all different sources)**, data-mining activities**. It must be performed before data analysis.

Data Analysis

This is the core of the forensics phases. Here is where the main operations take place:

- *Recovery of erased data*
- *Data-decryption* (password cracking based)
- *Recovery of hidden data* (slack areas, sector gaps, steganography)
- *Recovery of temporary data* (office temporary files, /tmp, swap files)

Reporting

In this phase there will be written trace of everything the investigator found during previous phases. It is very important a careful description, that will be used during legal proceeding. The written traces are authenticated through **digital signature**. Also, timestamps are authenticated for data recording of operations.

Italian best practise

The best practices are reported in the "*Manuale operativo in materia di contrasto all'evasione e alle frodi fiscali*" vol. II, 2018 book. It is a collection of best practises and how to perform the different parts of a computer forensics analysis. It is possible to have there all the different specifications and guidelines. It is implied the figure of *Computer Forensics and Data Analysis (CFDA)* which is something specifically trained to perform this kind of analysis, usually inside the "Polizia Postale".

This book explains the following steps:

- First make a **forensic copy** and **locate backups**: useful to identify last-hour modifications/erasure
- Use the **EWF** or **AFF format** for the copies
- Fully **acquire the data** (meta-data included, e.g., email headers) adopting specific devices (e.g., writeblocker) for anti-data-corruption and adopting specific applications (e.g., specific distributions like DEFT, specific software like FTK imager).
- **Record meta-data about the acquisition**: action taken, name of involved agents, place and date of the seizure, involved personnel of the organization under investigation, list and types of the digital evidences, digital evidence hashes, agents involved in the chain of custody, place of custody of any digital evidence.

SANS's suggested acquisition order

It is an organization devoted to the response of incidents. It suggests:

- *Photograph the computer and the scene*
- *If the computer is off, do not turn it on*
- *If the computer is on, photograph the screen*
- *Collect live data*: start with RAM image and then collect other live data "as required" such as network connection state, active users, running processes, …

- **If hard disk encryption detected** (using a tool like *Zero-View*) such as full disk encryption i.e., PGP Disk, then collect "*logical image*" of hard disk using *dd.exe* or *Helix* locally, or remotely via *F-Response*.
- **Package all components** paying attention by using anti-static evidence bags
- **Seize all additional storage media**
    - Create respective images (and place original devices in antistatic evidence bags)
- **Keep all media away from magnets, radio transmitters, and other potentially damaging elements** (e.g., heaters)
- **Collect instruction manuals, documentation, and notes**
- **Document all steps used in the seizure**

Chain of custody

It is another important concept to show that primary data have not been altered. The NIST definition says: "*A process that tracks the movement of evidence through its collection, safeguarding, and analysis lifecycle by documenting each person who handled the evidence, the date/time it was collected or transferred, and the purpose for the transfer*". Anything that happened to evidence must be tracked from the time it has been acquired until now, providing traces of all the actions and why any action has been performed. It is needed to **avoid any alteration of the original data**, **compute hashes to demonstrate identity** using more than one algorithm (collisions!) e.g., MD5 + SHA-256, **seal the original (with handwritten label) and store securely, create signed note of participants, procedure, values, storage location, and person in charge of custody.**

Privacy protection

Privacy has big relevance nowadays; we're always continuously signing either physical in paper or virtually some disclaimers. European Union is subject to **GDPR**.

GDPR

It stands for *General Data Protection Regulations,* and it applies to **all countries in EU**. It replaces *Data Protection Act* of the 1998. It is effective from *May 25th, 2018*. It's a **Regulation**, which means that it is composed by **Articles** (the law itself) and **Recitals** (explanatory notes within the body of GDPR). It is not usual for Italy (usually there is the Law and then the interpretation is up to the lawyer), since in this case there is the article and the recital.

There is a specific body within the EU named the **Article 29 Working Party** which is the **central guidance body for GDPR**. That's a set of delegates (one or more for each European country) that perform overall surveillance over the GDPR. The GDPR has got one "*Supervisory Authority*" in each country, which is the **national Privacy Officer**. For example, in Italy it is "Garante per la protezione dei dati personali", in UK it is "Information Commissioner's Office (ICO)".

It is important because the regulation applies throughout Europe, but the implementation and the surveillance are made at national level, so there must be a national person in charge (Supervisor Authority).

To understand what we are dealing with in GDPR, there are some *definitions*:

- **Personal data:** it is any information related to an **identified** or **<u>identifiable</u>** (IMPORTANT POINT!) living person. In some case some data might not identify that specific user but, if through some inference or with additional data that information can be related to the user, then it is subject to the GDPR. Some examples:
    - *HR records:* when entering Politecnico or as employee in a company there is always a section/department which is managing all the employees/attendees. That information, for example, must be stored only if the company has that employee.
    - *CCTV images of a student:* e.g., during an exam or while traversing the Politecnico
    - *Photograph of myself*
    - *E-mail with me cc'd:* not directly addressed to the cc'd user but still involved in that email.
    - *Confidential opinions written about me by my supervisor:* they are confidential but nonetheless they are related to me. Even if I don't know them, then there must be a protection for that information.
    - *Even anonymised monitoring data*: they are considered personal data even if they are not directly related to a person, because through the analysis it might be possible to identify a person.

  This definition of personal data applies for both **automated collected** and **manual collected** data. Normally we think about privacy related to computer, but it is not only that. If there are similar information collected by hand in a piece of paper, you must protect also that piece of paper.

- **Sensitive personal data**: this corresponds to *Special Category Personal Data* (art. 9.1) of the GDPR regulation. They are related to some characteristics that may create problems. The exhaustive list is:
    - *Racial/ethnic origin*: disclosing information about racial or ethnic origin is considered inappropriate due to racism problems.
    - *Political opinions*
    - *Religious/philosophical beliefs*
    - *Trade union membership*
    - *Genetic or biometric data*
    - *Health*
    - *Sex life / sexual orientation*
    - **No other sensitive personal data**. They are only those falling in this category.

Criminal offences/convictions are not included but separated out and similar extra safeguards put in place at Art. 10. It means that they have additional protection, because that is considered even more important than normal sensitive data.

When there are data managed inside a company which is subject to the GDPR some terms are used:

- **Data controller**: says how and why personal data is processed. The one that is controlling data and decides in which way and why personal data are processed.
- **Data processor**: it is the executor, the one that processes the data on controller's behalf.
- **Processing of data**: it is any kind of activity with personal data, including:
    - *Collecting*
    - *Storing*
    - *Using*
    - *Deleting*
    - *Sharing*

Data properties

These properties must be guaranteed in order to protect the privacy.

Data shall be processed:

- **Lawfully:** must not be in breach of other laws and must be lawful in accordance with Art. 6 & 9.
- **Fairly and transparently**: data subjects made *aware* (e.g., typically through privacy notices we know that data are being processed in a certain way). Companies which say "*give permission to process your data*" in generic way is not a valid statement. Companies must give the specific kind of processing that are applied to data, otherwise notification is not valid. User must *feel being treated fairly*. For example, there must be no discrimination if data is not provided, compared to the user that provided the data.

For **purpose limitation** data shall be collected for:

- **Specified**, **explicit** and **legitimate** purposes and not further processed in a manner that is incompatible with those purposes. When requesting consent, it **must be for a specific purpose and it's not valid for any other one**. This applies to any company in Europe. Lioy said that many times companies give a document of GDPR which is invalid (due to missing fields such as data or name of who was in charge at the time the document was created/updated) but also not updated (many times the network in the company changes but they do not ask again the user for a new permission) and it must be updated and then the user must be notified, since the consent of the user is valid only for a specific purpose.

Other protection that data should have to be protected (other properties):

- **Data minimisation**: data must be **adequate**, **relevant** and **limited** to what is necessary in relation to the purposes for which they are processed. It means that while designing an application it is required to *do not exceed in data collection*. For example, an airline selling flight tickets must ask for name, surname, birth data, etc. but there is no reason for which it should ask if user is married or not, or if it has sons or daughters and so on, since it is not relevant.
- **Data accuracy**: data shall be **accurate** and, if necessary, **kept up to date**. This is tricky because after collecting data, they must be periodically reviewed, and an update must be performed. Of course, if the user does not notify a company that he changed the e-mail address that is ok, but for example it could be possible to send periodically an e-mail with a reminder to confirm the e-mail address. If no answer is provided to this email, then that address must be cancelled, since it means that it is no more used. Basically, it is needed to design and perform update procedures.
- **Storage limitation in time**: data shall be kept in a form which permits **identification of subjects for no longer than is necessary** for the purposes for which the personal data are processed. If an airline

sells a ticket, data must be kept for sure until the flight has been completed, maybe some months after that in case of any complain about the credit card payment or an incident happened on the airplane. The time must be specified, and it must be a reasonable one. Company must also explain why that time has been selected. Then, company must design a process for periodic cancellation of "old" data.

Breach reporting

There is not only the obligation to protect data, but also the obligation that if something bad happens the users must be notified quickly that there has been a problem, named *breach reporting*. **Personal data breach** is a *breach of security* leading to the *destruction*, *alteration*, *unauthorised disclosure* of, or *access to*, personal data. Think, for example, if someone can enter in a hospital database and delete/change/can have access to the results of a clinical examination.

When bad things happen to data, it is needed to notify the Supervisory Authority (in Italy Garante della Privacy) where it is likely to result in a risk to the rights and freedoms of individuals (within **72 hours** of being aware of the breach). It can be any day in the year, but they are always 72 hours to notify both Supervisory Authority and notify individuals where it is likely to result in a high risk to the rights and freedoms of individuals. For example, if there is a database of a person belonging to a certain political party, it must be immediately informed (if data are read from database, for example).

Data transfers

Sometimes personal data need to be moved to another country where the GDPR does not take place, but it protects also that kind of transfer. GDPR imposes restrictions on the transfer of personal data outside the **EEA (European Economic Area)** which is more than European Union, to third countries or international organisations (e.g., United Nations).

The European commission may designate some non-EEA countries as having **adequate level of data protection.** Otherwise, if there was no declaration of equivalence, the transfer can take place only if there are appropriate safeguards (that can be technical or legal) such as:

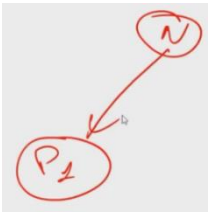
- *Agreements*: for example, the *Standard Commercial Clauses* (SCC) which applies when an iPhone/Mac is bought, for example in Italy. While buying it, the user implicitly signs and accepts the Standard Commercial Clauses, since personal data are being transferred to the USA. That would not be possible unless the user gives that authorization, because Apple has only one central management system for all the shops in the world and that is in the USA. This is a legal safeguard.
- In the past there was an agreement between EU and US called *Privacy shield* which said that when data are transferred between Europe and US the technical solutions reported in the document must be applied. Now the Privacy shield is no more in place because US adopted new regulations that say that the American Government has the right to access the data of any citizen when data are stored on the US territory, even if they are not American citizens. This is tricky because a non-US citizen is not subject to the American government so why they should have the right to access data? This creates big troubles in cloud computing, because in CC we don’t know where data are stored. Now when an infrastructure is created it is possible to limit the geographic area in which storage and computational resources are deployed, just to avoid those kinds of things. China and many other countries have a similar rule.

There are specific requirements about these agreements between *controller – controller*, or *controller – processor*. This is just to tell to be aware that what is being described as GDPR only applies within the EEA.

Information Lifecycle Management

In order to comply with GDPR or, in general, to protect the privacy, the company must implement an *Information Lifecycle Management* which means that specific steps must be performed to protect the privacy of the data that are managed in a company. It consists of:

- **Information Asset Registers (IAR)**: it is needed to have a register to collect and list all the information assets that are present in the company. This can be *cyber* information but also *paper*-based information and both must be managed.
- **Data Flow Mapping (DFM)**: listing the places where the data are stored is not sufficient. There must be also a DFM that is: if there are some personal data (for example the "N" in the picture which stands for the *name*), if it is processed by a certain process there must be a diagram like the one in the picture, which shows where the data are stored, where the data are moved and where data are processed. This is the data flow. The requirements are not only protecting the data when they are stored but also protecting them when they are in transit and processed. Without this DFM in case of an audit it must be demonstrated that the company performed protection.



- **Risk Assessment(s)**: for example, if the node P1 (in the picture) is in the cloud, it is a risk, so it must be identified and then later decide how to mitigate this risk.
- **Privacy Notice(s)**: the user must be informed with adequate privacy notices about all these things (the data will be stored for this amount of time, they will be processed in this way, maybe they will be exported to the cloud, etc.).
- **System Level Security Policy (SLSP)**: starting from the risks the policies for security must be created. There can be some risks that are not covered only if they are considered not harmful and it has to be demonstrated in case of an audit.

EU-GDPR art. 25 par. 1

"Taking into account the **state of the art**, the *implementation cost* and the *nature, scope, context and purposes of processing* as well as the **risks** of *varying likelihood* and *severity* for rights and freedoms of natural persons posed by the processing,

the controller shall, both at the time of the determination of the means for processing and at the time of the processing itself, implement **appropriate technical and organisational measures**, such as pseudonymisation,

which are designed to implement **data-protection** principles, such as data minimisation, in an effective manner

and to **integrate the necessary safeguards into the processing** in order to meet the requirements of this Regulation and protect the rights of data subjects."

Brief explanation of the text:

- The first paragraph talks about *costs* because if there is a small company, GDPR is not pretending that you spend more than you earn. This means that the protection is adequate to the value of the data and to the income of the company. Also, *risks* are mentioned because, as we saw before, we list storage, transmission and processing and then risks are evaluated.
- The second paragraph says that first when a decision is taken, but also day by day (since the time of processing is every day) there must be appropriate measures.

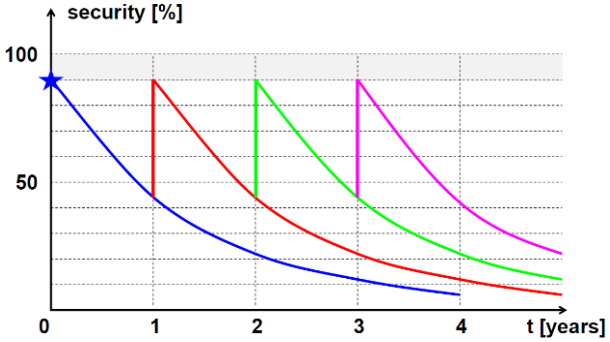
State of the art

We know that there are new attacks (millions per month) and the *windows of exposure* (patching may require months or never happen, e.g., Windows XP). This means that we must continuously think about windows of exposure and that in some cases there may be an indefinite duration of the window of exposure. For Windows XP no patches are available, so it must be protected in other ways. It is needed to have **periodic review and update**: it is not enough to have performed security analysis, risk analysis and defining security architecture

e.g., three years ago when the system was created, because the conditions change continuously. With the GDPR there is no fixed deadline for performing a review.

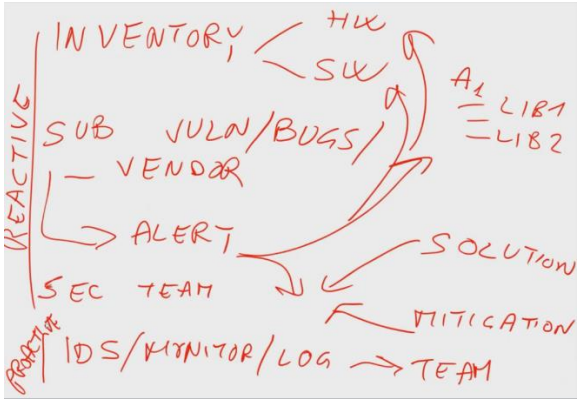
State of the art - periodic review

Imagine the system was created at time 0 achieving a certain amount of security which cannot be 100%. Given the number of new attacks, vulnerability, malware, etc. the statistics tells that we are losing nearly half of the effectiveness every year. In at most 4 years the security drops towards 0. The review should be done periodically in a time frame less than 1 year.



Recently a new vulnerability "*Log4J*" affects a huge number of applications, which is a remote code exploitation, which means that the attacker can inject code and execute it in the machine. It is not possible to wait 1 year to fix that, but immediately. This is where the organisational part is involved.

In addition to having the mapping, processing and transfer of the data, the company needs an **Inventory** in which are listed all the assets (both HW and SW) where for each software asset it is needed to know all the components and libraries from which it depends. Then it is needed **subscription** to *vulnerability/bugs*, if possible, by the vendor itself. This is not enough, since if an alert from the subscription is received it must be performed a check on all SW, libs, HW in the company. That is an **organization**, it means that there is someone that, maybe daily will promptly check the inventory. After that it is needed a **security team**, which will implement the solution, if it exists. If there is an attack but not a solution, it is needed to find a way to **mitigate** (e.g., Windows XP by putting it behind a firewall).



The first part is **reactive**, which means that there is a react to an announcement of a new vulnerability/bug/malware. It is possible to be **proactive**, so prevent by using *IDS/monitor/log* that need a dedicated team of analysts.

EU-GDPR art. 25 par. 2

"The controller shall implement appropriate technical and organisational measures for ensuring that, **by default**, only **personal data which are necessary** for each specific purpose of the processing are processed. That obligation applies to:

- the **amount** of personal data collected,
- the **extent** of their processing,
- the **period** of their storage,
- and their **accessibility**.

In particular, such measures shall ensure that by default personal data are not made accessible without the individual's intervention to **an indefinite number of natural persons**."

Explanation:

- "**by default**" means that you must do anything to put security, it must be already there. The default state must be secure (no option). Second, you must process only the data which are necessary for each specific purpose.
- "**an indefinite number of natural persons**" is a specific text that has been inserted in GDPR for social media. It means that any kind of social media (not only Facebook, Instagram), when data are

inserted, they must be available for yourself, then you take an explicit decision for who can access to them. The default is the most restrictive one.

EU-GDPR art. 25 par. 3

"An **approved certification mechanism** pursuant to Article 42 **may be used** as an element **to demonstrate compliance** with the requirements set out in paragraphs 1 and 2 of this Article."

This paragraph says that maybe in the future will be created some certifications, but it is not compulsory, since it "*may be used*".

The Article 42 says:

- *(par. 3)* "The certification shall be **voluntary** and available via a process that is **transparent**."
- *(par. 4)* "A certification pursuant to this Article **does not reduce the responsibility** of the controller or the processor for compliance with this Regulation and is without prejudice to the tasks and powers of the supervisory authorities which are competent pursuant to Article 55 or 56."

A company spend money and time to be certified but then it cannot say that it is not responsible. It is never possible to certify a system 100% secure. Typically, there are some limitations, related to time, budget, etc. Even a certification can give some assurance, but not a fully one.

Currently almost nobody is certified for GDPR.

EU-GDPR art. 46

Transfers of data abroad are subject to appropriate safeguards with various conditions that can be technical or legal.

Privacy by default

GDPR requires privacy by default. It means:

- The **most restrictive settings must be applied automatically** when a customer buys a service or a product. The customer must not request the privacy of its data, that must be automatic. Eventually, the customer may explicitly request that some of its data are unprotected.
- The **data must be stored only if they are needed to provide the specific service**. The customer must not make any action to have its data cancelled once the relation with the service provider is over. The key point is "*relation*"; it should be discussed legally when the relation finishes.

Privacy by design

It means that the system has been designed to protect the privacy, so each business service or process that uses personal data must consider their protection in each design phase and implementation too. So, privacy should not be added later but considered since the beginning. We talked about this in the first course saying that we must design security since the beginning, any stage has got its part in security.

The data controller/processor *must be able to demonstrate* that there are **adequate security measures**, and that **privacy principle is continuously applied and verified**.

The IT department must protect personal data during the whole lifetime of data, systems and processes.

Frequently privacy (and security) is not considered in the initial phase of IT projects but when we talk about privacy, we must remember *Ann Cavokian* (ex Information and Privacy Commissioner of Ontario, Canada) that defined the **seven principles** of PbD accepted all over the world. They are:

1. **Proactive not reactive; preventative not remedial**
2. **Privacy as the default setting**
3. **Privacy embedded into design**
4. **Full functionality**

5. **Full lifecycle protection**
6. **Visibility and transparency**
7. **Respect for user privacy – above all**

Since they are considered compulsory for any privacy implementation, they will be all discussed.

PbD #1: Proactive not reactive

Be proactive to avoid risks rather than limiting the damages, prevention is better than cure. It means to apply **strong** and **consistent** techniques, since conception. Furthermore, *risk evaluation for privacy* is different from risk evaluation of security, for example DoS is relevant for security but not for privacy since data has not been touched and otherwise data confidentiality in an internal network may not be important for security, since it is assumed that the network is trusted, but it is for privacy.

So, risk analysis for security is different from risk analysis for privacy.

PbD #2: Privacy by default

There must be default settings in all operations and configuration. Data collection must be **correct**, within legal boundaries, and **limited** to what is needed. When applications are designed, information, and communications use **non identifiable** interactions and transactions. Furthermore, **minimize data identifiability, observability, and associability**. Observability means that it is possible to read the data. Identifiability means that by looking at the data it is possible to identify a real person. Associability means that by looking at different data it is possible to associate them.

Data must be **automatically protected** so there must be no need of special procedures and no need of special input from the data subject.

PbD #3: Privacy embedded into design

Privacy must be embedded into the *design of the IT architectures*, into the design of the *business processes* (which is how we want to use the data), into the technologies used in the design, into the design of the operation and must be all in a *holistic* and *integrated* manner, which means everything implemented at the same time and together. Privacy must not be a last-minute addition, which is typically messy and ineffective.

An important issue here is: how to deal with existing systems and applications not designed like that because the law came afterwards?

If it is possible to demonstrate that the system was designed before GDPR entered in practice, then that system is exempt from this specification, but if a redesign of that architecture or application is made, then it must keep into account this requirement from GDPR.

PbD #4: Full functionality

It means that **privacy** must be a **basic component of the design**, without any diminution of its functionality and it must **avoid trade-offs** (e.g., privacy versus security) without giving the user the choice to have a functionality or the data protection. Implementing correctly the privacy controls will make benefits to the whole system (e.g., less need for other security controls). Considering privacy will lead you to implement things that will be useful also in security. It is a **win-win situation** (or at least with a positive outcome rather than a neutral one).

PbD #5: Full lifecycle protection

Privacy must be considered during the **whole lifecycle** of the personal data. There must be no "holes" in the protection or accountability. It means that privacy must be considered when the system is set-up before collection, during data processing and storage and when data are cancelled (because as we seen during forensics lessons data can be recovered).

One of the most important things in a company is that when laptops, smartphones, disks are being destroyed, the must be physically destroyed to protect data. Very often companies give laptops or other devices to schools and students, which can recover data. So, companies need to have a specific protocol for dismissing their

devices. Otherwise, specific programs to cancel data at a very low level are needed. Security is highly important in all these phases because it is not possible to have privacy without a solid security foundation.

PbD #6: Visibility and transparency

This says to follow the **"trust but verify" principle** which means "*I trust that we have done the correct thing but let's perform verification*".

An example of this is: I got a firewall and I trust it to do the correct things, but then I have an IDS which is checking that what should be blocked by the firewall is not passing. So, I trust the firewall but I'm verifying if it's doing the correct thing.

Since the design is usually done by the company the verifier should be an **external** one, that must test how the company has designed security if it's correct and if the implementation matches the design. Also, the *operations* must be checked since the design follows the implementation but then there is a day-by-day activity in which some configurations and updates are performed. Also, this verification must be external.

The adopted operations and solutions, technical or procedural, must be transparent for the data subject and service providers.

PbD #7: Respect for user privacy – above all

This is a general concept and must be a **guiding principle** (*correct privacy management*). There must be **strong privacy protection**, we must **handle notification of problems** in an appropriate and timely manner as there are obligations to notify immediately the user if there's a problem with the privacy, and we must **use user friendly mechanism** for information and verification.

We must adopt a **user-centric approach**, which many companies usually don't do. The user must be the center of your thoughts because privacy is a matter of your users.

PIA (Privacy Impact Assessment)

While implement previous things, the law requires to implement PIA, which is a procedure similar but not identical to the *risk analysis*. The system is studied not to understand if there are generic security risk but to see what the impact on privacy of the various solutions is that we have implemented. This is explicitly required by the GDPR, so an inspector can ask to a company for its PIA.

Inside the PIA the phases must be well documented, and they are:

- Identify personal data and discuss them with the stakeholders, it means *"Is this data really needed by the stakeholders controlling the business process?"*
- Identify risks, keeping into account the stakeholders' perception
- Given the risks, identify good countermeasures
- Given the countermeasures, define the protection rules
- Implement countermeasures and rules
- Once you have performed the implementation, create rules and mechanisms for review, audit, **responsibility** (which means who is responsible of doing something)

The accountability principle

This is another thing required by GDPR. The data controller must be able to:

- demonstrate to have adopted a complete set **of legal, organizational, and technical measures** to protect personal data
- demonstrate in an **affirmative** and **proactive** manner that the data processing is adequate and conformant to the GDPR.

In other words, as a requirement of GDPR it is possible to implement and write down what has been implemented in response to that article, so it is possible to show it to someone who comes to perform an audit. This also means that the specific person who made the implementation is accountable, so it is responsible if anything wrong has been performed.

This means, at least for the Italian companies, **passing from a formal approach to a substantial one**. In the past many companies were just claiming that they were protecting privacy by writing some papers, instead here companies must **demonstrate** that they have a reasonable measure to minimize the risk, and since the risk change and evolve over time you need to continuously adapt. Keep also in mind that the risks may change for external (e.g., new threats) of internals (e.g., new data and/or processes) factors.

Records of processing activities (art. 30)

The records must be in a **written form**, or **electronic**, in which are listed all the processing of the data that are subject to privacy.

That is not required for SMEs (small and medium enterprises, the ones that have less than 250 employees) "*unless the processing carries out is likely to result in a risk to the rights and freedom of data subjects, the processing is not occasional, or the processing includes special categories of data as referred to in Art. 9(1) or personal data relating to criminal convictions and offences referred to in Art. 10.*"

This is normally quite rare, so usually SMEs can skip the records of processing activities.

The specific requests are:

- (art. 30.1.f) where possible, the envisaged time limits for erasure of the different categories of data
- (art. 30.1.g) where possible, a general description of the technical and organizational security measures referred to in Art. 32(1)

EU-GDPR art. 32

Article 32 is dealing with technical solutions, and it goes back saying again:

"*Considering the **state of the art**, the costs of implementation and the nature, scope, context, and purposes of processing as well as the **risk** of varying **likelihood** and **severity** for the rights and freedoms of natural persons, the controller and the processor shall implement appropriate **technical and organizational** measures to ensure a level of security appropriate to the risk, including inter alia…*" (par. 1)

Explanation:

- The state of the art means which are the common attacks and vulnerabilities, the risk analysis means that you have considered those attacks and vulnerabilities and how much they are likely (probability) as well as their severity, if the attack takes place what is the result.
- "*Appropriate to the risk*" if the risk has not been defined, is not possible to know what is appropriate.
- "*Inter alia*": there are many options, also consider this. If there is something that is already listed in the regulation you better implement that, then it is possible to add other things (if wanted).

"*In assessing the **appropriate level of security** account shall be taken in particular of the risks that are presented by processing, in particular from accidental or unlawful de**struction, loss, alteration, unauthorized disclosure of, or access to** personal data transmitted, stored or otherwise processed.*" (par. 2)

Destruction and loss

We do our best to avoid data destruction and loss; but let's make arrangements for the worst case. If data are destroyed or lost it is needed a **good backup technique** and strategy. The backup must be done by using these guidelines:

- **Offline** (otherwise the backup itself may be attacked)
- **Offsite** (otherwise the backup itself may be hit in a disaster)
- Must have **minimal** or possibly **null manual operations** to minimize/avoid human errors
- **Periodic** (which data history can be reconstructed?)
- **Verified** (immediately after backup creation, for verification, and periodically, for technical obsolescence or support wear-out): after backup is created, immediately try to re-read it because if the external disk is broken it might not be detected until it is read.

EU-GDPR art. 32 par. 1

The suggested technical protection measures are:

- The **pseudonymization and encryption** of personal data
- The ability to ensure the ongoing *confidentiality, integrity, availability,* and **resilience** (continuing operations even if degraded way despite the attack) of processing systems and services.
- The ability to restore the availability and access to personal data in a timely manner in the event of a physical or technical incident
- A **process** for regularly testing, assessing, and evaluating the effectiveness of technical and organizational measures for ensuring the security of the processing.

$$= \boldsymbol{cybersecurity} + \boldsymbol{business\ continuity} + \boldsymbol{disaster\ recovery}$$

All these things together means that we need for privacy: **cybersecurity** of course because there is CIA and so on, but when it is written "*resilience*", "*availability*", it means **business continuity**, which means that even if there is a problem, the system can continue to work, and the ability to restore those things when physical incidents occur, means **disaster recovery.**

Business continuity and disaster recovery are different concepts, in which the first means that maybe the system is duplicated somewhere else and when technical problems occur, the system can continue to operate, while *disaster recovery* means that, for example, if there is flooding covering all Turin, maybe in 2/3 days we'll have a site in Rome and we'll be able to restart operations.

Anonymizations or pseudonymization?

In par. 1 pseudonymization is mentioned, which is quite different from anonymization.

- **Anonymization -** *optional*
  - Data that may lead to identification are completely removed, and it is impossible to identify the person, but using statistical techniques (habits, join of different category...) and in this case anonymization is not required.
    - E.g., Alice, Bob, Charlie → xxx, xxx, xxx
- **Pseudonymization -** *compulsory*
  - The real identity is replaced with a pseudonym and keep a table (under strict access control) which gives the correspondence between the pseudonym and the real identity. In case of need, it is possible to disclose the person behind the pseudonym. Of course, if it is possible to guarantee anonymization, it is much better.
    - E.g., Alice, Bob, Charlie > A, B, C

Confidentiality
Confidentiality can be achieved in two different ways:

- **Data encryption** – not required
    - Data are transformed so that they can be understood only after decryption with a specific "key", and it is called **intrinsic protection,** but it is not requested in the GDPR.
- **Access control** – required
    - Data are stored in some place, and in front of them there is a **guard** that checks access to data (including reading) and the access is denied to unauthorized people. In this case it is called **procedural protection** (requires also **log** and **audit**).

Data encryption may be a solution, but then keys must be properly managed (if keys are lost, then data is lost). Implementing encryption of all data has its risks. The option is access control, which is less intrinsically secure, but it is safer.

Integrity
We should be able to verify if the data has been altered. The questions here are:

- Transmitted data = received data?
- Read data = written/stored data?

If the answer is "No", it is important to:

- **Alarm**, this is an attack!
- **Get back the original correct data**, very important. (Is this possible?)

Note that to perform this kind of verification it is required a MAC or a digital signature. The digest must be protected either with a symmetric key or asymmetric key (depends about the speed, key management and so on).

We should be able to **avoid data alteration**. You should ask in your security policy **who** has the right to *create, write, modify, delete* the data. This has an answer in the access control (composed by *authentication, authorization, policy*) and verify the correct implementation and operation, as we do in confidentiality using **secured** log and audit.

Availability and resilience
It is especially important for those data whose owner may need to access at any time, also in real time, like medical data. In this situation, data and systems should be **redundant**, and it is needed to perform **monitoring**, not only for attacks (using an IDS) but also for **capacity exhaustion** (rightsizing), so build the system to manage requests also under emergency.

Other privacy problems
Network-level log data **may also affect user privacy**. Some examples are:

- **Ip addresses**, because they are somehow associated to a user via subscription or authentication via a captive portal
- **HTTP log** because the visited pages tell something about the user, and the parameter submitted may be sensitive
- **DNS queries**, because it's the base for access through most applications, and it's visible even for encrypted channels. In fact, even if TLS is used, and so it is not possible to see the whole communication, before connecting to a server there is a query to the DNS server to ask for the IP address, and it detects that someone is accessing that server. It is possible to use *DNS-over-TLS* to protect the user from sniffing. But in any case, there is a fight for offering open DNS nameservers, not to attack the client, but only to collect statistics. One example is the 8.8.8.8 DNS offered by Google.