Advanced Information Systems Security

Alessandro Milani

October 26, 2024

Contents

Chapter 1

TLS

1.1 Original SSL (Secure Socket Layer)

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

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

1.2 TLS achitecture

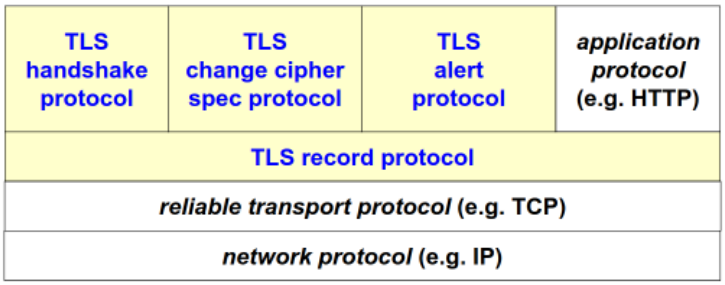


Figure 1.1: Arch TLS

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

1.3 TLS sessions and connections

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

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

1.4 TLS handshake protocol

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

1.5 Data protection

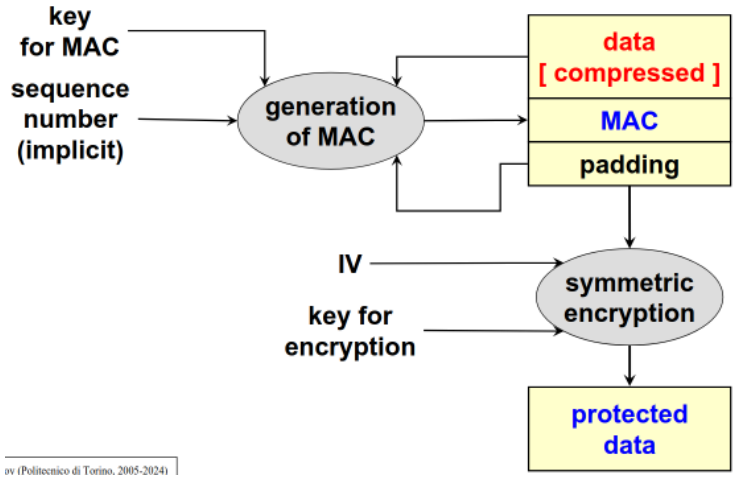
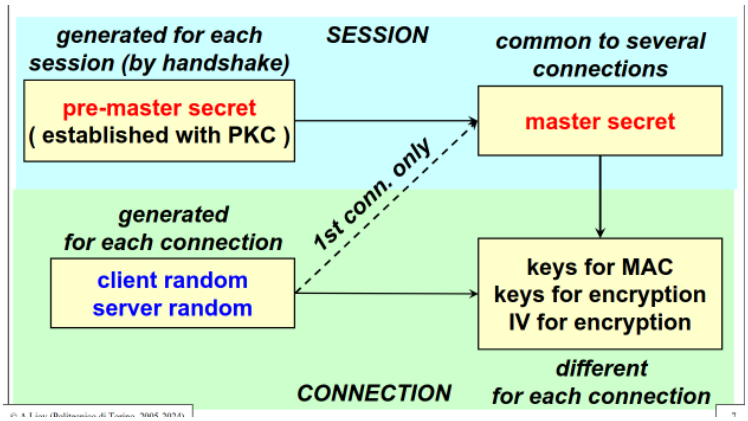


Figure 1.2: Authenticate then encrypt

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

1.6 Relationship among key and sessions



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

1.6.1 Problem

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

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

1.7 Perfect forward secrecy (PFS)

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

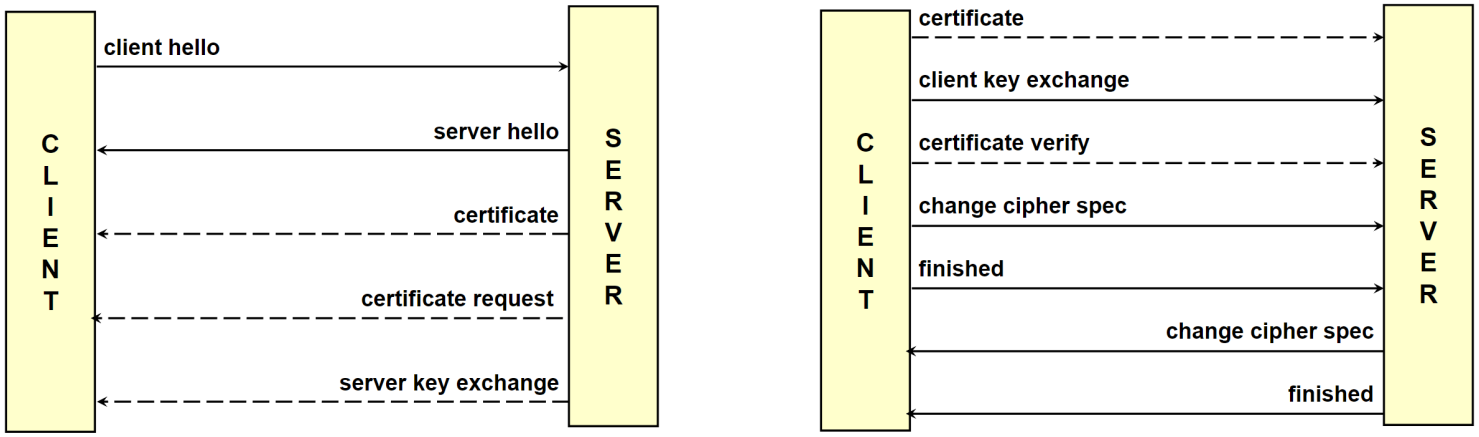
1.7.1 Ephemeral mechanisms

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

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

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

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



(a) Image 1

(b) Image 2

Figure 1.3: Steps of ephemeral key exchange

Client Hello

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

Server Hello

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

Cipher suite

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

  complete list maintained by IANA

Certificate (server)

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

Certificate Request

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

Server key exchange

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

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

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

Certificate (client)

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

Client key exchange

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

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

Certificate verify

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

Change cipher spec

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

Finished

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

1.8 TLS Comparison

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

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

1.9 TLS setup time

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

- (C > S) **SYN**

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

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

1.10 TLS versions

1.10.1 TLS 1.0 (SSL 3.1)

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

1.10.2 TLS 1.1

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

1.10.3 TLS 1.2

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

1.11 TLS evolution

1.11.1 Ciphersuites and Encryption

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

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

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

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

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

1.11.2 Compression and Other Features

The TLS evolution also includes advancements in compression methods:

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

Additionally, several other protocols and extensions have been established:

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

!! vedere attacchi da slide 34 a 41 !!

1.12 Attacks

1.12.1 Heartbleed

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

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

1.12.2 Bleichenbacher attack (and ROBOT)

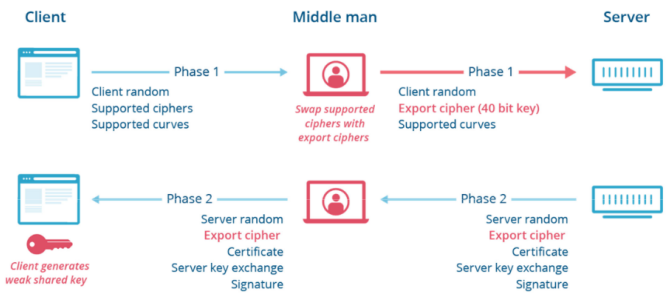
In **1998**, **Daniel Bleichenbacher** introduced the so-called "million-message attack," a cryptographic vulnerability targeting the way **RSA encryption** was implemented. By sending approximately a million specially crafted messages to a server, an attacker could deduce the server's **private key**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

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

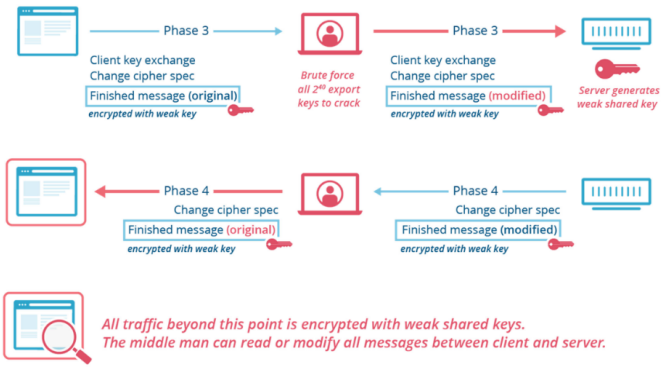
1.12.3 Other minor attacks

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

1.12.4 Freak phases



(a) Image 1



(b) Image 2

Figure 1.4: Freak Attack step by step

1.13 ALPN extension (Application-Layer Protocol Negotiation)

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

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

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

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

1.14 TLS False Start

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

1.15 The TLS Downgrade Problem

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

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

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

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

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

1.16 TLS Fallback Signalling Cipher Suite Value (SCSV)

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

1.16.1 Notes

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

1.17 TLS session tickets

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

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

There are some issues:

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

1.18 TLS and virtual servers

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

1.18.1 The problem

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

1.18.2 The solution

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

1.19 TLS 1.3

The goals of TLS 1.3 are to:

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

1.19.1 key exchange

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

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

1.19.2 message protection

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

Previous Pitfalls

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

TLS 1.3 Improvements

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

1.19.3 digital signature

previous pitfalls:

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

TLS-1.3 uses:

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

1.19.4 cipher suites

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

- **Avoids combinatorial complexity**: In previous TLS versions, the ciphersuite list grew exponentially as each new algorithm was added. TLS 1.3 addresses this by specifying only the essential orthogonal elements.
- **Orthogonal elements**: TLS 1.3 ciphersuites specify only two key components:
  - **Cipher (and mode)**: E.g., AES, ChaCha20.
  - **HKDF hash**: The hash used for the HMAC key derivation.
- TLS 1.3 does not specify certificate types (RSA, ECDSA, EdDSA) or key exchange methods (DHE/ECDHE, PSK, PSK+DHE/ECDHE), reducing complexity.
- **Only 5 ciphersuites** are supported in TLS 1.3:
  - `TLS_AES_128_GCM_SHA256`
  - `TLS_AES_256_GCM_SHA384`
  - `TLS_CHACHA20_POLY1305_SHA256`
  - `TLS_AES_128_CCM_SHA256`
  - `TLS_AES_128_CCM_8_SHA256` (deprecated)

1.19.5 EdDSA

Variant of the DSA.
DSA requires a PRNG that can leak the private key if the underlying generation algorithm is broken or made predictable, otherwire **edSA does not need a PRNG**, but **picks a nonce** based on a hash of the private key and the message, which means after the private key is generated there's no more need for random number generators.
This lead to a **faster signature** and verification wrt ECDSA (simplified point addition and doubling) and in general it use N-bit private and public keys, 2N-bit signatures.

EdDSA implementation

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

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

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

1.19.6 Other improvements

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

1.20 HKDF

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

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

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

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

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

1.20.1 HKDF in TLS 1.3

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

HKDF-Expand-Label

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

HkdfLabel is structured as:

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

Key Derivation Functions

- **Derive-Secret**:

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

- **Finished Key**:

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

- **Ticket PSK**:

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

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

1.21 TLS 1.3 Handshake

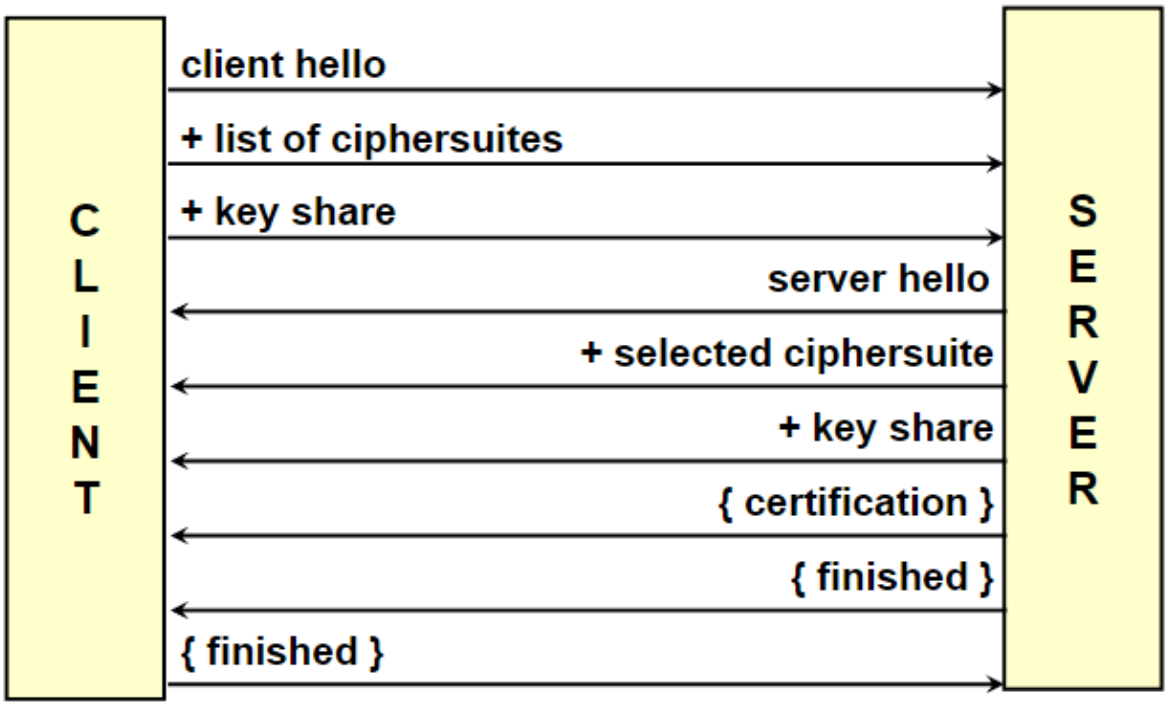


Figure 1.5: TLS 1.3 Handshake

1.21.1 Notes

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

- { data } = protected by keys derived from a

  sender

  _handshake_traffic_secret

- $$data$$

  = protected by keys derived from a

  $$sender$$

  _application_traffic_secret_N

1.21.2 client request

Client Hello:

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

contains extensions for key exchange:

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

1.21.3 Server Response

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

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

1.21.4 client finish

1.21.5 Pre Shared Key

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

1.21.6 0-RTT connection

TLS 1.3 - 0-RTT Connections

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

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

1.21.7 Incorrect Share

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

1.22 TLS and PKI

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

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

To check the revocation staus:

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

1.22.1 TLS and certificate status

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

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

Possible Approaches to handle these situations:

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

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

Pushed CRL

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

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

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

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

- Unauthorized Microsoft Digital Certificates
- Microsoft Security Advisory 2718704

1.23 OCSP stapling

1.23.1 Concept

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

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

1.23.2 Implementation

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

How it Works

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

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

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

Client Request for OCSP Stapling

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

Problems and Solutions

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

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

1.24 OCSP Must Staple

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

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

Benefits:

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

1.24.1 actors and duties

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

The **OCSP Responder** must:

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

The **TLS client** must:

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

The **TLS server** must:

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

The **TLS server administrators** should:

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

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

1.25 TLS status

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

Chapter 2

SSH

2.1 Introduction

2.1.1 history

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

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

2.2 Architecture

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

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

2.2.1 Transport Layer Protocol

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

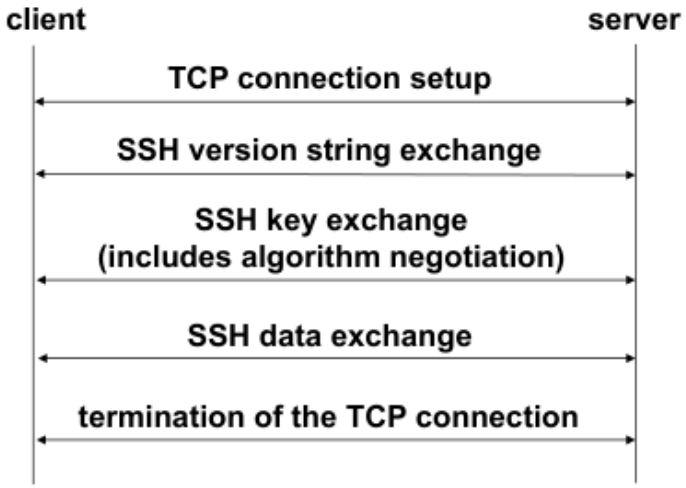


Figure 2.1: SSH Transport Layer Protocol

Binary Packet Protocol

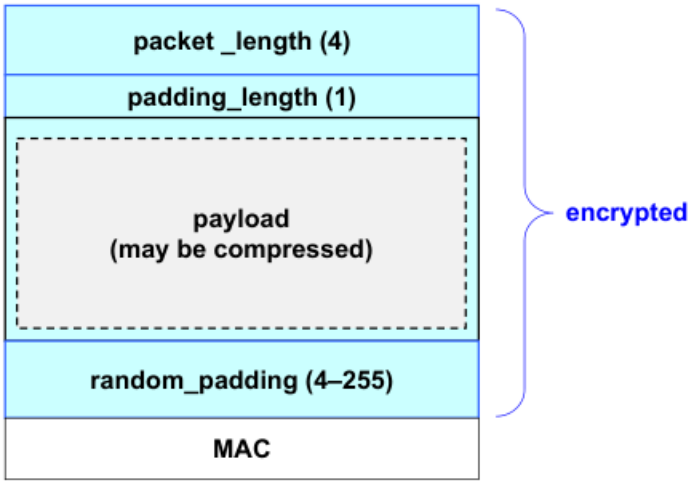


Figure 2.2: SSH Binary Packet Protocol

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

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

Payload

- The payload size is calculated as:

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

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

Random Padding

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

MAC (Message Authentication Code)

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

TLP key exchange

algorithm negotiation:

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

Algorithm Specification

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

SSH: Diffie-Hellman (DH) Key Agreement

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

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

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

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

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

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

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

  The server then computes the exchange hash:

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

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

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

  The client then computes the exchange hash:

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

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

2.2.2 SSH: Key Derivation

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

Initial IV (Initialization Vector)

- **Client to Server:**

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

- **Server to Client:**

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

Encryption Key

- **Client to Server:**

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

- **Server to Client:**

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

Integrity Key

- **Client to Server:**

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

- **Server to Client:**

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

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

2.3 Encryption

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

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

  [itemsep=0pt]
- **Recommended:**
  - aes128-cbc
- **Optional:**
  - blowfish-cbc
  - twofish256-cbc
  - twofish192-cbc
  - twofish128-cbc
  - aes256-cbc
  - aes192-cbc
  - serpent256-cbc
  - serpent192-cbc
  - serpent128-cbc
  - arcfour
  - idea-cbc
  - cast128-cbc
  - none

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

2.3.1 Encryption: Handling of IV

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

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

2.4 MAC

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

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

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

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

2.5 SSH: Peer Authentication

2.5.1 Server

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

2.5.2 Client

Can be done in two ways:

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

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

Chapter 3

The X.509 standard and PKI - Appunti AI CORREGGERE

3.1 Introduction to PKI and X.509 Certificates

3.1.1 Public Key Certificates (PKC)

A Public Key Certificate (PKC) is a data structure that securely binds a public key to some attributes. The term "securely bind" typically refers to the use of a digital signature by a Certification Authority (CA), although other techniques such as blockchain or direct trust are possible. The attributes bound to the public key are the ones relevant for the transaction being protected by the certificate. PKCs are crucial in achieving non-repudiation in digital signatures. Each PKC is the public complement of the corresponding private key.

3.1.2 Key Generation

Generating an asymmetric key-pair involves complex algorithms and usually requires a reliable Random Number Generator (RNG). Once generated, protecting the private key is crucial, both during storage and usage. Key generation and storage can be performed in software (e.g., a web browser), which introduces concerns about platform security (e.g., malware, weak implementation), or in dedicated hardware such as smart cards, which offer better protection but may be difficult to update.

3.2 Certification Architecture

3.2.1 Entities in PKI

A PKI involves the following key entities:

- **Certification Authority (CA)**: Generates and revokes certificates, publishes them along with status information.
- **Registration Authority (RA)**: Verifies the identity of the certificate requestor and authorizes the certificate issuance or revocation.
- **Validation Authority (VA)**: Provides services to verify the validity of a certificate, such as through a Certificate Revocation List (CRL) or Online Certificate Status Protocol (OCSP).

3.2.2 Certificate Generation Process

The basic process for certificate generation includes the following steps:

1. The user or RA generates a key-pair.
2. The RA verifies the user's identity.

3. The CA issues a certificate, binding the public key to the verified identity.
4. The certificate is published in a repository (along with potential revocation lists).

3.3 X.509 Certificate Standard

X.509 is a widely adopted standard for public key certificates. It was first defined in 1988 and has undergone multiple revisions:

- **Version 1 (1988)**: Basic certificate structure.
- **Version 2 (1993)**: Introduced minor improvements.
- **Version 3 (1996)**: Introduced extensions and attribute certificates.

3.3.1 Certificate Structure

An X.509 certificate is defined in ASN.1 (Abstract Syntax Notation 1). The basic structure includes:

```
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,
    extensions [3] Extensions OPTIONAL
}
```

3.3.2 Extensions

Certificates in X.509v3 support a variety of extensions, some of which are critical and others non-critical. Critical extensions must be recognized and processed by the entity verifying the certificate. If not, the certificate must be rejected.

Key and Policy Information

This class of extensions includes:

- **Authority Key Identifier (AKI)**: Identifies the public key used to sign the certificate.
- **Subject Key Identifier (SKI)**: Identifies the public key used by the subject of the certificate.
- **Key Usage**: Defines the cryptographic operations for which the public key can be used (e.g., digital signature, key encipherment).
- **Certificate Policies**: Lists policies under which the certificate was issued.
- **Policy Mappings**: Maps policies between different certification domains.

Certificate Path Constraints

Extensions that constrain the certificate path include:

- **Basic Constraints (BC)**: Indicates whether the certificate subject can act as a CA.
- **Name Constraints (NC)**: Limits the set of names that a CA can certify.
- **Policy Constraints**: Restricts the application of policies in the certification path.

Subject and Issuer Attributes

- **Subject Alternative Name (SAN)**: Allows for alternative methods to identify the subject (e.g., email, IP address).
- **Issuer Alternative Name (IAN)**: Alternative ways to identify the certificate issuer.

3.4 Certificate Revocation

A certificate can be revoked before its natural expiration. Reasons for revocation include key compromise, misuse, or incorrect issuance. The revocation status of a certificate must be checked by the relying party before accepting it.

3.4.1 Revocation Mechanisms

Two main mechanisms exist for checking the revocation status of a certificate:

- **Certificate Revocation List (CRL)**: A list of revoked certificates published by the CA. The CRL is periodically updated.
- **Online Certificate Status Protocol (OCSP)**: Provides real-time information on the status of a specific certificate.

3.5 Certificate Transparency and Auditing

3.5.1 Certificate Transparency (CT)

Certificate Transparency is an open auditing system that logs every issued certificate to ensure that no fraudulent certificates are issued without detection. It provides an infrastructure for domain owners and CAs to monitor certificate issuance.

3.5.2 Key Components of CT

- **Submitters**: Entities (usually CAs) that submit certificates to CT logs.
- **Loggers**: Servers that maintain the public log of certificates.
- **Monitors**: Entities that inspect the log for suspicious activity.
- **Auditors**: Verifiers of log integrity, ensuring the logs are consistent and certificates are properly logged.

3.6 Online Certificate Status Protocol (OCSP)

OCSP is a protocol used to query the current status of a certificate. Unlike CRLs, which provide a list of revoked certificates, OCSP queries the status of an individual certificate and returns a response indicating whether the certificate is "good", "revoked", or "unknown".

3.6.1 OCSP Response Structure

Each response is digitally signed to prevent tampering. A typical OCSP response contains:

- **Certificate ID**: Identifies the certificate being queried.
- **Certificate Status**: The current status (good, revoked, unknown).
- **This Update**: The time of the response.
- **Next Update**: The time at which the status is expected to change.

3.7 Cryptographic Hardware

3.7.1 Cryptographic Smart Cards

Smart cards are widely used to store private keys securely. They can also perform cryptographic operations, such as signing or encryption, directly on the card. Modern smart cards support both symmetric (e.g., AES) and asymmetric (e.g., RSA, ECDSA) algorithms. However, due to limited memory, the key size and operational capabilities of smart cards are constrained.

3.7.2 Hardware Security Modules (HSMs)

HSMs are hardware devices used to manage digital keys and perform cryptographic operations in a secure environment. HSMs are typically used by large organizations to protect high-value keys, and they support operations such as encryption, decryption, and digital signatures. HSMs come in various form factors, such as PCI cards, external devices, or network-attached appliances.

3.8 Automated Certificate Management Environment (ACME)

The ACME protocol automates the process of obtaining and managing public key certificates. Developed by the Internet Security Research Group (ISRG) for the Let's Encrypt service, ACME allows web servers to request, renew, and revoke certificates without human intervention.

3.8.1 ACME Process

1. The client (typically a web server) registers with a CA (e.g., Let's Encrypt) and proves control over the domain by responding to challenges.
2. The CA verifies domain control and issues a certificate.
3. The client can automatically renew or revoke certificates as needed.

Chapter 4

Integrity and attestation of distributed infrastructures (cloud, SDN, NFV, ...) Appunti AI CORREGGERE

4.1 Introduction

4.1.1 Typical distributed infrastructures

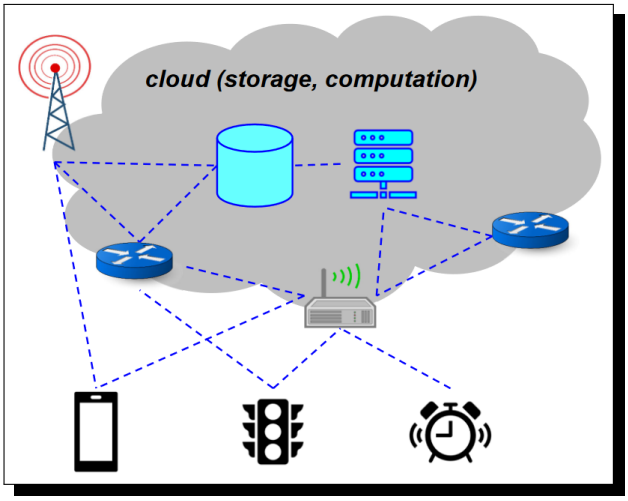


Figure 4.1: Distributed infrastructure

1. Cloud computing (Server, Storage ...)
2. edge devices (Router, Switch,modems ...) for guarantee access the cloud
3. IoT and personal devices (Smartphone, robot cleaner, ...) for use the data and services

All these conncection are different (wired, wireless, virtulized, ...) and the secure all these kind of connection is very hard.

4.1.2 Trend towards softwarization

So now a lot of hardware device are repliaced by software, this is a trend that is called **softwarization**.

- **SDN (Software-Defined Networking)**
- **... but also:**

  - SDR, NFV, ...
  - AI (!)

- **As a consequence, more flexible but more vulnerable:**
  - Software more prone to bugs than hardware
  - Software updates (if you receive an update, is it trusted? it include a malware?)

> - EASY (FAST, FLEXIBLE, ...)
> - CHEAP
> - SECURE
> ... PICK TWO!

Can I trust this infrastructure?

A trustworthy infrastructure is defined as one that will behave in the expected way. However, several **problems** arise in achieving this trust:

- Trust in the cloud provider(s) (as HW, as SW, as people managing the infrastructure)
- Trust in the network/edge provider(s) (as cloud)
- Low or no access control for edge- and end-devices
- Low-cost IoT devices (typically implying low security, so most of them are completely unsecure)
- Personal devices (typically managed by "ignorant" users)

If possible, the infrastructure should be protected to avoid or block attacks. Otherwise, it is essential to **at least monitor the "state"** of the system for early detection and possible reaction.
Some of this work is done with IDS, but can't find all the possible attacks, so we need to do some **Integrity Verification**

4.1.3 Integrity

Hardware:

- Am I talking to the right (intended) node?
- Does it host the expected (physical) components?

Software:

- Am I talking to the right (intended) software component?
- Is it correctly configured?
- Is the baseline software the expected one?

4.2 TEE - Trusted Execution Environment

A small part of the system that is isolated from the rest of the system, and that is trusted to execute some code in a secure way.

4.2.1 Trusted vs. Trustworthy

- **Trusted** refers to someone or something that you rely upon to not compromise your security. On the other hand, **Trustworthy** refers to someone or something that will not compromise your security.
- *Trusted* is about how you use something, while *Trustworthy* is about whether it is inherently safe to use.
- A **Trusted Execution Environment** is what you may choose to rely upon to execute sensitive tasks, such as running **Trusted Applications (TA)**. This environment should be **Trustworthy** as well!

4.2.2 TEE evolution

The evolution of the **Trusted Execution Environment (TEE)** has gone through several significant milestones.

Open Mobile Terminal Platform (now part of GSMA) started the initiative.

- **2006**: Introduction of the **TR0 specifications**, outlining the basic security requirements.
- **2008**: Release of the **TR1 specifications**, which added the TEE layer on top of TR0.
- **2010**: **GlobalPlatform** was launched to define interfaces and provide certifications, becoming the de-facto standardization body for TEEs.
- **2012**:
  - **ARM**, **Gemalto**, and **G+D** formed **Trustonic**, aiming to create an open TEE.
  - **GlobalPlatform** and the **Trusted Computer Group (TCG)** created a joint working group to focus on TEE specifications and its use combined with the **Trusted Platform Module (TPM)**.

The first major business case for the Trusted Execution Environment (TEE) was Netflix's high-resolution content on smartphones and tablets. Following this, TEE's application rapidly expanded into various sectors, including financial services, enterprise solutions, government security, automotive industries, and the Internet of Things (IoT)

4.2.3 TEE and REE

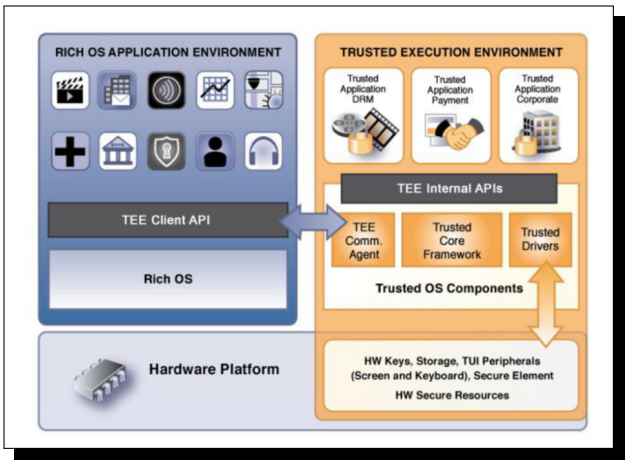


Figure 4.2: TEE and REE

(recuperare descrizione da transcript)

- *Note:*
- TUI: Trusted User Interface
- TEE run only on device that respect RT0 specifications
- The most risky part is the communication between TEE client API and TEE internal APIs

4.2.4 TEE - Some key points

Trusted Execution Environment (TEE) is currently a hot topic in the realm of *confidential computing*, which is promoted by the **Confidential Computing Consortium (CCC)**.
One of the key objectives of TEE is the protection of **data in use**, ensuring that no unauthorized entity can read or write this data, and it is processed only by an authorized application. This is in contrast to various cryptographic techniques used to protect *data at rest* and *data in motion*.

A critical component of TEE is the **Root of Trust (RoT)**, an element whose misbehavior cannot be detected during runtime (it's trusted and stop). It is essential that the RoT be both *trusted* and *trustworthy*. The RoT is part of the **Trusted Computing Base (TCB)**, which includes all hardware, firmware, and software components that are crucial to the system's security. Any vulnerability within the TCB could potentially jeopardize the security of the entire system.

4.2.5 TEE Security Principles

Trusted Execution Environments (TEEs) are built on foundational security principles to ensure the confidentiality and integrity of sensitive tasks. These principles include:

- Being part of the device's secure boot chain, which is based on a Root of Trust (RoT). This ensures code integrity is verified during each device boot.
- Hardware-based isolation from the rich operating system (OS) environment, allowing sensitive code execution in a protected manner.
- Isolation of Trusted Applications (TAs) from each other to prevent interference or data leakage.
- Secure data storage, leveraging a hardware-unique key accessible only by the TEE OS. This prevents unauthorized access, modification, or exploitation of data across devices.
- Privileged and secure access to peripherals (also known as a trusted path). For instance, peripherals like fingerprint sensors, displays, and touchpads can be isolated from the rich OS environment and controlled solely by the TEE during specific operations. This setup ensures no visibility or access from the Rich Execution Environment (REE), including from potential malware.

4.2.6 Intel IPT

Intel Identity Protection Technology (IPT) is a security feature that runs a Java applet on a separate CPU. The **Management Engine** is part of the chipset, meaning it is bound to the physical hardware. This setup enables a variety of secure applications, including:

- Key generation and storage (integrated with Windows Cryptographic API)
- One-Time Password (OTP) generation (e.g., VASCO MYDIGIPASS.COM)
- Secure PIN entry
- – These features are made possible because the chipset also manages video

4.2.7 ARM TrustZone

ARM TrustZone extends CPU buses to a "33rd bit," which signals whether the CPU is in secure mode or not. This signal is exposed outside of the CPU, enabling the use of secure peripherals and secure RAM.

Key characteristics include:

- The system is open and well-documented.
- **But**, it only allows for *one* secure enclave.
- ARM is working on adding a *third mode* to enhance its capabilities.

Additionally, there is potential for an indicator to signal the current mode the CPU is operating in.

4.2.8 Trustonic

TrustZone by itself is limited due to only allowing one secure enclave, but several companies have developed solutions to overcome this limitation:

- **Gemalto** developed the *Trusted Foundations system*.
- **G+D (Giesecke+Devrient)** developed *MobiCore*.
- Both of these solutions essentially split the one secure enclave into several ones, using a smart-card-like operating system.

Trustonic's development is based on MobiCore, and license fees are required to implement its code.

Trustonic's TEE OS, **Kinibi**, powers various systems, such as version 500, which supports 64-bit SMP for embedded systems.

Samsung Knox provides a similar functionality, but also introduces secure boot for added security.

4.2.9 Intel SGX

Intel Software Guard Extensions (SGX) is a security feature tightly integrated with the CPU, designed to modify memory management and enhance protection for enclaves. These enclaves are isolated from other code, ensuring that their contents are secure and cannot be accessed by unauthorized code.

Key features include:

- As enclaves are built, the code is measured in a manner similar to TPM, ensuring integrity.
- SGX can be combined with Intel Identity Protection Technology (IPT) for trusted display functionality.
- Initially, **SGX1** was available on both low- and high-end CPUs, but now **SGX2** is mostly restricted to server-oriented CPUs, such as Xeon processors.

4.2.10 Keystone

Keystone is an open-source framework designed to build Trusted Execution Environments (TEEs) with flexibility and modularity. The key features include:

- An open-source framework that allows developers to select only the required features, reducing the Trusted Computing Base (TCB) to a minimum.
- It operates in an untrusted environment, typically a general-purpose OS, while supporting multiple trusted and segregated enclaves.
- Built on top of the RISC-V open-source customizable hardware architecture.
- Can be implemented on FPGA or SoC platforms, with a core that includes cryptographic extensions.
- Supports various execution modes: Machine (M), Supervisor (S), and User (U).
- Physical Memory Protection (PMP) is utilized for both memory and I/O protection, ensuring secure execution and isolation (So create a secure page that can give to a process and this page is not accessible from the other process).

Motivation for Keystone

The motivation behind Keystone arises from the limitations of existing Trusted Execution Environments (TEEs), which are often rigid and lack customizability. Several existing solutions inherit significant design constraints, such as:

- **Intel SGX**: Characterized by a large software stack, which introduces complexity and a lot of libaries (so a big trust in Intel).
- **AMD SEV**: Involves a large Trusted Computing Base (TCB), need a large use of software, so a large trusted domanin
- **ARM TrustZone**: Restricts flexibility due to the limited number of secure domains it can handle.

Keystone aims to address these issues by providing a more customizable, modular TEE framework.

Keystone architecture

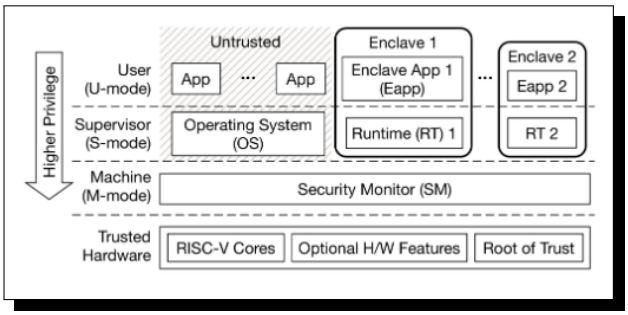


Figure 4.3: Keystone architecture

- risc-v core: normal operations
- Root of trust: needed
- Optional H/W features: can be added
- Security Monitor: The only program that run in Machine mode, and mediates all the operations that are sent to the HW.
- At User level, we can have all the untrusted applications.
- between U and S mode, we can create all the enclaves that we want (Enclave app (Eapp) at U-mode and Enclave runtime (RT) at S-mode)

Keystone: Further Reading

Documenti da leggere per l'esame

For those interested in exploring Keystone in more detail, here are some useful resources:

- **Keystone presentation at CCC**:
  - Video: youtube
  - Slides: *CCCWebinar-Keystone.pdf* (available in course material)

- **Keystone presentation at EuroSys'20**:
  - Slides: *233_lee_slides.pdf* (available in course material)
  - Video: youtube
  - Paper: n.ethz.ch (also available in course material)

4.3 Trusted computing and remote attestation

4.3.1 Baseline Computer System Protection

Attackers often aim to inject malware at the lowest possible level to stay undetected and gain control over large portions of a system. Common tactics include:

- Modifying the operating system (OS)
- Attempting to boot an alternative OS
- Altering the boot sequence or boot loader (for start an untrusted part)

To counter these threats, it is essential to protect the boot system and the OS itself. In earlier systems, the BIOS was used, but it was very difficult to secure (no support to signature). Now, with the advent of UEFI, systems have:

- Native support for firmware signature and verification
- The ability for the boot loader to verify the OS before activation

4.3.2 Rootkits

Rootkits are malicious software designed to gain privileged access by infiltrating different layers of a computer system. Common types include:

- **Firmware rootkits:** These overwrite the BIOS/UEFI or the firmware of other hardware components (like a network card for a server, that have another processor and firmware, so it can be object of an attack), allowing the rootkit to start before the operating system (OS) loads.
- **Bootkits:** These replace the OS's bootloader, enabling the system to load the bootkit before the OS, compromising the system early in the boot process.
- **Kernel rootkits:** These modify portions of the OS kernel, ensuring the rootkit loads automatically when the OS starts, giving the attacker control at the core level.
- **Driver rootkits:** These impersonate trusted hardware drivers, such as those used by operating systems like Windows, to communicate with hardware, allowing the rootkit to operate undetected.

> **Software to Protect Software?**
>
> Relying solely on software to protect other software is generally *not* a good idea, as software can fail. To ensure robust protection, hardware support is necessary for safeguarding software. A key element in this context is the **Root of Trust (RoT)**, which should be part of the **Trusted Computing Base (TCB)**. This is essential because hardware itself is operated by software or firmware. Therefore, the TCB should be kept minimal to reduce potential vulnerabilities.

4.3.3 Firmware Self-Protection (SW Root-of-Trust)

Example courtesy of HPE.

1. HPE has a *signature* region at a fixed location in the final BIOS image (16MB).
2. After the BIOS build (i.e., at manufacturing time), the SHA256 of specific BIOS regions is calculated; these regions include static code, the BIOS version information, and microcode.
3. The hash is sent to the HPE signing server, which returns a signed hash image (32 bytes + signature and certificate size), which is copied into the *signature* region.
4. After power-on, the early BIOS code calculates the combined hash of each of the specific valid regions in the BIOS image.
5. After verifying that the *signature* region contents are valid, the BIOS compares the stored hash and the calculated hash.
6. If both are the same, the boot continues; otherwise, the system halts.

HW root-of-trust for firmware protection

For hardware root-of-trust, self-verification is based on the firmware itself, where the static portion verifies the updatable part. However, an external chip can also be used to verify the firmware, as shown in the system diagram below. The external crypto chip validates the BIOS in the SPI flash after power-on. Only once validation is successful does the x86 CPU exit reset mode; otherwise, the system remains in a reset state.

This chip also includes a fusing option, enabling one public key hash to be fused, which will be used to verify the signature of the hash file stored in the "signature" region. The validation flow is similar to the BIOS self-integrity check, but with the external chip performing the validation, it establishes the real hardware root of trust.

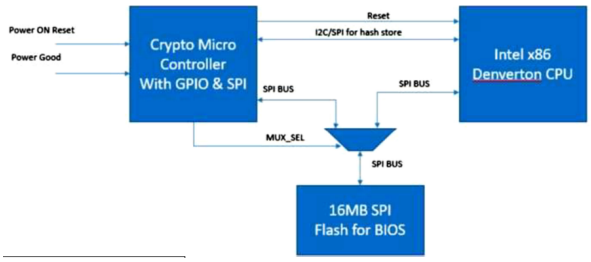


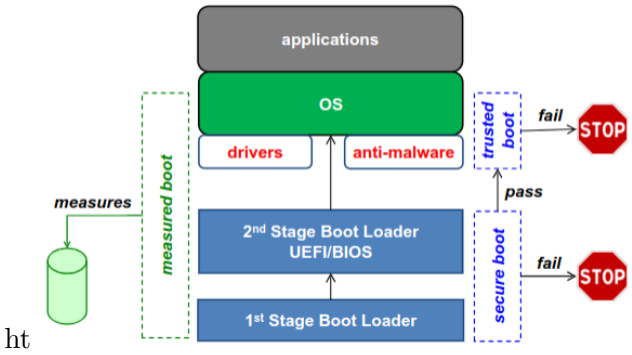
Figure 4.4: Firmware Root of Trust - System Diagram

4.3.4 Boot Types

There are several types of boot processes with varying degrees of security.

- The simplest type is the **plain boot**, which offers no security mechanisms at all, making the system vulnerable to attacks.
- **Secure boot** ensures that the firmware verifies the signature of critical components and will halt the platform if the verification fails. This type of boot is mostly hardware-based, verifying components up to the OS loader.
- **Trusted boot** goes further by having the OS verify the signature of its components, such as drivers and antimalware. If the verification fails, the OS will stop operations. Trusted boot is primarily software-based and ensures verification up to the OS operational state.
- **Measured boot** records measurements of all components executed from the beginning of the boot process up to a defined point (X). Unlike secure and trusted boot, it does not stop operations when an issue is detected but can securely report these measures to an external verifier for further analysis.

Note: trusted driver and antimalware start before the OS



ht

Figure 4.5: Hardware Root of Trust - System Diagram

Note: ELAM: Early Launch Anti-Malware (signed by Microsoft)

4.3.5 Trusted Computing

Trusted Computing refers to schemes that establish trust in a platform by identifying its hardware and software components. A trusted component or platform is one that behaves as expected, though it is important to note that trust is not synonymous with being good or secure. For trust to be meaningful, the component's behavior must be verified against its expected behavior.

One key element in Trusted Computing is *attestation*, which provides verifiable evidence of the platform's state. The *Root of Trust* (RoT) represents an inherently trusted component in this context, forming the foundation upon which trust is established.

A central part of Trusted Computing is the *Trusted Platform Module* (TPM). The TPM is responsible for collecting and reporting the identities of a platform's hardware and software. In a typical system, the TPM reports on these components in a way that allows the system to determine whether their behavior aligns with expectations, thereby establishing a trusted environment.

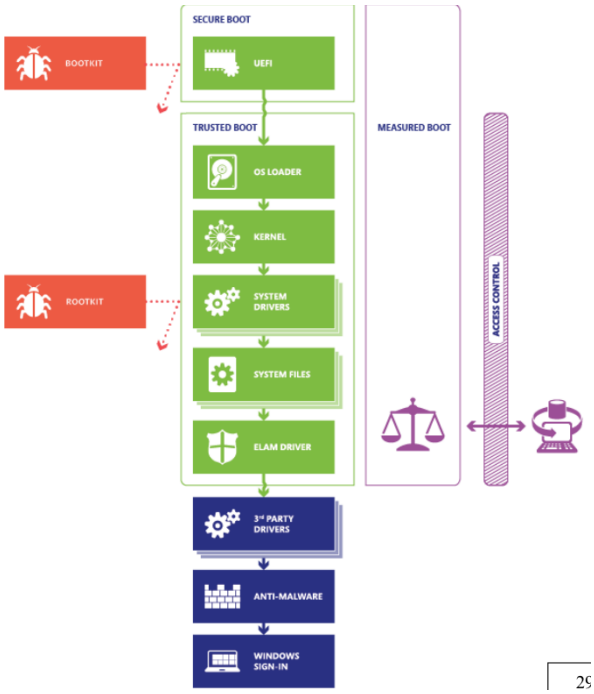


Figure 4.6: Windows boot protection

4.3.6 Trusted Computing Base (TCB)

The *Trusted Computing Base (TCB)* is the collection of system resources, including both hardware and software, responsible for maintaining the system's security policy. One critical attribute of the TCB is its ability to protect itself from being compromised by any hardware or software that is not part of the TCB.

It is important to note that the *Trusted Platform Module (TPM)* is not the TCB of a system. Instead, the TPM serves as a component that enables an independent entity to determine if the TCB has been compromised. In certain situations, the TPM can assist in preventing the system from starting if the TCB cannot be properly instantiated.

4.3.7 Root of Trust (RoT)

The *Root of Trust (RoT)* refers to a component that must always behave as expected, as its misbehavior cannot be detected. It forms the fundamental building blocks for establishing trust in a platform. There are three main types of RoT:

Root of Trust for Measurement (RTM): Responsible for measuring system integrity and sending integrity measurements to the **RTS**. Typically, the CPU runs the *Core Root of Trust for Measurement (CRTM)* at boot, as the first piece of BIOS/UEFI code, initiating the chain of trust (because we want that all that is executed is mesure).

Root of Trust for Storage (RTS): Provides shielded and secure storage for sensitive information

Root of Trust for Reporting (RTR): Securely reports the content of the RTS, ensuring reliable communication of the platform's integrity status.

4.3.8 Chain of Trust

The *Chain of Trust* is a sequential process where each component measures the integrity of the next component in the system. The process proceeds as follows:

Component A (typically the *Core Root of Trust for Measurement*, CRTM) measures the integrity of **Component B** and stores the measurement in the *Root of Trust for Storage (RTS)*. Next, **Component B** measures **Component C** and similarly stores the measurement in the RTS. This process continues for each subsequent component.

An important point to note is that a verifier, by using the *Root of Trust for Reporting (RTR)*, can securely retrieve the measurements of components like B and C from the RTS. However, components B and C can only be trusted if **Component A** (the CRTM) is trustworthy, ensuring the integrity of the entire system chain.

4.4 Trusted Platform Module (TPM) overview

The Trusted Platform Module (TPM) is an **inexpensive component**, typically costing less than $1, and is available on most servers, laptops, and PCs. It is designed to be **tamper-resistant**, although it is important to note that it is not tamper-proof.

While the TPM is **not a high-speed cryptographic engine** and is generally considered rather slow, it is certified with a **Common Criteria EAL4+ rating**. The TPM acts as a **passive component that requires a CPU** to function effectively.

Although the TPM cannot prevent the boot process, it can protect data and securely report it. In this context, the TPM serves as both an RTS (Root of Trust for Storage) and an RTR (Root of Trust for Reporting), but it is not classified as an RTM (Root of Trust for Measurement).

4.4.1 TPM Features

The Trusted Platform Module (TPM) provides several important features:

- **RTS**: Secure storage (extend-only).
- **RTR**: Reports the content of the RTS with a digital signature.
- Hardware random number generator.
- Supports various cryptographic algorithms, including hash, MAC, symmetric, and asymmetric encryption. However, it is **NOT a crypto accelerator** and is considered slow.
- Secure generation of cryptographic keys for limited uses.
- **Binding**: Data encrypted using the TPM bind key, a unique RSA key descending from a storage key.
- **Sealing**: Similar to binding, but specifies the TPM state required for the data to be decrypted (i.e., unsealed).
- Computer programs can use a TPM to authenticate hardware devices, as each TPM chip has a unique and secret **Endorsement Key** (EK) burned in during production.

4.4.2 TPM-1.2

The Trusted Platform Module (TPM) version 1.2 includes a fixed set of algorithms, specifically SHA-1, RSA, and optionally AES. It features one storage hierarchy for the platform user and provides a single root key known as the **Storage Root Key (SRK)**, which uses RSA-2048 encryption. The TPM establishes hardware identity through the built-in **Endorsement Key (EK)** and supports sealing to **Platform Configuration Register (PCR)** values.

4.4.3 TPM-2.0

The Trusted Platform Module (TPM) version 2.0 offers enhanced features, including cryptographic agility with support for algorithms such as SHA-1, SHA-256, RSA, ECC-256, HMAC, and AES-128. It introduces three key hierarchies: **platform**, **storage**, and **endorsement**, allowing for multiple keys and algorithms within each hierarchy. Additionally, TPM 2.0 implements policy-based authorization, enabling more flexible security management. Furthermore, it provides platform-specific specifications for various environments, including:

- **PC client**,
- **mobile**
- **automotive-thin**

Implementation of TPM-2.0

The Trusted Platform Module (TPM) 2.0 can be implemented in various ways:

- **Discrete TPM**: A dedicated chip that implements TPM functionality in its own tamper-resistant semiconductor package.
- **Integrated TPM**: A part of another chip that is not required to implement tamper resistance (for example, Intel has integrated TPMs in some of its chipsets).
- **Firmware TPM**: A software-only solution that runs in a CPU's trusted execution environment. AMD, Intel, and Qualcomm have implemented firmware TPMs.
- **Hypervisor TPM**: A virtual TPM provided by a hypervisor, which runs in an isolated execution environment and is comparable to a firmware TPM.
- **Software TPM**: A software emulator of TPM that is useful for development purposes.

TPM-2.0 three hierarchies

The Trusted Platform Module (TPM) 2.0 defines three key hierarchies.

The **Platform Hierarchy** is dedicated to the platform's firmware and provides non-volatile (NV) storage for keys and data.
The **Endorsement Hierarchy** is used for the privacy administrator and contains keys and data associated with the endorsement process.
The **Storage Hierarchy** is for the platform's owner, who is usually also the privacy administrator, and offers NV storage for keys and data.

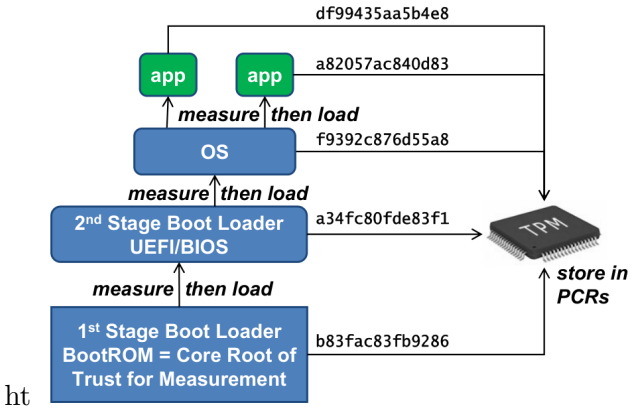
Each hierarchy has dedicated authorization (password) and policy, along with a specific seed for generating the primary keys.

4.4.4 Using a TPM for securely storing data

Using a Trusted Platform Module (TPM) for securely storing data provides several benefits. The TPM offers **physical isolation**, allowing data to be stored securely within the TPM itself, specifically in the non-volatile random access memory (NVRAM). It supports both **primary keys** and **permanent keys**, although it has very limited storage capacity.

Furthermore, the TPM enforces **Mandatory Access Control** to ensure that only authorized users can access sensitive data. In addition to storing data within the TPM, it can also facilitate **cryptographic isolation** for storage outside of the TPM, such as on the platform's HDD or SSD.

When storing keys or data outside the TPM, it is crucial that any blobs of data are protected. This protection is achieved by encrypting the data with a key controlled by the TPM, ensuring that unauthorized access is prevented while maintaining **Mandatory Access Control**.



ht

Figure 4.7: Measured Boot

4.4.5 TPM objects

In the context of Trusted Platform Modules (TPM), objects are primarily categorized into primary keys and other associated elements. The **primary keys** include endorsement keys and storage keys, which are derived from one of the primary seeds. Notably, the TPM does not return the private value of these keys; however, they can be recreated using the same parameters, assuming the primary seed has not been altered.

Additionally, TPM objects include **keys and sealed data objects (SDOs)**, which are protected by a **Storage Parent Key (SPK)**. The SPK is essential within the TPM for loading or creating a key/SDO. The randomness required for key generation comes from the TPM's Random Number Generator (RNG). When generating keys, the TPM returns the private part, which is also protected by the SPK. It is important to note that this private part needs to be stored securely somewhere.

4.4.6 TPM object's area

- **public area**: used to uniquely identify an object
- **private area**: object's secrets, only exists in the TPM
- **sensitive area**: encrypted private area, used for storage outside of the TPM

4.4.7 TPM Platform Configuration Register (PCR)

The TPM Platform Configuration Register (PCR) is the TPM's implementation of **Runtime State (RTS)**, serving as a core mechanism for recording the integrity of the platform. PCRs are only reset during a platform reset or when triggered by a hardware signal, ensuring that any malicious code cannot alter their measurement retrospectively.

PCRs are extended using a cumulative hash function, which can be expressed as follows:

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

This operation is commonly referred to as the **EXTEND** operation. Furthermore, PCR values can be utilized to gate access to other TPM objects, such as when BitLocker seals disk encryption keys to specific PCR values, enhancing the security of the data stored on the platform.

4.4.8 Remote attestation procedure

1. challenge (=nonce)
2. measurements (and nonce) signed with the device's key
3. validate signature (crypto + ID) and check measurements against Reference Measurements (golden values)

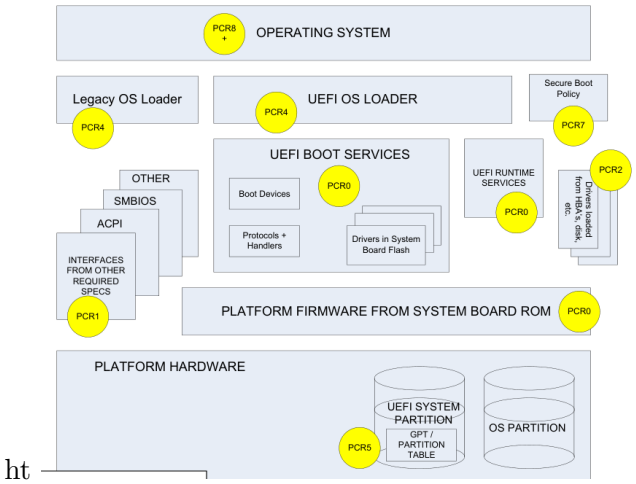


Figure 4.8: architecture of TCG PC Client

4.4.9 Management of Remote Attestation

Management of Remote Attestation involves several considerations. One of the first questions is whether to focus on **boot attestation** (static) or also include periodic (dynamic) attestation. It is essential to consider the attack model, particularly regarding runtime vulnerabilities, as well as the periodicity of the operation. The speed of an attack must also be taken into account, especially given the implementation limits, which include factors such as signature verification, protocol handling, and database lookups. Currently, the response time for these operations is in the range of several seconds, largely due to the inherent slowness of the TPM.

Another critical aspect is **whitelist generation**. This task can be challenging in general but becomes more manageable within limited environments, such as IoT, edge devices, Software-Defined Networking (SDN), and Network Function Virtualization (NFV). Labels for attestation states, such as "good," "old," "buggy," and "vulnerable," play a significant role in this process. Moreover, it is crucial to include configurations, which may come from management and orchestration frameworks (like MANO) or network management systems. Whitelist generation is typically easier when dealing with file-based configurations, but it becomes significantly more difficult when configurations are stored in memory.

4.4.10 TCG PC Client PCR use

Table 4.1: PCR Index and Usage

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

4.4.11 Measured execution

In the context of measured execution, a notable **problem** arises: there are often not enough Platform Configuration Registers (PCRs) available to capture the necessary integrity measurements. A **solution** is to utilize just one PCR. However, this approach introduces a significant caveat: the PCR value now becomes dependent on the execution order of the measurements.

4.4.12 Linux's IMA

Linux's Integrity Measurement Architecture (IMA) extends attestation capabilities to dynamic execution, such as applications. The process can be broken down into several key functions:

- **Collect**: IMA measures a file before it is accessed, ensuring that any changes to the file's integrity are detected early in the execution process.
- **Store**: The measurement is then added to a kernel-resident list known as the Measurement List (ML) and is used to extend the IMA Platform Configuration Register (PCR), specifically PCR 10.
- **Appraise**: IMA can enforce local validation of a measurement against a "good" value, which is stored in an extended attribute of the file. This step ensures that only trusted files are executed in the system.
- **Protect**: Finally, IMA protects a file's security extended attributes, including the appraisal hash, against offline attacks, thereby maintaining the integrity of the file's security features.

Details

Linux's Integrity Measurement Architecture (IMA) extends the concept of UEFI measured boot to the operating system and applications. It specifically uses PCR 10 for integrity measurements. The first measurement recorded is **boot_aggregate**, which is a hash of the TPM's PCRs 0 to 7 (i.e., UEFI-related PCRs).

Measurement configuration is facilitated through the IMA template, primarily using the **ima-ng** format, though it can be customized to meet specific needs. These configurations are exposed in the kernel's `securityfs` at the path `/sys/kernel/security/ima/ascii_runtime_measurements`.

Table 4.2: IMA Measurement Records

PCR	template_hash	fitemplate	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

4.4.13 Verification of the IMA ML

With IMA enabled, the attestation report includes not only the nonce and PCR values but also the Measurement List (ML). However, the value of PCR 10 is variable; it depends on two factors: (A) the applications executed and (B) the order of execution of those applications. Consequently, the verifier must compute the correct value of PCR 10 by utilizing the ML.

The process can be outlined as follows:

- myPCR10 = 0
- myPCR10 = Extend (boot_aggregate)
- For each measure M of component C:
  - If (C not authorized) then alarm

- If (M different from gold_measure(C)), then raise alarm
- myPCR10 = extend (M)

- If myPCR10 equals PCR10, then OK, else alarm

In summary, this verification process ensures that the components executed on the system are authorized and that their measurements match the expected values, thereby maintaining the integrity of the system.

4.4.14 Size and variability of the TCB

The Trusted Computing Base (TCB) represents the smallest amount of code, hardware, personnel, and processes that must be trusted to fulfill security requirements. Confidence in the TCB can be enhanced through various means, including static verification, code inspection, testing, and formal methods. However, all these methods tend to be expensive, making it crucial to reduce the complexity of the TCB. Nevertheless, simply reducing complexity is not sufficient on its own.

The Trusted Platform Module (TPM) attempts to establish a TCB via the Core Root of Trust for Measurement (CRTM). Despite these efforts, the TCB has often become too large and excessively dynamic. Consequently, two "identical" computers may yield different measurements, which complicates the assurance of trustworthiness across systems.

4.4.15 Dynamic Root of Trust for Measurement

The Dynamic Root of Trust for Measurement (DRTM) offers an alternative approach by not trusting all operations that occur since the BIOS. Instead, it resets the CPU and begins measuring from that point onward. In this context, TPM version 1.2 introduced dynamic Platform Configuration Registers (PCRs), specifically PCRs 17 to 23, which are initially set to -1 upon boot. These dynamic PCRs can subsequently be reset by the operating system to 0 when necessary.

DRTM utilizes special processor commands, such as `SENTER` for Intel TXT, which executes the SINIT binary module, and `SKINIT` for AMD SVM. This mechanism halts all processing on the platform, allowing DRTM to hash the contents of a designated memory region and store the measurement in the dynamic PCR. Control is then transferred to a specified location in memory. This process is also referred to as Late Launch.

One of the significant advantages of DRTM is its ability to address the problem of incorrect PCR values when firmware is updated, which can lead to issues with sealed data. By establishing a new root of trust at a later stage, DRTM helps ensure the integrity of the platform even amidst changes to the firmware.

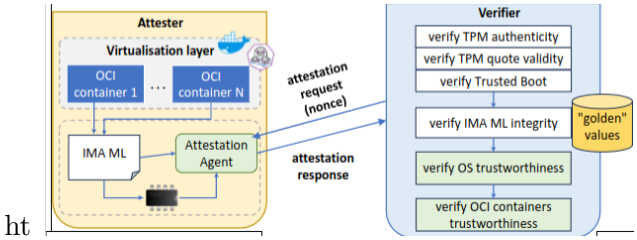
4.4.16 Hypervisor TEE

The Dynamic Root of Trust for Measurement (DRTM) was designed to facilitate the loading of a hypervisor, such as Xen or VMware ESX. Once the hypervisor is loaded, it can isolate virtual machines (VMs) effectively. The Trusted Platform Module (TPM) plays a crucial role in attesting the hypervisor, ensuring that it has been loaded properly before allowing access to TPM sealed storage. This capability may prove particularly beneficial in cloud computing environments, where security and isolation are paramount. However, it is important to note that the hypervisor itself comprises a substantial amount of code that must be validated. For instance, Xen includes a complete copy of Linux, while VMware is of a similar scale, which raises concerns regarding the potential attack surface.

4.4.17 RA in Virtualized Environments

Having a hardware Root of Trust (RoT) is a critical aspect of security in virtualized environments. In the case of full virtualization, such as with virtual machines (VMs), there is often only a software implementation of the RoT, exemplified by virtual Trusted Platform Modules (vTPMs) provided by platforms like Xen, Google, and VMware. To enhance security, it is essential to establish a strong link between the vTPM and the physical TPM (pTPM).

One promising approach to achieve this is through deep attestation, which leverages hardware-based security. This method involves using sealed objects that are rooted in the pTPM to protect the vTPM. However, implementing



ht

Figure 4.9: RA for OCI Containers

this strategy requires an extension of the standard interfaces defined by the Trusted Computing Group (TCG), and this is currently an ongoing area of development.

In contrast, when adopting lighter forms of virtualization, such as Docker containers, a different solution becomes feasible. Since hardware, including the TPM, is shared among containers, this allows for alternative security mechanisms that can still ensure the integrity and confidentiality of applications without the overhead associated with full virtualization.

4.4.18 RA for OCI Containers

Remote Attestation (RA) for Open Container Initiative (OCI) containers is designed to be agnostic, meaning it is not tied to a specific containerization technology. This approach ensures that RA is transparent to both the container runtime and the containerized workloads, allowing for seamless integration. The RA process provides assurance of the integrity of the host system along with the containers themselves, leveraging a hardware Root of Trust (RoT) to establish a secure and trusted environment.

Implementation

The implementation of Remote Attestation (RA) for Open Container Initiative (OCI) containers involves the introduction of a new Integrity Measurement Architecture (IMA) template known as `ima-dep-cgn`. This template is designed to assess the dependencies of entries, determining whether they belong to the host or to a specific container.

A key feature of this implementation is the inclusion of a `control-group-name`, which uniquely identifies the container in question. Furthermore, the `template-hash` is computed using a digest algorithm other than SHA-1, allowing for the use of more secure hashing methods such as SHA-256 or SHA-512. This enhances the overall integrity and security of the RA process for OCI containers.