Security - Signature & Certificate

1. Digital Signature
2. Certificate
3. Public Key Infrastructure (PKI)

1. Digital Signature

Repudiation

Repudiation means **denying you sent something**.

Example:

A sends a message with an HMAC to B.
Later, A says, “I never sent that! B made it up!”

A Denies Sending the Message

If A and B share the same secret key (as in HMAC):

- **Case 1 – A is lying:**
  A really sent the message but now denies it.
  Because B also knows the shared key, A can claim “B could have faked it.”

- **Case 2 – A is actually compromised:**
  Someone else (a hacker or even B) used A’s shared key to create the message.
  In this case, A truly didn’t send it — but it looks like A did.

We need a Way to Solve this Problem

Either way, we can’t prove who actually sent it
because **both sides share the same key**.

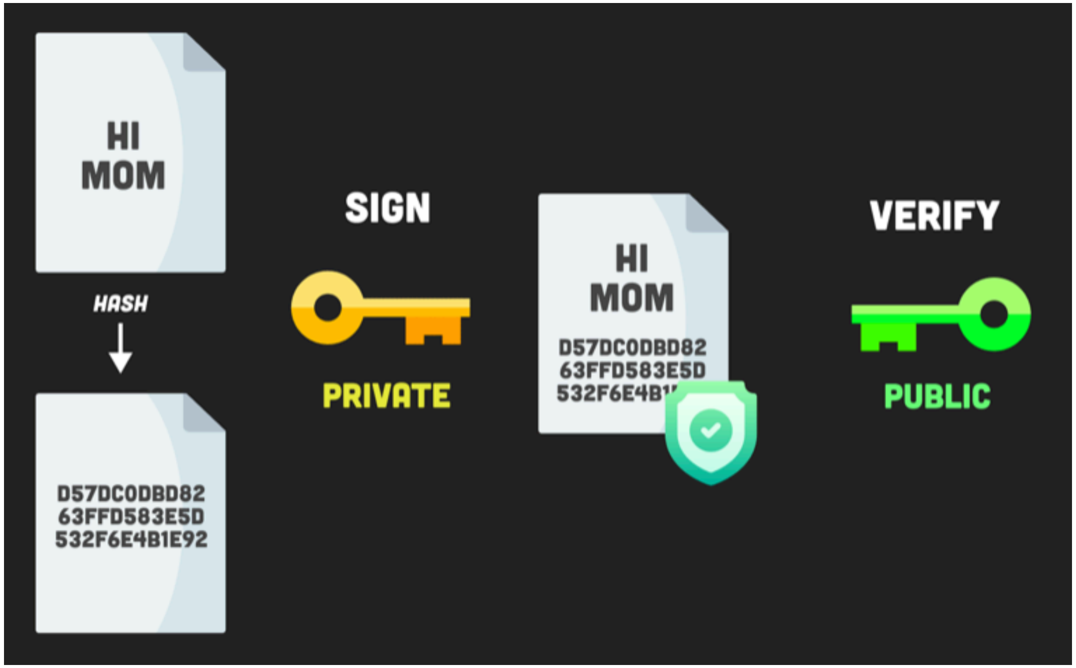
- That’s why we use a **Digital Signature**,
  which uses **A’s private key** (known only to A).
- Then only A could have signed the message —
  so A **can’t deny it later** (no repudiation).

What is Digital Signing?

Signing is the process of creating a digital signature of a message.

- A signature is a `hash` of the original message, which is then `encrypted` with the sender's private key (not the shared key as in HMAC)
- The signature can be verified by the recipient using the public key of the sender

This can guarantee the original message is authentic and unmodified (integrity verified).



How to Use Digital Signature

Goal: Make sure file F is not modified (Integrity + Authenticity)

Scenario using Digital Signature

1. **A (the sender)**
    - Creates a **hash** of file F.
    - **Encrypts the hash using A's *private key*** → this becomes the **digital signature**.
    - Sends **F + signature** to B.

2. **B (the receiver)**
    - **Decrypts the signature using A's *public key*** → gets the original hash.
    - **Generates a new hash** from the received file F.
    - Compares both hashes.
        - If they match → file is original and from A.
        - If not → file was changed or fake.

Why This Works

- Only A’s **private key** can create that signature.
- Anyone can verify it using A’s **public key**.
- So, B knows:
    i. The file wasn’t changed (**Integrity**)
    ii. It really came from A (**Authenticity**)

JavaScript Code

Creating a Signature

We need to check the integrity of the data. We create a signature from a private key and a sign that contains the original data.

```javascript
const { createSign, createVerify } = require('crypto');
const { publicKey, privateKey } = require('./keypair');

const data = 'I need to sign this document.';

// SIGN
const signer = createSign('rsa-sha256');
signer.update(data);
const signature = signer.sign(privateKey, 'hex');
```

Verifying the Signature

When we receive the data and signature, we can create a verifier with data to verify the results.

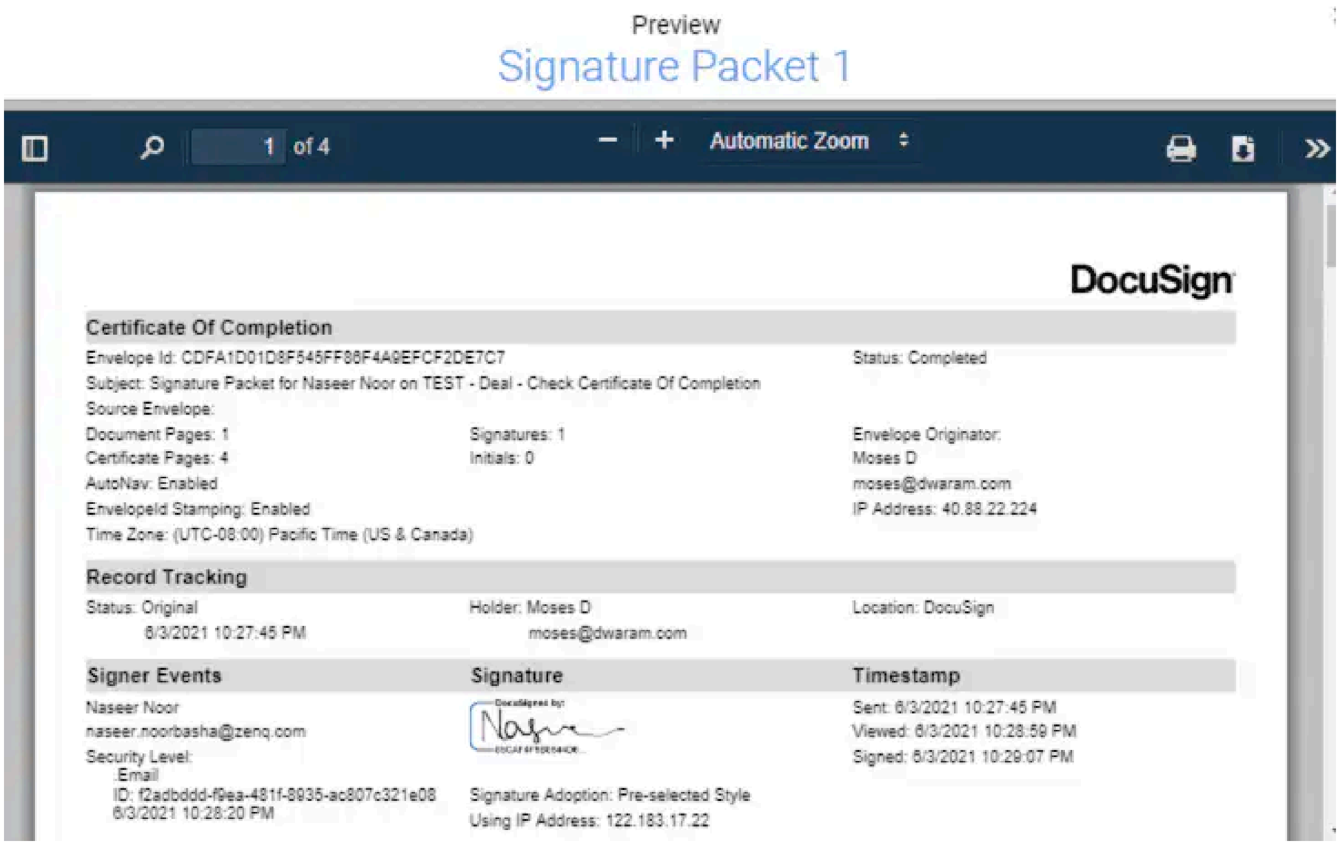
```
const verifier = createVerify('rsa-sha256');
verifier.update(data);
const isVerified = verifier.verify(publicKey, signature, 'hex');
console.log(isVerified);
```

This is an example of signature (one long line):


```
4a35450c510aeee57291d272e0cef367877b56052f364f76a244988
98c23ebacfa4435fb401f179bccbdb3df942d96209bc194b2854fd4
13df9bf5c4bccef05b621afbbfda06a2e8fb5676bcba8e4cc465f03
d7220ecb2897eef184e65c81121ecfa2493b43b415573de56d226f1
35a665e12c3cccfa3a7f0781b12fd75e709a7ad25506d1951cf0005
0adafacef22a3e946fdd693da7e399347b6179f6a219bfc4c6b0cb6
e5424d9bf388409b613e3ca71bbdffde3c05741db56ab58676c584e
1ce53e8f4e3c15c305d5c2790ef29ff0828b54f81c37c6547c20154
a0ace931e7ce8099c93b708d8bb1a963e5375ee5ed3c626cd46ee67
0f62b542c5de37d21
```

Real World Example: DocuSign

Docusign PDF with Signature (Visible) & Digital Signature (Hidden)


Preview

Signature Packet 1

1 of 4 Automatic Zoom

DocuSign

Certificate Of Completion

Envelope Id: CDFA1D01D8F545FF88F4A9EFCF2DE7C7 Status: Completed

Subject: Signature Packet for Naseer Noor on TEST - Deal - Check Certificate Of Completion

Source Envelope:

Document Pages: 1	Signatures: 1	Envelope Originator:
Certificate Pages: 4	Initials: 0	Moses D
AutoNav: Enabled		moses@dwaram.com
EnvelopeId Stamping: Enabled		IP Address: 40.88.22.224
Time Zone: (UTC-08:00) Pacific Time (US & Canada)		

Record Tracking

Status: Original 6/3/2021 10:27:45 PM	Holder: Moses D moses@dwaram.com	Location: DocuSign

Signer Events	Signature	Timestamp
Naseer Noor naseer.noorbasha@zenq.com Security Level: .Email ID: f2adbddd-f9ea-481f-8935-ac807c321e08 6/3/2021 10:28:20 PM	DocuSigned by: Signature Adoption: Pre-selected Style Using IP Address: 122.183.17.22	Sent: 6/3/2021 10:27:45 PM Viewed: 6/3/2021 10:28:59 PM Signed: 6/3/2021 10:29:07 PM

Visible Signature

When you open a completed envelope or signed PDF, you might notice:

1. Visible Signature: The signer’s typed or drawn signature (“Naseer Noor”) is shown clearly.
2. Digital Certificate Seal
3. Audit Trail

- The table at the bottom records:
- Who signed (Naseer Noor <email>) and when it was sent, viewed, and signed (timestamps)
- IP address used for signing

Hidden (Embedded) Signature

Digital Signature is used to verify its authenticity: and it is hidden in the PDF:

```
/Type /Sig
/Filter /Adobe.PPKLite
/SubFilter /adbe.pkcs7.detached
/ByteRange [...]
/Contents <3082...>  ← Encrypted digital signature (huge Base64 blob)
```

What Happens Behind the Scenes

1. When the document is finalized:

- DocuSign computes a hash of the PDF content.
- It encrypts that hash using DocuSign's or the signer's private key (PKI).
- That (1) encrypted hash (signature) + (2) certificate (with public-key) are embedded into the PDF.

2. Anyone opening the file in Adobe Acrobat or similar software:

- The program reads the embedded signature (encrypted hash).
- It uses the public key in the certificate to decrypt the signature.
- It recalculates the document hash and compares it.
- If they match → “Signature valid.”, if not → “Document has been altered.”

2. Certificate

In the previous example, the **certificate** included a **public key**.

But how can we be sure that this public key hasn’t been faked or compromised?

That’s where the **Certificate** and **CA** come in.

Certificate = Public Key + ID + CA Stamp

A **digital certificate** is like an online ID card.

It contains:

- **Public Key** – used for encryption or verification
- **Identity Info** – owner name, organization, domain, validity period
- **CA Stamp** – a digital signature from a trusted Certificate Authority (CA) proving authenticity

> Confirms that *this public key truly belongs to this owner.*
> (Think of it as a driver's license issued by the government.)

Certificate Example 1 – Apple Developer Certificate

The iPhone is used by billions of users, and anyone can download apps from the App Store.

- But what if a hacker uploads a fake or modified app pretending to be from Apple or Microsoft?
- To prevent this, Apple uses the **Apple Developer Certificate**.

> Each developer’s app is **digitally signed** with their unique **certificate**.
> The App Store and iOS verify the signature to ensure the app truly comes from a trusted developer.

Certificate Structure

```
┌──────────────────────────────────────────────┐
│  Issuer: Apple CA                            │
│  Subject: John Doe (Team ID: ABC123)         │
│  Valid: 2024.01.01 ~ 2045.01.01              │
│  Public Key: MIIBIjANBg...                   │
│  Signature: Apple's digital seal             │
└──────────────────────────────────────────────┘
```

- The signature is a cryptographic hash of the certificate data, encrypted using Apple CA’s private key.
- As we already have Apple's public key, we can verify that the certificate is not compromised.

Certificate Request Process

```
Developer (You)                 Apple
    │                            │
    │ 1. Generate Key Pair       │
    │    - Private Key (Secret)  │
    │    - Public Key            │
    │                            │
    │ 2. Create & Send CSR       │
    │    (includes public key)   │
    │──────────────────────────> │
    │                            │ 3. Verify Identity
    │                            │    - Check payment
    │                            │    - Validate account
    │                            │
    │ 4. Receive Certificate     │
    │<────────────────────────── │ Issue Certificate
    │                            │ (Signed by Apple)
    │ 5. Store in Keychain       │
    │    - Certificate           │
    │    - Private Key           │
    └────────────────────────────┘
```

Now, in the Keychain, we have (a) Certificate from Apple, and (b) Private key to make a signature.

CSR (Certificate Signing Request)

A **CSR** is a file you create when you request a digital certificate from a Certificate Authority (CA), such as Apple; think of it as a driver's license request to the government.

It includes:

Field	Description
Public Key	The public part of your key pair
Identity Info	Name, organization, email, country, etc.
Optional Extensions	e.g., Team ID, usage type (signing, encryption)
Signature	Encrypted with your **private key** to prove you own it

How Certificates Are Used - Code Signing

Here's what happens when you build your app:

```
┌──────────────────┐      ┌──────────────┐      ┌──────────────────┐
│   Your App       │      │(a) Developer │      │  Signed App      │
│                  │      │  Certificate │      │                  │
│ - Executable     │ ───> │(b) Private   │ ───> │ - Executable     │
│ - Resources      │      │     Key      │      │ - Resources      │
│ - Info.plist     │      │              │      │ - Signature      │
└──────────────────┘      └──────────────┘      └──────────────────┘
                                 │
                                 ▼
                          Hash of app
                          encrypted with
                          private key
```

The app is signed by you and certified by Apple — so users' devices can trust and safely run it.

Why So Complex? - Chain of Trust

```
Apple Root CA (Top Level Authority)
    │
    ├──> Apple Worldwide Developer Relations CA
    │       │
    │       ├──> John Doe's Developer Certificate
    │       │       │
    │       │       └──> Apps signed by John
    │       │
    │       └──> Jane Smith's Developer Certificate
    │               │
    │               └──> Apps signed by Jane
    │
    └──> Other Apple Services...
```

Apple's Root CA vouches for Apple's Developer CA, which vouches for your Developer Certificate, which signs your app.

Verification Process When Running App on iPhone

Step	What iOS Verifies	Where the Data Comes From
1 Signature	Hash of app matches the signature	From app bundle
2 Certificate	Certificate is valid & signed by Apple CA	From app bundle
3 Chain	Trace back to Apple Root CA	Apple Root CA is built into iOS
4 Provisioning	Device is allowed to run it	From embedded provisioning profile

3. Public Key Infrastructure (PKI)

When two parties communicate securely, they must **trust each other's public keys** — but how can we be sure that a public key really belongs to the claimed person or organization?

This is the **problem that PKI solves.**

Key Distribution Center (KDC)

Because of PKI, we can safely send emails, make online transactions,
and exchange sensitive data with confidence that the other party is genuine.

> The **Key Distribution Center (KDC)** is one example of a PKI component that distributes and verifies cryptographic keys.

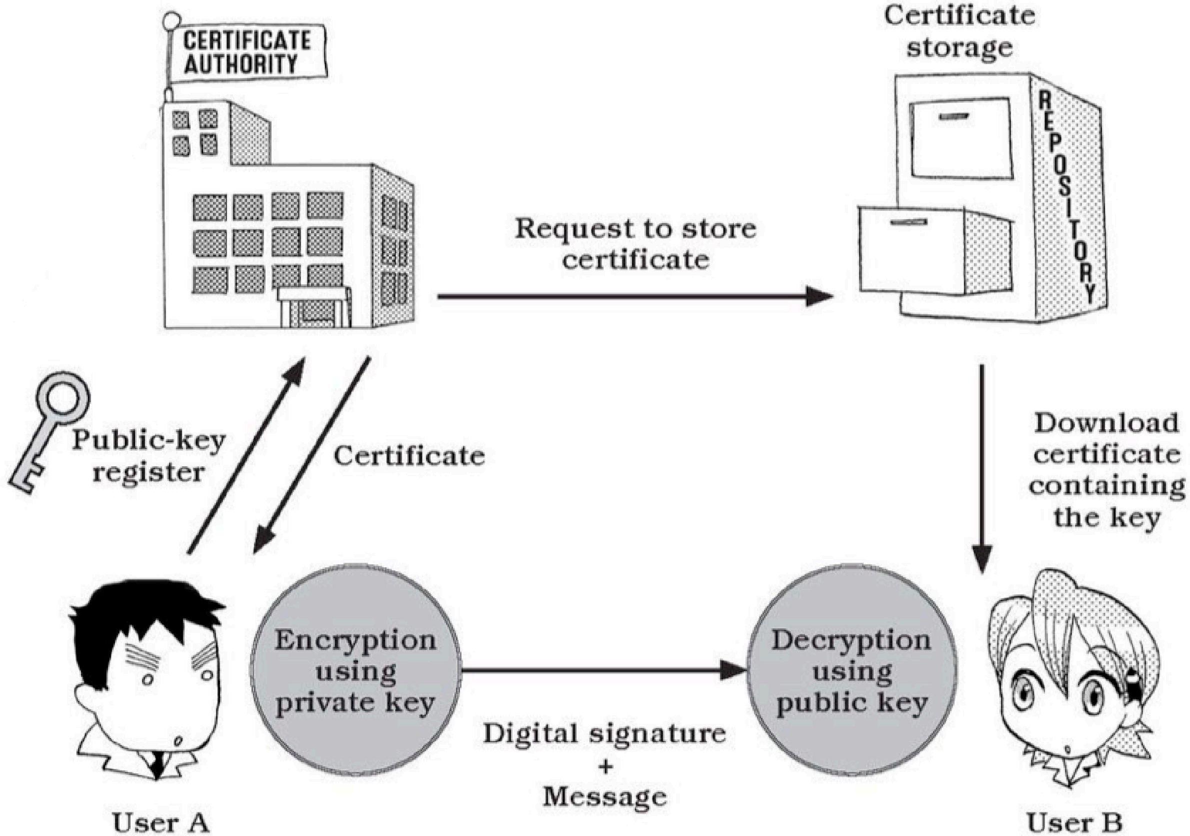
How KDC Works

- Anyone can register their **public key** in the **KDC**,
  or the KDC can generate public keys when necessary.

- When **User A** registers their public key using CSR
  (Certificate Signing Request),
  the KDC issues a **digital certificate** that verifies A's identity
  and key.

- When **User B** needs A's public key,
  B can safely retrieve **A's authenticated public key** from the
  KDC.

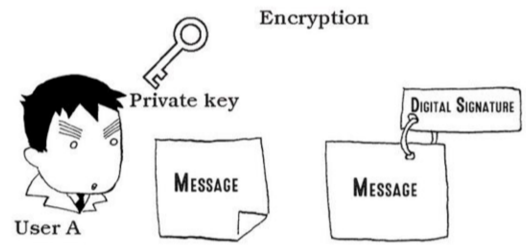
✅ This ensures that B can trust the key really belongs to A,
and not to an attacker pretending to be A.


CERTIFICATE AUTHORITY
Certificate storage
REPOSITORY
Request to store certificate
Public-key register
Certificate
Download certificate containing the key
Encryption using private key
Decryption using public key
Digital signature
+
Message
User A
User B

Message Exchange Using KDC

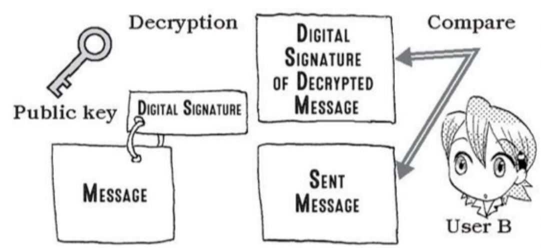
1. **User A (Sender)** uses their **private key** to create a **digital signature** for the message.
2. **User B (Receiver)** uses **User A's public key** (obtained from the KDC or certificate) to **verify** the signature.
3. If the **two hashes match**, the message is:
   - ✅ **Authentic** (from A)
   - ✅ **Untampered** (integrity preserved)

Sender (User A):



→

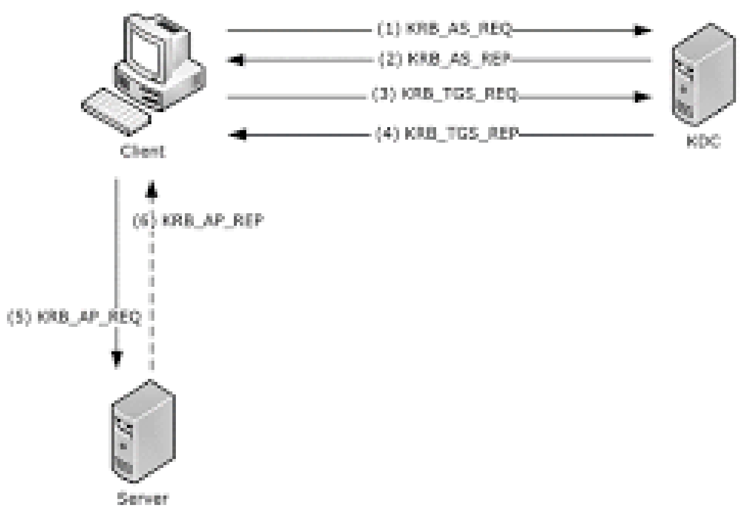
Receiver (User B):



Kerberos KDC Protocol

Kerberos is the most widely used **KDC (Key Distribution Center)** protocol.

- It provides **secure authentication** between users and services over an **untrusted network** — without sending passwords directly.



KERBEROS

The Trusted Certificate Authority (CA)

CA’s Role in PKI

- The **CA verifies the identity** of User A.
- Once verified, the CA **issues a digital certificate** that:
  - Contains **User A’s public key**
  - Includes **User A’s identity information**
  - Is **digitally signed by the CA**

We know Apple issues Apple developer certificate as the CA.

Certificate Lifecycle

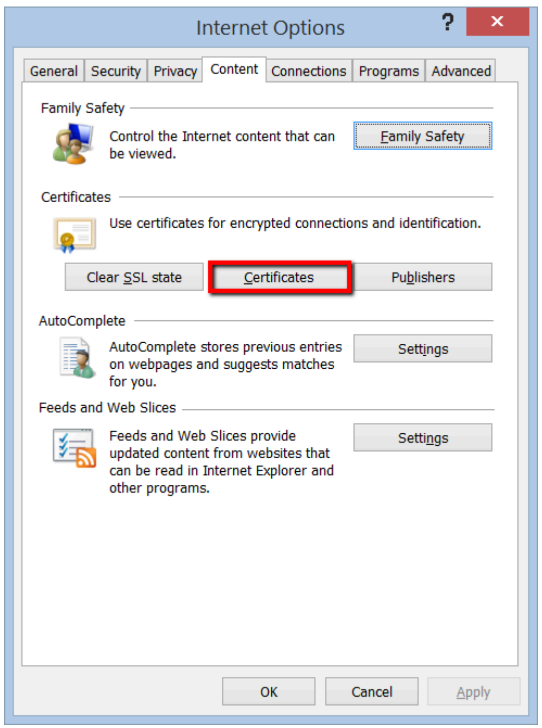
1️⃣ **User A** → creates key pair (private/public key)
2️⃣ **User A** → sends a **Certificate Signing Request (CSR)** to CA
3️⃣ **CA** → verifies identity and **signs** A's public key
4️⃣ **CA** → publishes A's **digital certificate**

Now, anyone who trusts the CA can safely use A's public key — because it's **certified** by the CA's own digital signature.

> ✅ The CA is the root of trust in the Public Key Infrastructure (PKI).

Automatization of the CA Related Process

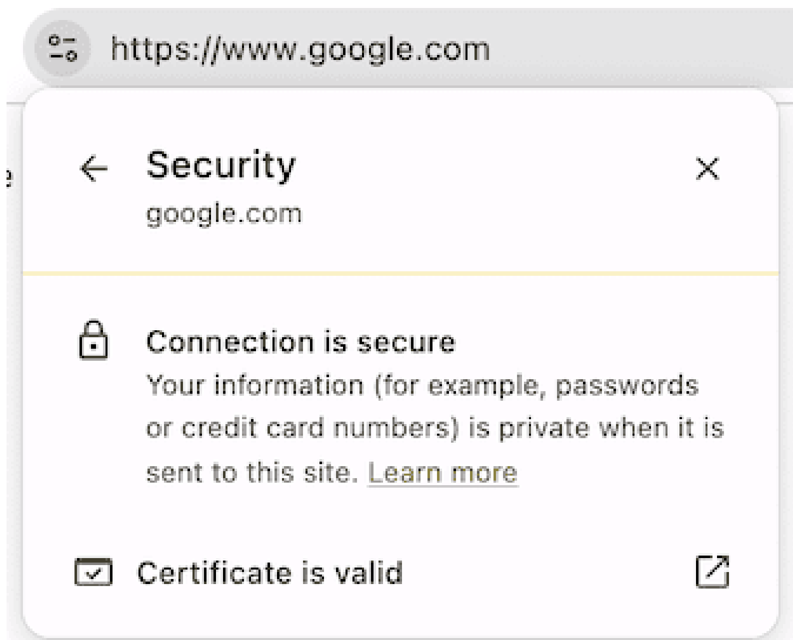
In practice, all validation and authentication steps are handled automatically by software.



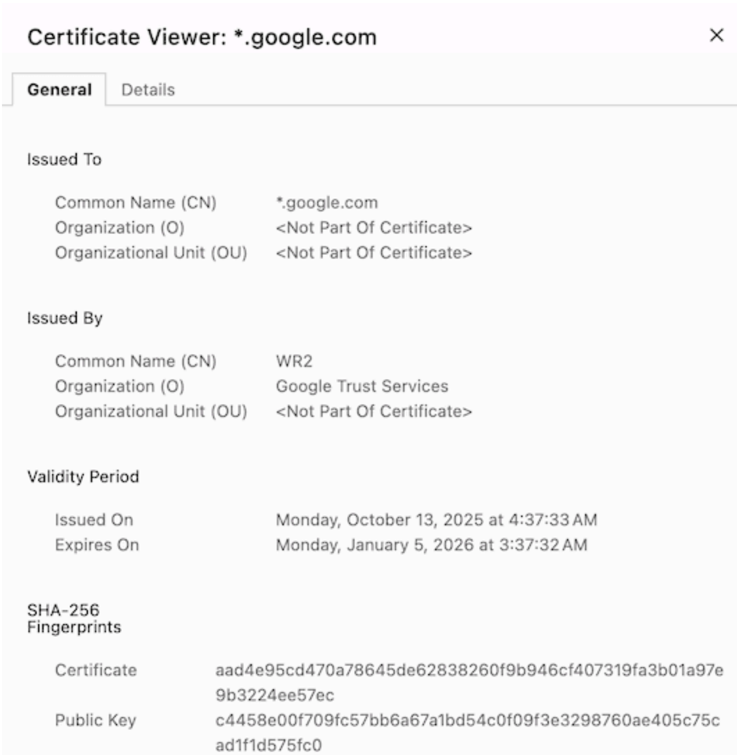
- Modern **web browsers**, **registration systems**, and **card readers** already include trusted CA information, allowing them to **verify certificates automatically** without user intervention.

Https and Certificate

Compared to the http protocol, the https protocol uses certificate to verify if the website we visit is the real one, not imposter's.



Certificate = Public Key + ID + CA Stamp



- **Public Key** – used for encryption or verification
- **Identity Info** – owner name, organization, domain, validity period
- **CA Stamp (signature)** – a digital signature from a trusted Certificate Authority (CA) proving authenticity

Certificate and Public Key

1. Browser already stores the CA’s public keys (trusted roots).
2. When it receives a website’s certificate, it checks the digital **signature** on it.
3. That **signature** was created by the CA using its private key.
4. The browser uses the CA's public key to verify that **signature** using hash comparison.

- Compute the hash of certificate
- Decrypt the signature to get the hash

5. If the signature is valid, the certificate (and its public key) is confirmed authentic and unaltered.

Details

- Issued To: the domain (*.google.com)
- Issued By: a Certificate Authority (CA) Google Trust Services
- Validity period: start and expiration dates
- SHA-256 Fingerprints: Certificate

What It Proves

- The certificate’s "Issued to" domain matches the site you’re visiting (google.com).
- It's signed by a trusted CA, your browser knows it’s the real site — not a fake.

If something’s wrong (expired, wrong name, untrusted CA), browsers show warnings like

⚠️ “Your connection is not private”