ICS - Information Security Systems
Lecture 4: Attacks and PKI

2024-2025 ICS

Outline

Attack Models in Cryptography

Black Box Model

- The adversary only sees the system's **inputs and outputs**.
- No access to the internal implementation.
- Example: encryption or decryption oracle (CPA/CCA).

Gray Box Model

- The adversary has **partial access** to the inside of the system.
- Observes **physical leakages**: timing, power consumption, electromagnetic emissions.
- Example: side-channel attacks.

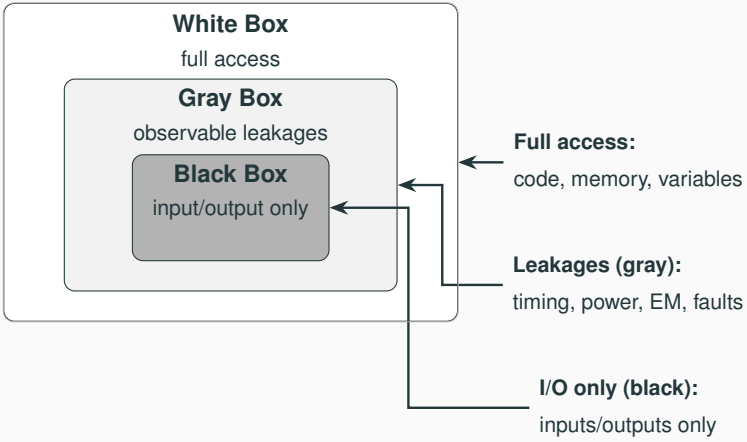
White Box Model and Comparison

White Box Model

- The adversary has **full access** to the code and execution.
- Can **observe and modify** memory and internal variables.
- Context: software in untrusted environments (DRM, mobile applications).
- Objective: keep the key **non-extractable** even with total access.

Model	Access	Examples	Adversary objective
Black Box	None	Oracle, CPA/CCA	Recover the key via I/O
Gray Box	Partial	Side-channel, faults	Exploit leakages
White Box	Total	Reverse engineering	Extract the key anyway

Models: Visual Comparison

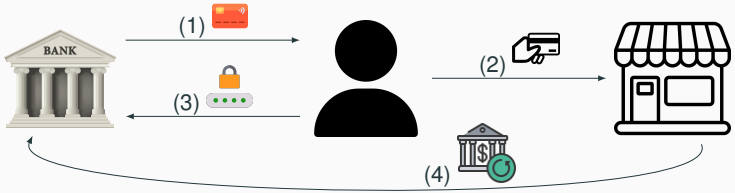


What security can we expect in the **gray**/**white** box model?

Outline

Card Payment

Procedure:



Card Issuance

Card Issuance

Payment

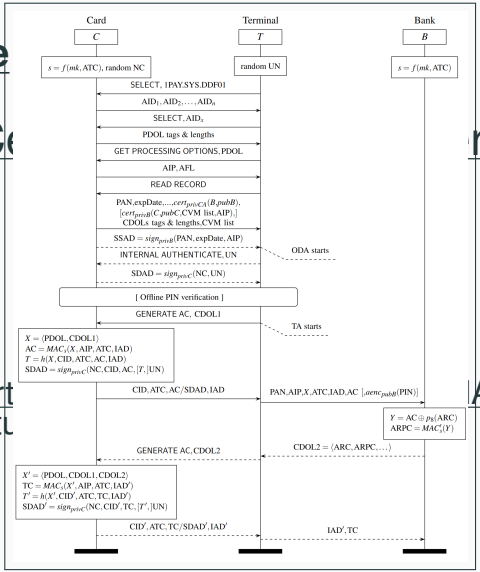
The EMV Payment Protocol

Card Issuance

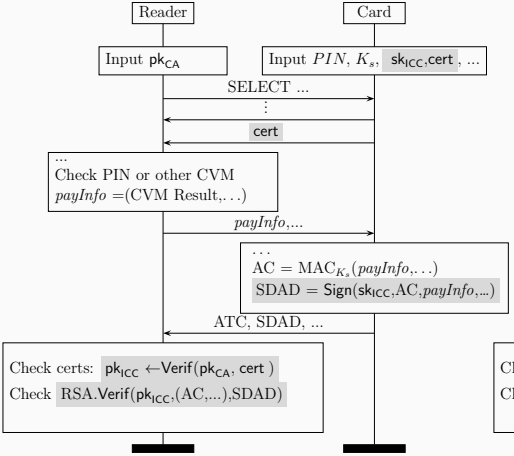
Ce … rtifies

Payment

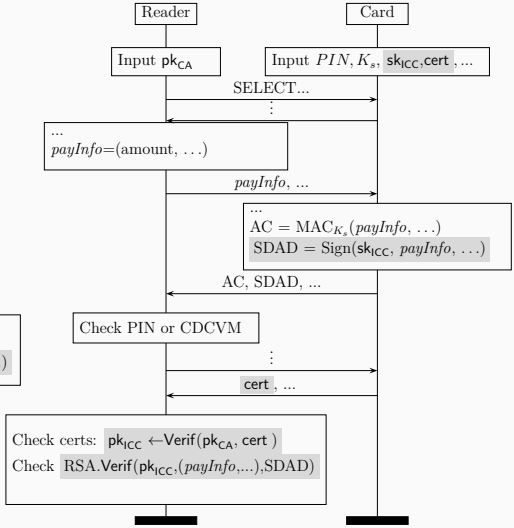
Cert… Signatu… … AC



EMV Payment Protocol (Simplified)



Mastercard protocol.



Visa protocol.

Cryptanalysis of RSA

Cryptanalysis

Goal of an attacker: recover the secret key.

Methods

- brute-force search $\Longrightarrow$ far too long
- break the algorithm $\Longrightarrow$ far too hard
- retrieve it directly from the user $\Longrightarrow$ often impossible

$\Longrightarrow$ Side-channel attacks.

Side-channel attack:

Attack that recovers the secret using physical information (execution time, power consumption, etc.).

Simple Power Analysis

RSA weakness: exponentiation

The simplest method to compute exponentiation is fast exponentiation. Its power consumption is directly linked to the exponent.

Simple Power Analysis (SPA)

This method recovers information by analysing the power consumption curve during exponentiation by d. The curve differs depending on the executed instructions and the processed data.

Fast Exponentiation Algorithm

Compute h^d

$d = [d_n d_{n-1} \dots d_0]_2$

$T \leftarrow h$

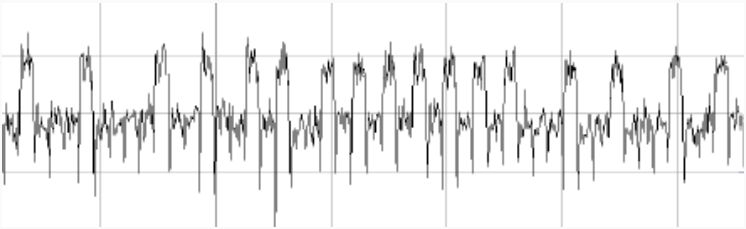
for $i = n$ to 0

  $T \leftarrow T \times T$

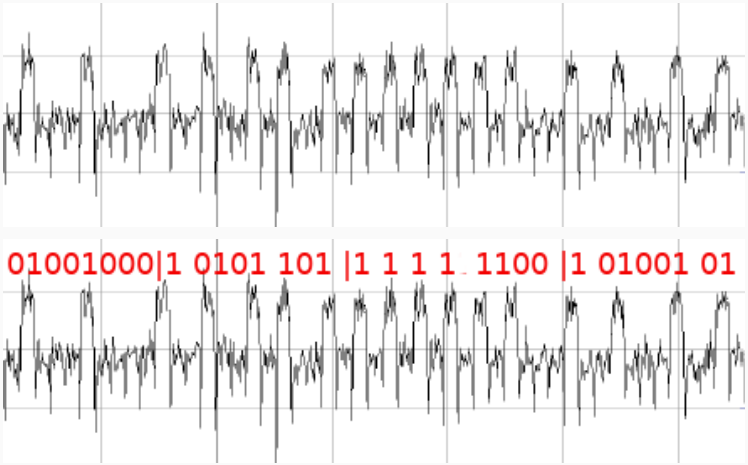
  if $d_i = 1$, $T \leftarrow T \times h$

return T

Power Consumption of Fast Exponentiation



Power Consumption of Fast Exponentiation



Constant consumption

To prevent reading the exponent from the consumption curve, the curve must be made constant. To do this: use dummy operations.

Compute h^d

$d = [d_n d_{n-1} \dots d_0]_2$

$T \leftarrow h$ and $U \leftarrow T$

for $i = n$ to 0

  $T \leftarrow T \times T$

  if $d_i = 0, U \leftarrow T \times h$

  if $d_i = 1, T \leftarrow T \times h$

return T

Here, inserting dummy operations prevents direct extraction of the key.

Speeding up RSA

Classic RSA (padding irrelevant)

The message m can be $OAEP(m')$, for example.

$$m = c^d \mod n$$

$\Longrightarrow$ We can do faster! RSA-CRT

Riddle...

Let there be x objects, number unknown.

- If arranged by 3, remainder is 2
- If arranged by 5, remainder is 3
- If arranged by 7, remainder is 2

How many objects are there?

Chinese Remainder Theorem

$n_1, n_2, \ldots, n_k$ pairwise coprime. Let $n = n_1 \times n_2 \times \ldots \times n_k$.

Given the system with known n_i and a_i:

$$\begin{cases} x = a_1 & \mod n_1 \\ x = a_2 & \mod n_2 \\ \ldots = & \ldots \\ x = a_j & \mod n_k \end{cases}$$

Chinese Remainder Theorem

There exists a unique solution modulo n with

$$x = \sum a_i \times e_i$$

where $e_i = k_i \times (k_i^{-1} \mod n_i)$ and $k_i = \frac{n}{n_i}$.

Speeding up RSA with CRT

Let the RSA modulus be $n = p \times q$, with p and q prime.

Twice as many computations — but much faster

Instead of computing

$$C = M^d \mod n \quad \text{e.g. 2048-bit multiplications}$$

we compute

$$C_1 = M^d \mod p \quad \text{and} \quad C_2 = M^d \mod q \quad \text{e.g. 1024-bit multiplications}$$

Compute $C_1 = M^e \mod p$, $C_2 = M^e \mod q$, $k_1 = \frac{n}{n_1} = q$, $k_2 = \frac{n}{n_2} = p$.

Using the Chinese remainder theorem result:

$$M = AC_1 + BC_2$$

Speeding up RSA with CRT

Again with $n = p \times q$ primes.

Compute:

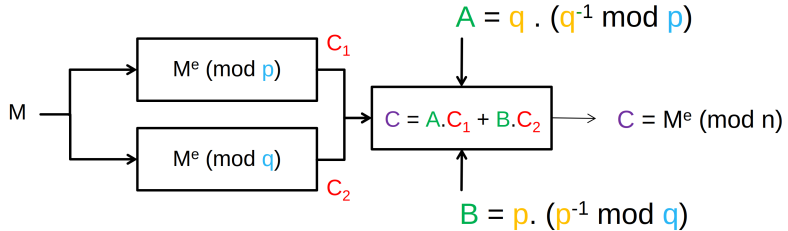
$$C_1 = M^e \mod p \quad C_2 = M^e \mod q \quad k_1 = \frac{n}{n_1} = q \quad k_2 = \frac{n}{n_2} = p$$

Using CRT:

$$M = e_1 \times C_1 + e_2 \times C_2$$

with $e_1 = q \cdot (q^{-1} \mod p)$ and $e_2 = p \cdot (p^{-1} \mod q)$, precomputed to speed up the calculation.

Speeding up RSA with CRT



Fault Attack

Principle

- Side-channel attacks exploit hardware weaknesses to recover information.
- Fault-injection attacks deliberately disrupt execution, for example using lasers.
- Analysing the faulty result reveals information on the secret.

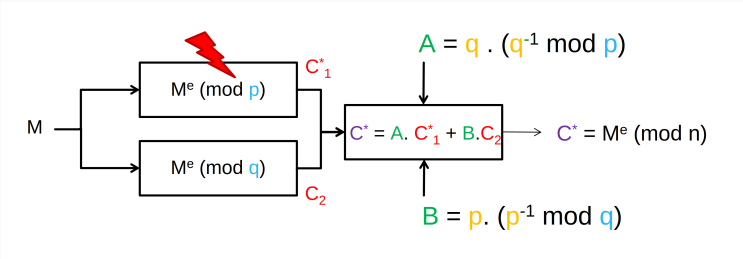
Target of the attack

We induce perturbations during execution when addition steps occur, e.g.:

- overvoltage
- laser pulse
- heating the device

The Fault

$$M_1 = C^e \mod p \quad \rightarrow \quad M_1 \text{ random.}$$



We sign the same message M twice: once without fault (C), once with fault (C^*).

$$C_1 \neq C_1^* \qquad C_2 = C_2^*$$

Attaque de Bellcore

Recall : $C = C_1 \cdot q \cdot (q^{-1} \mod p) + C_2 \cdot p \cdot (p^{-1} \mod q)$

$$C \mod q = \left[C_1 \cdot q \cdot (q^{-1} \mod p)\right] \mod q + \left[C_2 \cdot p \cdot (p^{-1} \mod q)\right] \mod q$$

$$= 0 + \left[C_2 \cdot p \cdot (p^{-1} \mod q)\right] \mod q$$

Similarly : $C^* \mod q = \left[C_2 \cdot p \cdot (p^{-1} \mod q)\right] \mod q$

Thus : $C \mod q = C^* \mod q$ c-à-d $q \div (C - C^*)$

However : $C \mod p = \left[C_1 \cdot q \cdot (q^{-1} \mod p) + p \cdot (p^{-1} \mod q)\right] \mod p$

$$= \left[C_1 \cdot q \cdot (q^{-1} \mod p)\right] \mod p$$

$$C^* \mod p = \left[C_1^* \cdot q \cdot (q^{-1} \mod p) + p \cdot (p^{-1} \mod q)\right] \mod p$$

In summary:

p and q primes, $n = p \times q$

q divides $(C - C^*)$ and p does not divide $(C - C^*)$

Therefore $GCD(C - C^*, n = p \times q) = q$

We have n, C and C^*, hence we deduce q, and then p.

We have everything!

Setting Up the Attack & Countermeasure

Conditions for the attack:

- Ability to sign the same message twice (message is arbitrary).
- Ability to inject a fault during exactly one exponentiation.

No assumption on the type of fault!

Countermeasure

- Verify the signature before returning it.
- Compute C_1 and C_2 twice (fault assumed random).

Outline

Where do I find public-keys? How to be sure of the real owner of a key?

Certificates

- $\text{cert}_{B \to C} = \text{Sign}_{sk_B}(id_C || pk_C)$: B certifies that C's public-key is pk_C
- If A trusts B:
  - C can send pk_C together with $\text{cert}_{B \to C}$
  - A can verify $\text{cert}_{B \to C}$ and accept pk_C as the public-key of C

Certificate authorities and chains

Certificate authority: trusted entities, used as roots in certificate chains

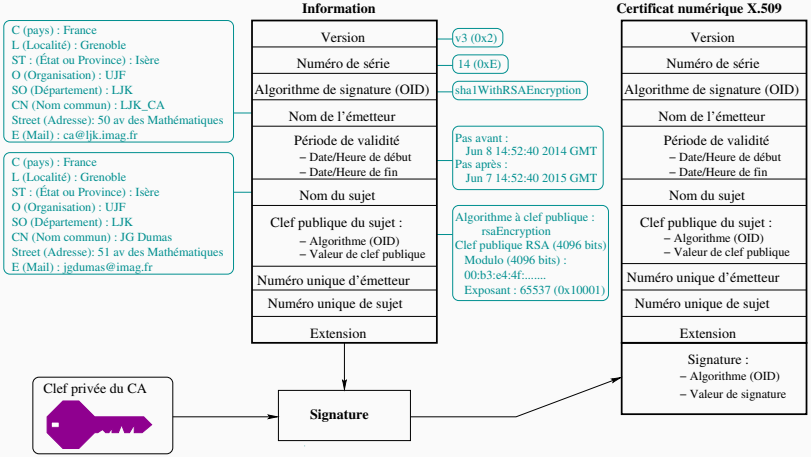
e.g. DigiCert

Certificate chains: trees of certifications, from authorities to end users

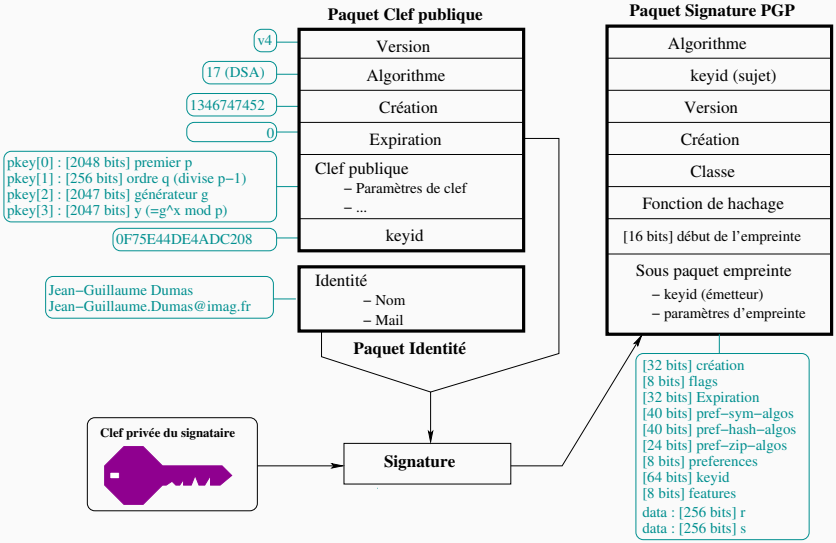
Certificate revocation

- Short-lived certificates: expiration date $\text{cert}_{B \to C} = \text{Sign}_{sk_B}(id_C || pk_C || T)$
- Certification revocation lists, using a serial number for each certificate

Certificat X509

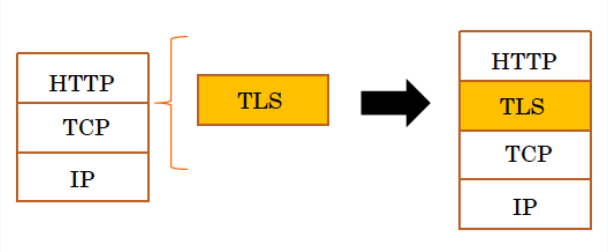


Certificat PGP



Outline

SSL/TLS Protocol



Core Security Properties

- **Confidentiality:** Encryption of traffic.
- **Integrity:** MAC verification on all records.
- **Authentication:** Certificates and handshake verification.
- **Forward secrecy:** Compromise of keys does not reveal past sessions.

- HTTP (80) → HTTPS (443)
- IMAP (143) → IMAPS (993)
- POP3 (110) → POP3S (995)

History

- SSL (*Secure Sockets Layer*), designed by Netscape
  - SSL 1.0 (1994) : theoretical protocol, never used
  - SSL 2.0 (1995-2011) : first version used
  - SSL 3.0 (1996-. . . , RFC 6101)
- TLS (*Transport Layer Security*), designed by IETF (*Internet Engineering Task Force*)
  - TLS 1.0 (1999-. . . , RFC 2246)
  - TLS 1.1 (2006-. . . , RFC 4346)
  - TLS 1.2 (2008-. . . , RFC 5246)
  - TLS 1.3 (2017 -. . . ,)
- Compatibility of HTTPS servers in (source : SSL Pulse)

	SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3
2016	8,4 %	26,8 %	98,2 %	71,9 %	74,2 %	0 %
2020	0,8 %	4,6 %	53,5 %	61,7 %	98,4 %	32,8 %
2025	0,1%	1%	23,5%	25,2%	100%	75,2%

Cryptography in TLS 1.2

Server [client] Authentification

- Public Key : RSA, DSS, ECDSA
- Secret key : PSK (*Pre-Shared Key*), SRP (*Secure Remote Password*)
- No authentication : ANON

Key Exchange

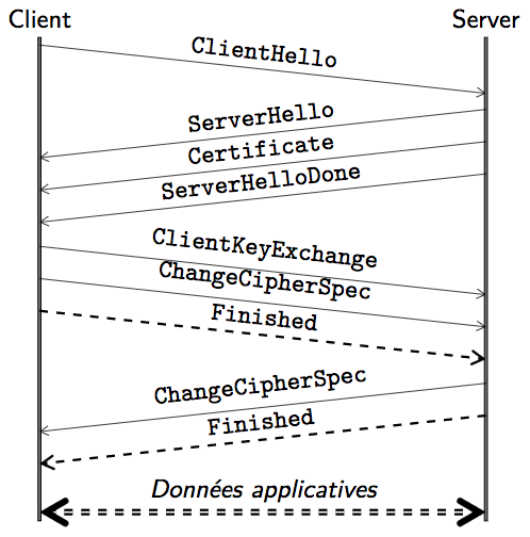
- Public Key (same on the server) : RSA
- Diffie-Hellman statique : DH, ECDH
- share secret : PSK, SRP
- Diffie-Hellman éphemeral (different secret keys each time) ; *forward* secrecy) : DHE, ECDHE

Cryptography in TLS 1.2

- Encryption :
  - Mode: AES-CBC, 3DES-CBC, DES-CBC, etc.
  - Stream cipher: RC4
  - No encryption: NULL
- Integrity and authentication:
  - HMAC : HMAC-MD5, HMAC-SHA1, HMAC-SHA256, etc.
  - AEAD: AES-CCM, AES-GCM, etc.

Example of *cipher* suite : TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

High level


Client
Server
ClientHello
ServerHello
Certificate
ServerHelloDone
ClientKeyExchange
ChangeCipherSpec
Finished
ChangeCipherSpec
Finished
Données applicatives

Handshake SSL/TLS simple

1. Client → Serveur
   - `ClientHello` : version supported, *cipher suites*
2. Serveur → Client
   - `ServerHello` : version supported, *cipher suite* chosen
   - `Certificate` (opt.) : server public key in a certificat X.509
   - `ServerKeyExchange` (opt.) : Diffie-Hellman partial key
   - `ServerHelloDone`

Handshake SSL/TLS simple

3. Client → Serveur
   - `ClientKeyExchange` : *PreMasterKey* encrypted by server public key, or Diffie-Hellman partial key
   - `ChangeCipherSpec`
   - `Finished` : message encrypted and authenticated using a transcript of the *handshake*
4. Serveur → Client (If `Finished` of the client is valid)
   - `ChangeCipherSpec`
   - `Finished` : message encrypted and authenticated using a transcript of the *handshake*
5. If `Finished` of the server is valid, then connexion is established.

Public Key Infrastructure

Allow to trust a public key !

- Certificat : signature of a public key by a trusted third party.

Two possible infrastructures

- Hierarchic : Certificate authorites (SSL/TLS et X.509, EMV)
- Decentralised : Web of Trust (PGP, GnuPG)

Certification Authorities

- Root CA : certifying public key of other CA
- Other CA : certifying server keys

Chain of trust :

Root CA $\xrightarrow{\text{sign}}$ CA 1 $\xrightarrow{\text{sign}}$ CA 2 $\xrightarrow{\text{sign}}$ Server

Authentication : server sends 3 certificats

- Its own public key, signed by CA 2
- Public key of CA 2, signed by CA 1
- Public Key of CA 1, signed by Root CA

Verification : If the client knowns and trusts the public key of Root CA, then he can verify all certificats.

Certification Authorities

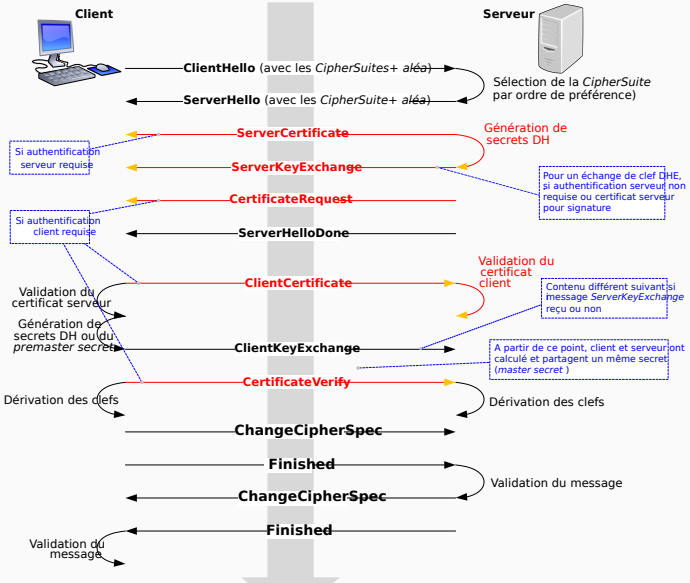
List of Trusted Root CA are in the browsers

- > 80 Root CA in Firefox
- Need to trust the browser!

Attention! CA security (root and others)

- If private key is compromise : false certificats like DigiNotar in 2011 : 500 false certificats, including `*.google.com`
- Regular security audits
- CRL (*Certificate Revocation List*) ou OCSP (*Online Certificate Status Protocol*)

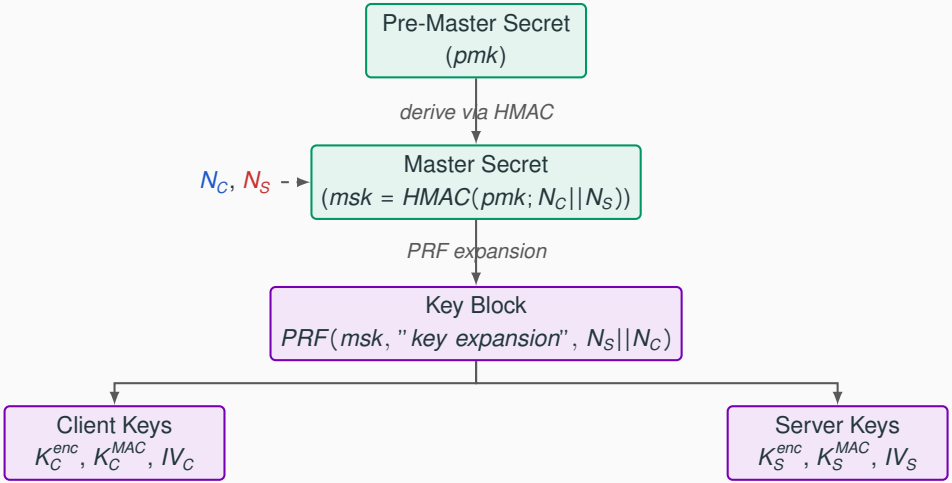
Application : TLS Handshake



Outline

Key Hierarchie in TLS 1.2

Aims : derive all sessions keys from the pré-master secret (*pmk*).



Client Serveur Key & dérivations Data / Intégrity

TLS 1.2 Overview

Goal: Establish a shared secret and authenticate peers.

Client (C)

- Chooses nonce N_C
- Proposes cipher suites, extensions
- *ClientHello*(N_C, *ciphers*)

Server (S)

- Chooses nonce N_S
- Owns (PK_S, SK_S) and certificate $Cert_S = \{S, PK_S\}_{CA}$
- *ServerHello*(N_S, $Cert_S$, *ciphers*)

Handshake goal: derive session keys K_C, K_S from pre-master key *pmk*:

$$msk = HMAC(pmk;\ N_C||N_S) \qquad K_C||K_S = HMAC(msk;\ 0||N_C||N_S)$$

Authentication by verifying $Cert_S$.

Computing the Pre-Master Key (*pmk*) — TLS 1.2 Modes

TLS 1.2 supports three main key establishment methods:

RSA Mode

- $pmk \xleftarrow{R} \{0,1\}^{48}$
- $KE_C = RSA_{PK_S}(pmk)$
- S decrypts with SK_S

No forward secrecy.

DH Mode

- $KE_S = g^{ke_S} \bmod p$
- $pmk = (KE_S)^{ke_C} \bmod p$
- S uses ke_S to recover pmk

Static → no forward secrecy.

DHE Mode

- Ephemeral ke_S, ke_C per session
- $pmk = g^{ke_C \cdot ke_S} \bmod p$
- Ensures forward secrecy.

All modes: *pmk* feeds HKDF-like HMAC chain for master secret derivation.

TLS 1.2 Handshake Flow (Simplified)

Client		Server
$ClientHello(N_C)$	$\longrightarrow$	
	$\longleftarrow$	$ServerHello(N_S, Cert_S)$
	$\longleftarrow$	$ServerKeyExchange(ifDHE)$
$KeyExchange(KE_C)$	$\longrightarrow$	
$ChangeCipherSpec$	$\longrightarrow$	
$Finished$	$\longrightarrow$	
	$\longleftarrow$	$ChangeCipherSpec, Finished$

At this point:

K_C, K_S established, *ApplicationData* encrypted under symmetric AEAD.

Integrity and Authentication in TLS 1.2

Integrity: verified through the Finished messages.

$$Fin_C = HMAC(msk; 1||\tau), \qquad Fin_S = HMAC(msk; 2||\tau)$$

where τ = transcript of all prior handshake messages.

Server Authentication:

- $Cert_S$ validated against trusted CA.
- Signature on ServerKeyExchange (if DHE) binds PK_S to key share.

Optional Client Authentication:

- Server requests client certificate.
- Client signs handshake transcript.

Session Resumption and Abbreviated Handshake

Goal: Reduce handshake latency by reusing existing msk_{sID}.

Protocol:

- $C \rightarrow S : N_C,\ sID$
- $S \rightarrow C : N_S,\ sID$
- Keys derived from stored msk_{sID}

Derivation:

$$K_C||K_S = HMAC(msk_{sID}; 0||N_C||N_S)$$

$$Fin_C = HMAC(msk_{sID}; 1||\tau)$$

$$Fin_S = HMAC(msk_{sID}; 2||\tau)$$

Resumption = 1-RTT handshake with previously authenticated session.

TLS 1.2 — Cryptographic Summary

- Key exchange: RSA, DH, or DHE $\rightarrow$ derive *pmk*
- Key derivation: HMAC-based chaining $\rightarrow$ msk, K_C, K_S
- Authentication: server via $Cert_S$, optionally client
- Integrity: Finished MACs over transcript
- Confidentiality: symmetric encryption (CBC or AEAD)

Cryptographic Limitation

RSA and static DH lack forward secrecy — DHE fixes this. TLS 1.3 removes these legacy modes entirely.

Summary

Session Freshness

- Nonces N_C and N_S involved in key derivation
- Prevent replay attacks

Server Authentication

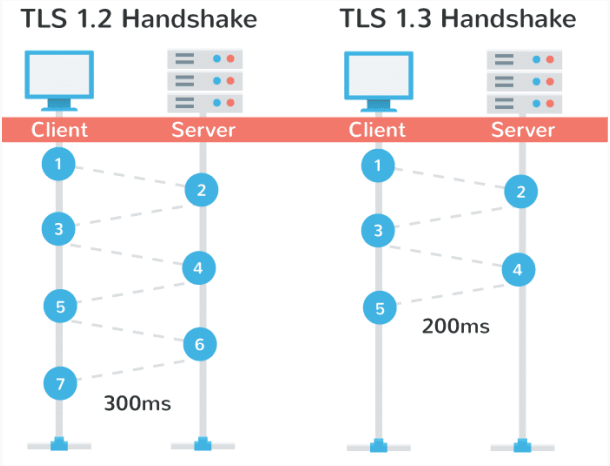
- Certificate enseures only server shares key with client
- Unilateral : anyone can exchange keys with server

Key Confirmation

- Last message: auhtenticated encryptuon with session keys
- Both parties are sure they computed the same keys

Outline

Aperçu


TLS 1.2 Handshake
Client
Server
1
2
3
4
5
6
7
300ms
TLS 1.3 Handshake
Client
Server
1
2
3
4
5
200ms

TLS 1.3

- Clean up: Remove unused or unsafe features
- Security: Improve security by using modern security analysis techniques
- Privacy: Encrypt more of the protocol
- Performance: Our target is a 1-RTT handshake for naive clients; 0-RTT handshake for repeat connections
- Continuity: Maintain existing important use cases

`https://tlswg.github.io/tls13-spec/`

TLS 1.3 removes obsolete and insecure features

- SHA-1
- RC4
- DES
- 3DES
- AES-CBC
- MD5
- Arbitrary Diffie-Hellman groups — CVE-2016-0701
- EXPORT-strength ciphers – Responsible for FREAK and LogJam

TLS 1.3 1-RTT handshake: 12 messages in 3 flights, 16 derived keys, then data exchange.

TLS 1.3

$C \rightarrow S : CHello, PK_C$, where $CHello = N_C||ciphers||ext$, supported by C

$S \rightarrow C : SHello, PK_S$ where $CHello = N_S||ciphers||ext||sub$, supported by S include in $CHello$.

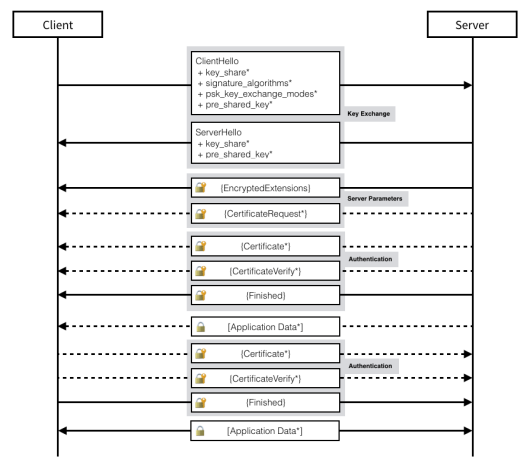
$S \rightarrow C : \{Cert_S\}_{htk}$

$S \rightarrow C : \{CVf\}_{htk}$

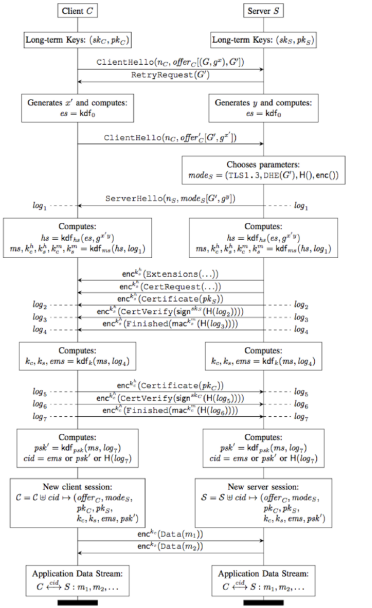
$S \rightarrow C : \{Fin_S\}_{htk}$

$C \rightarrow S : \{Fin_C\}_{htk}$

TLS 1.3


Client
Server
ClientHello
+ key_share*
+ signature_algorithms*
+ psk_key_exchange_modes*
+ pre_shared_key*
Key Exchange
ServerHello
+ key_share*
+ pre_shared_key*
{EncryptedExtensions}
Server Parameters
{CertificateRequest*}
{Certificate*}
Authentication
{CertificateVerify*}
{Finished}
[Application Data*]
{Certificate*}
Authentication
{CertificateVerify*}
{Finished}
[Application Data*]

TLS 1.3



Key Derivation Functions:

$\mathsf{hkdf\text{-}extract}(k, s) = \text{HMAC-H}^k(s)$

$\mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, h) =$
$\quad \text{HMAC-H}^s(len_{\mathsf{H}()} \| \text{“TLS 1.3,”} \| l \| h \| \text{0x01})$

$\mathsf{derive\text{-}secret}(s, l, m) = \mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, \mathsf{H}(m))$

1-RTT Key Schedule:

$\mathsf{kdf}_0 = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, 0^{len_{\mathsf{H}()}})$

$\mathsf{kdf}_{hs}(es, e) = \mathsf{hkdf\text{-}extract}(es, e)$

$\mathsf{kdf}_{ms}(hs, log_1) = ms, k_c^h, k_s^h, k_c^m, k_s^m$ where
- $ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$
- $hts_c = \mathsf{derive\text{-}secret}(hs, \text{hts}_c, log_1)$
- $hts_s = \mathsf{derive\text{-}secret}(hs, \text{hts}_s, log_1)$
- $k_c^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \text{key}, \text{“”})$
- $k_c^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \text{finished}, \text{“”})$
- $k_s^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \text{key}, \text{“”})$
- $k_s^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \text{finished}, \text{“”})$

$\mathsf{kdf}_k(ms, log_4) = k_c, k_s, ems$ where
- $ats_c = \mathsf{derive\text{-}secret}(ms, \text{ats}_c, log_4)$
- $ats_s = \mathsf{derive\text{-}secret}(ms, \text{ats}_s, log_4)$
- $ems = \mathsf{derive\text{-}secret}(ms, \text{ems}, log_4)$
- $k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ats_c, \text{key}, \text{“”})$
- $k_s = \mathsf{hkdf\text{-}expand\text{-}label}(ats_s, \text{key}, \text{“”})$

$\mathsf{kdf}_{psk}(ms, log_7) = psk'$ where
- $psk' = \mathsf{derive\text{-}secret}(ms, \text{rms}, log_7)$

PSK-based Key Schedule:

$\mathsf{kdf}_{es}(psk) = es, k^b$ where
- $es = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, psk)$
- $k^b = \mathsf{derive\text{-}secret}(es, \text{pbk}, \text{“”})$

$\mathsf{kdf}_{0RTT}(es, log_1) = k_c$ where
- $ets_c = \mathsf{derive\text{-}secret}(es, \text{ets}_c, log_1)$
- $k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ets_c, \text{key}, \text{“”})$

Outline

Attacks cryptography Specification Implentation randomness

- 1995 : Downgrade to SSLv2 during the negociation
- 1998 : Bleichenbacher atttack on PKCS#1 v1.5
- 2002 : Bad interpretation of X.509 (IE)
- 2008 : Overpass validation of OpenSSL certificats
- 2009 : Collision MD5 on concrete certifiats
- 2009 : Attack on renegociation
- 2009 : confusion of nuls characters in the certificats
- 2011 : BEAST Browser Exploit Against SSL/TLS, (IV implicit in CBC mode)
- 2011 : Bad interpretation of the extension X.509 (iOS)
- 2012 : Mining your Ps and Qs (no randomness in RSA key generation)
- 2012 : CRIME, Compression Ratio Info-leak Made Easy
- 2013 : Lucky 13 (oracle of padding CBC) + biais statistic on RC4
- 2014 : goto fail Apple
- 2014 : Overpass validation of GnuTLS certificats
- 2014 : Triple Handshake (renegociation and ressumption session)
- 2014 : Heartbleed and EarlyCCS
- 2014 : POODLE
- 2015 : FREAK and LogJam
- 2016 : DROWN

Outline

OpenPGP — Overview

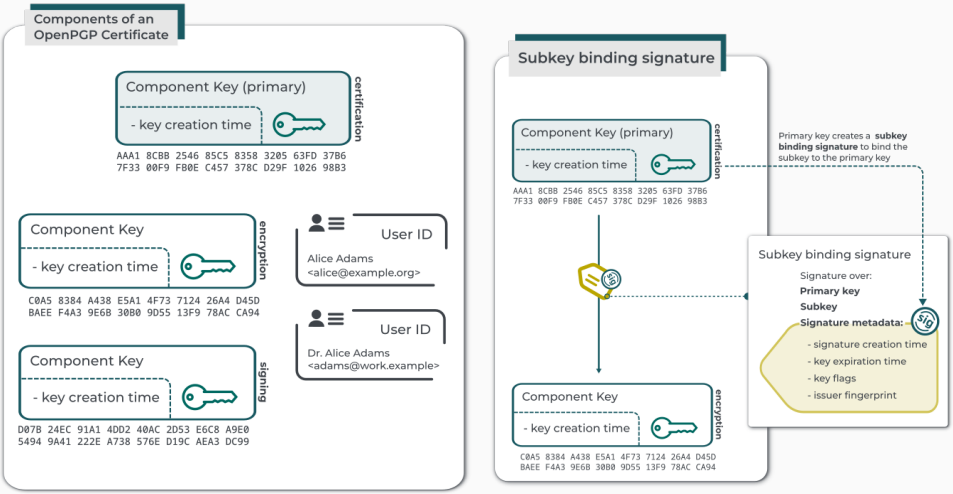
What is OpenPGP?

- Open standard for encrypting, signing, and verifying data (RFC 4880)
- Used by GnuPG, ProtonMail, Thunderbird, etc.
- Combines:
  - **Public-key cryptography** (for key exchange + signatures)
  - **Symmetric cryptography** (for message encryption)
  - **Hash functions** (for integrity + signatures)

OpenPGP — Key Concepts

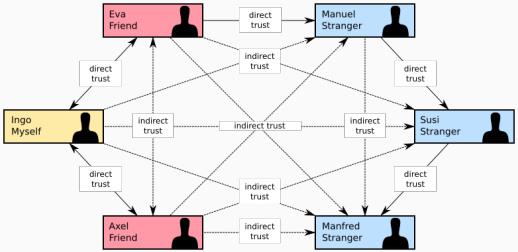
- Each user has a **Key Pair**:
  - **Private key** — kept secret, used for decryption and signing.
  - **Public key** — shared with others, used for encryption and verification.
- Keys can include:
  - User ID(s): name, email
  - Multiple subkeys (e.g., one for signing, one for encryption)
- Public keys are distributed via:
  - Key servers
  - Direct sharing (email attachments, QR codes)
  - The **Web of Trust** — users sign each other's keys

OpenPGP — Signature Process



How Do I Know a Public Key Belongs to the Right Person?

- Unlike PKI (X.509), OpenPGP does not rely on a single CA
- Instead, users **sign each other's keys**:
  - Alice verifies Bob's key in person (checks fingerprint)
  - Alice signs Bob's key ⇒ trust statement
  - Others can rely on Alice's signature to trust Bob's key



Cryptographic Components in OpenPGP

- **Public-key algorithms:** RSA, ElGamal, Ed25519, ECDH
- **Symmetric ciphers:** AES-128/192/256, Camellia, 3DES
- **Hash functions:** SHA-256, SHA-512 (legacy: SHA-1, MD5)
- **Compression:** ZIP, ZLIB — optional, before encryption

Security Notes:

- Prefer modern algorithms (AES, Ed25519, SHA-256)
- Avoid deprecated ones (IDEA, MD5, SHA-1)

Outline

Birthday Paradox : Hash Function

Let an Hash function $H : D \to 2^k$ (think $D = \{0,1\}^*$)

Naïve Collision

Birthday Paradox : Hash Function

Let an Hash function $H : D \to 2^k$ (think $D = \{0,1\}^*$)

Naïve Collision

With $2^k + 1$ try there is a collision

$$P(\textit{at least } 1 \textit{ collision}) = 1 - P(\textit{no collision})$$

Probability of no collision

Birthday Paradox : Hash Function

Let an Hash function $H : D \to 2^k$ (think $D = \{0,1\}^*$)

Naïve Collision

With $2^k + 1$ try there is a collision

$$P(\textit{at least } 1 \textit{ collision}) = 1 - P(\textit{no collision})$$

Probability of no collision

- Try $1 : 1 - 0$
- Try $2 : 1 - 1/2^k$
- $\vdots$
- Try $q : 1 - (q-1)/2^k$

$$P(\textit{no collision}) = \prod_{i=1}^{i=q} (1 - i/2^k)$$

Birthday Paradox : Hash Function

- $P(\textit{no collision}) = \prod_{i=1}^{i=q}(1 - \frac{i}{2^k})$
- $P(\textit{at least } 1 \textit{ collision}) = 1 - P(\textit{no collision})$

Using $e^{-x} \approx 1 - x$ (as $e^{-x} = 1 - x + \frac{x^2}{2!} + \cdots$) on $P(\textit{no collision})$, we have :

$$P(\textit{no collision}) \approx \prod_{k=0}^{q-1} e^{-\frac{i}{2^k}} = e^{-\sum_{i=1}^{i=q-1}(i/2^k)} = e^{-q(q-1)/2^k}$$

If you want a probability of ϵ to have a collision

Need of solve $P(\textit{at least } 1 \textit{ collision}) = \epsilon = 1 - e^{-q(q-1)/2^k}$

$$q(q-1) = 2^{k+1} \ln(1/(1-\epsilon))$$

$$k \approx \sqrt{2^{k+1}} \ln(1/(1-\epsilon))$$

Examples

- $\epsilon = \frac{1}{2} \Rightarrow k \approx 1.177\sqrt{2^{k+1}}$
- $\epsilon = \frac{3}{4} \Rightarrow k \approx 1.665\sqrt{2^{k+1}}$
- $\epsilon = 0.9 \Rightarrow k \approx 2.146\sqrt{2^{k+1}}$

Remark: if 2^{k+1} is 365 among $1.77\sqrt{365} \approx 23$

So should be at least > 64 or even 80. > 128 or 160 to resist birthday attack.

Thank you for your attention.

Questions ?