Takahiko Kawasaki · Follow

13 min read · Jul 8, 2020

Introduction

This article explains **X.509 certificate**.

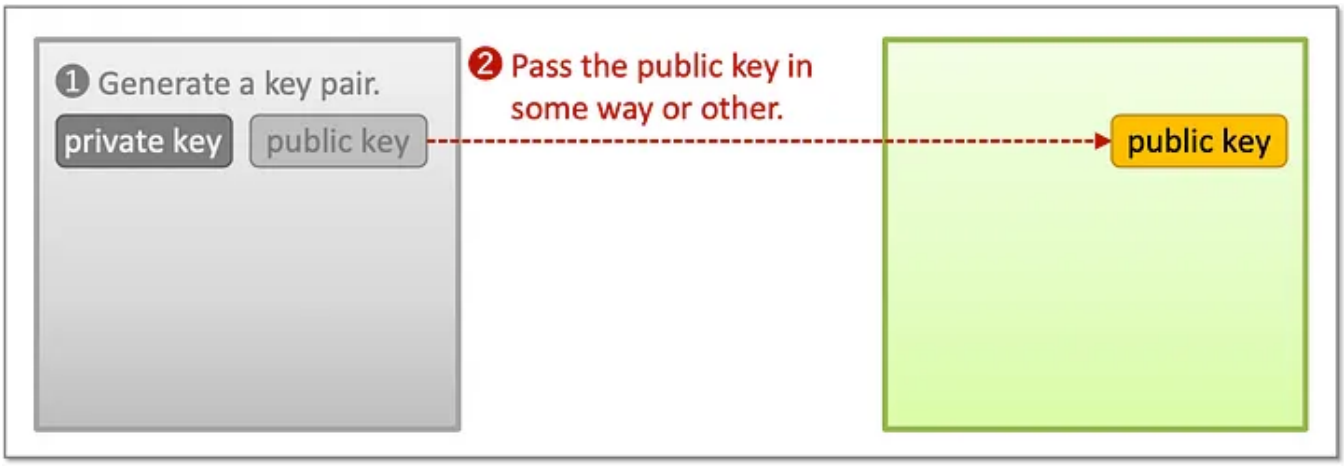
1. Digital Signature (Prior Knowledge)

To read this article, knowledge about **digital signature** is needed. That is, this article assumes that you understand "By verifying **signature** with the **public key** which is paired with the **private key** used to generate the signature, you can confirm that the target data has been surely signed by the owner of the private key and has not been altered in transit."

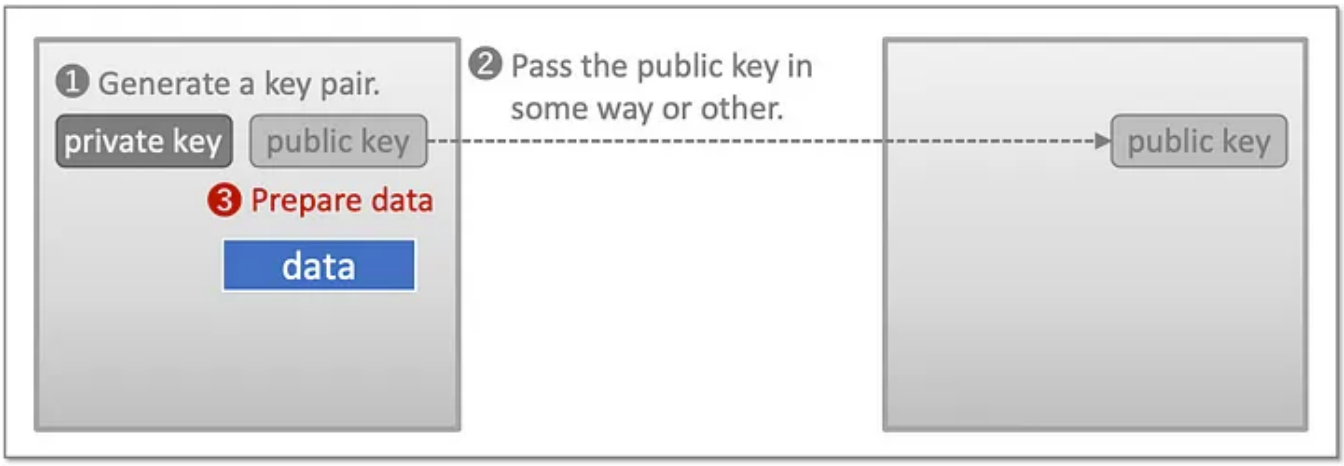
In other words, this article assumes that you know the flow illustrated below.

(1) Generate a pair of private key and public key.

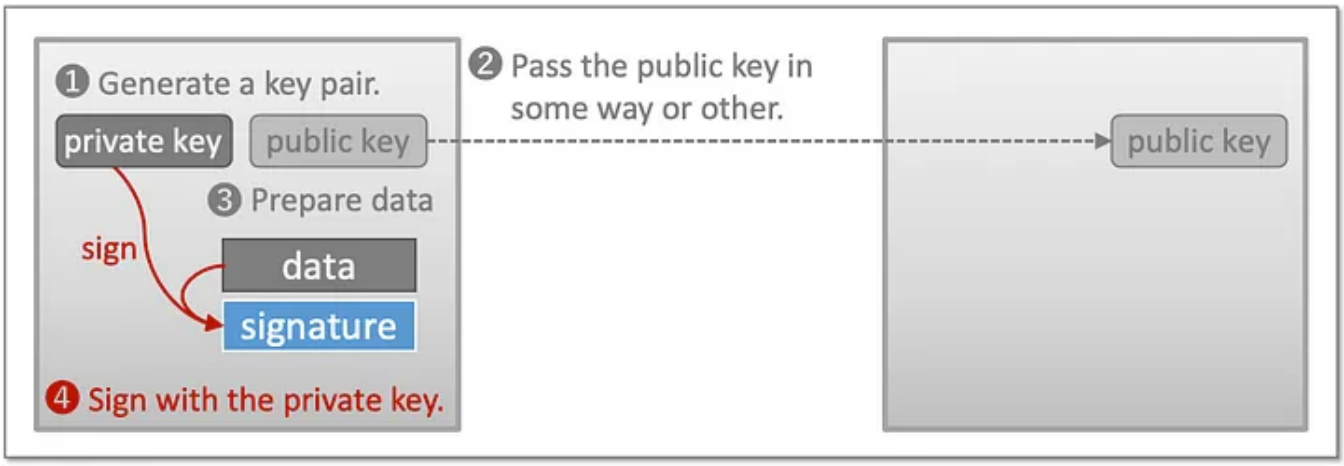
(2) Pass the public key to the other party in some way or other.



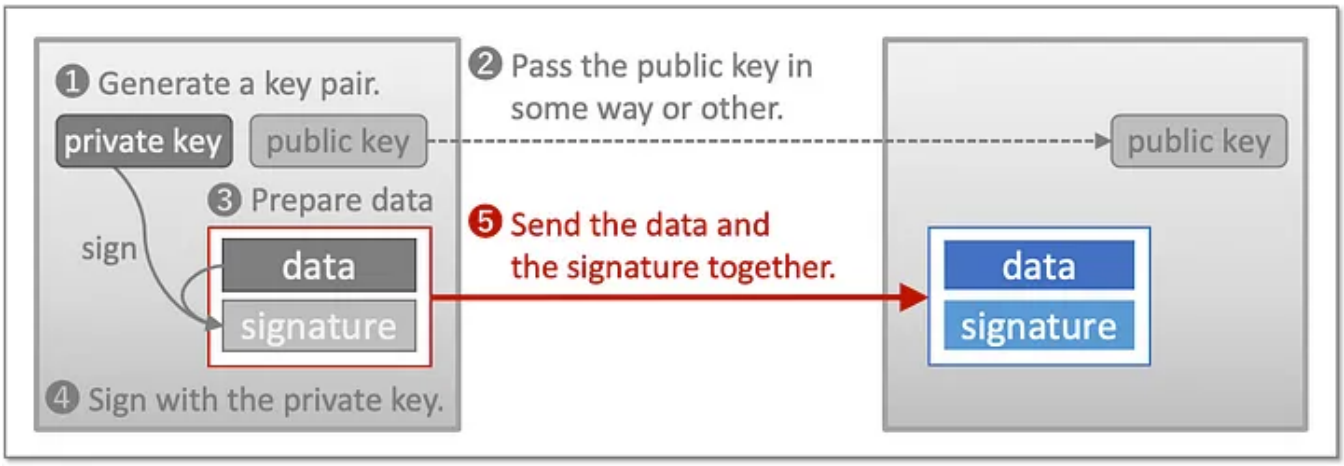
(3) Prepare data to send.



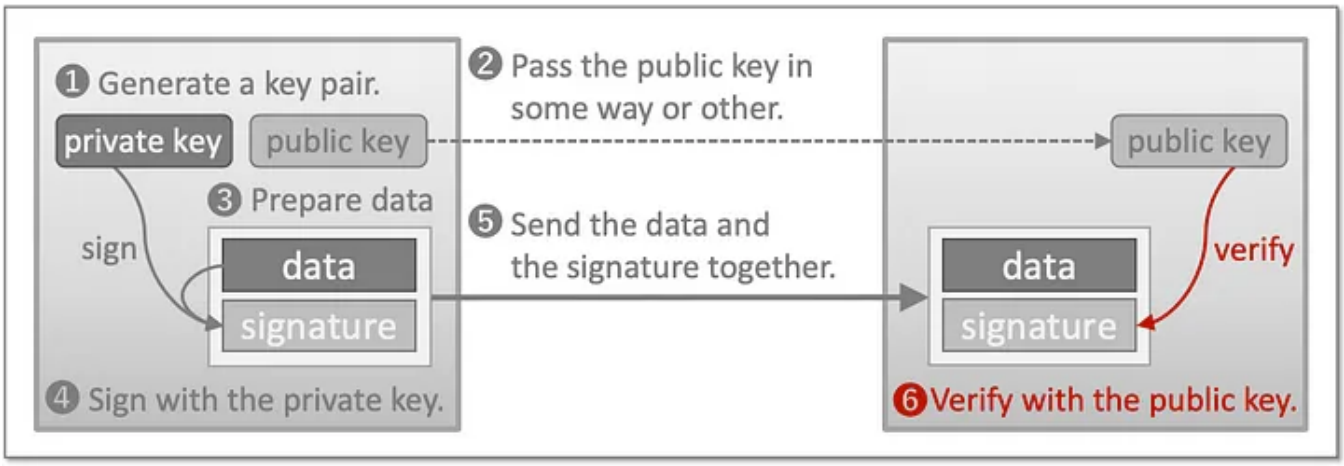
(4) Generate signature for the data with the private key.



(5) Send the data and the signature together.



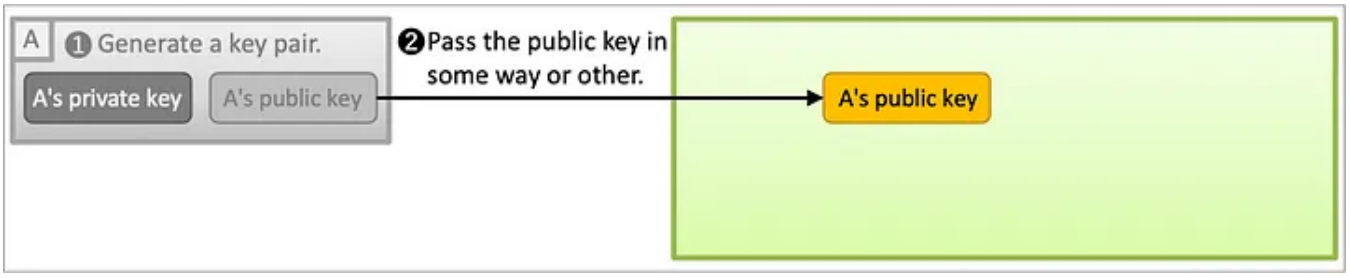
(6) Verify the signature with the public key.



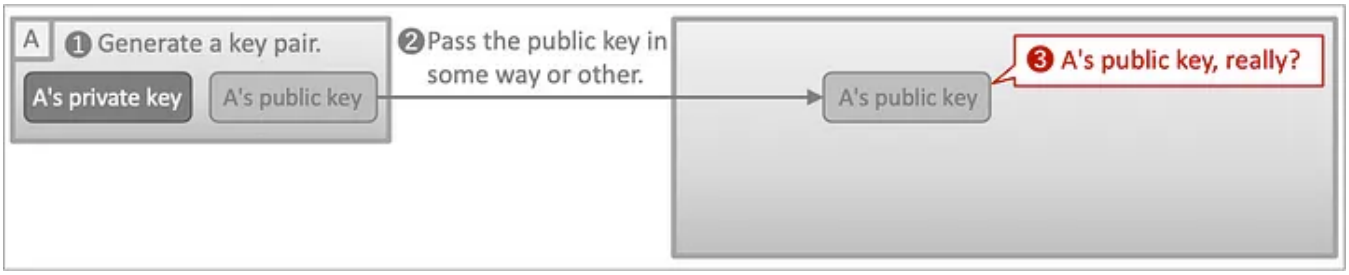
2. Certificate Chain

The first step to utilize public-key cryptography is to generate a pair of private key and public key.

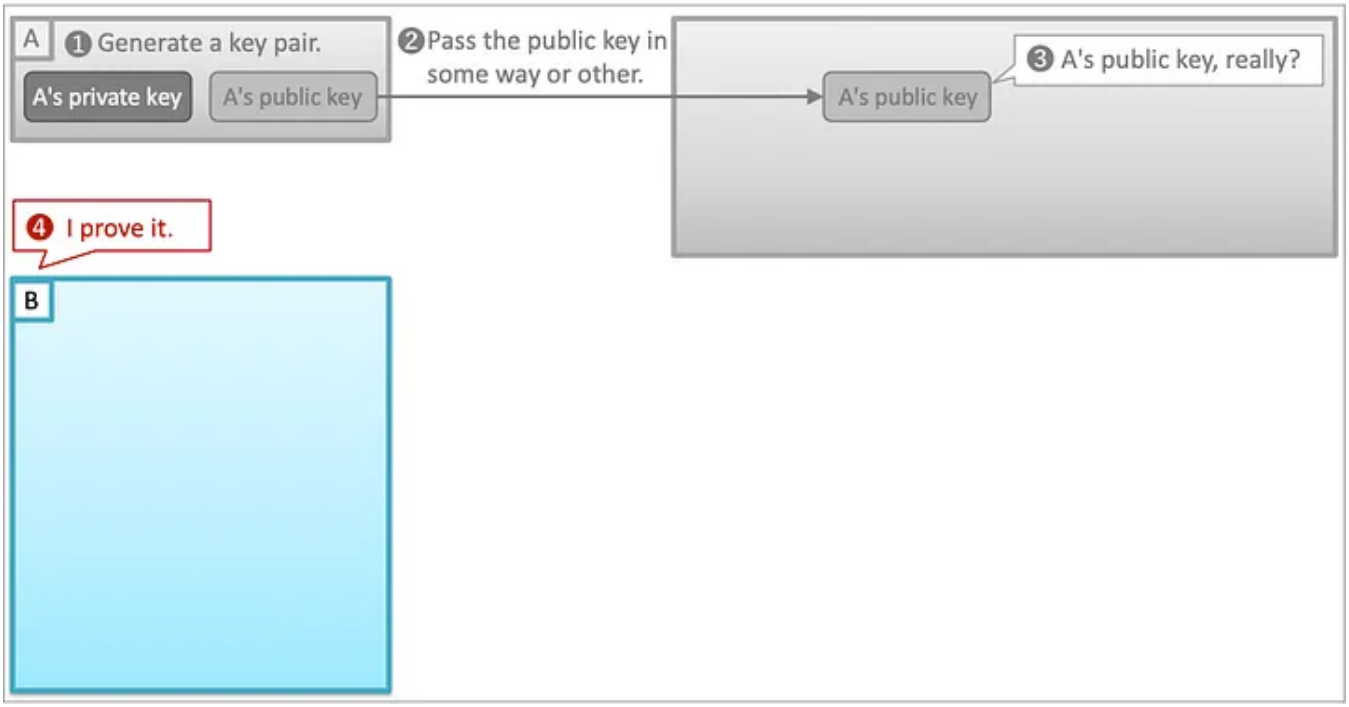
The public key needs to be passed to the other party in some way or other.



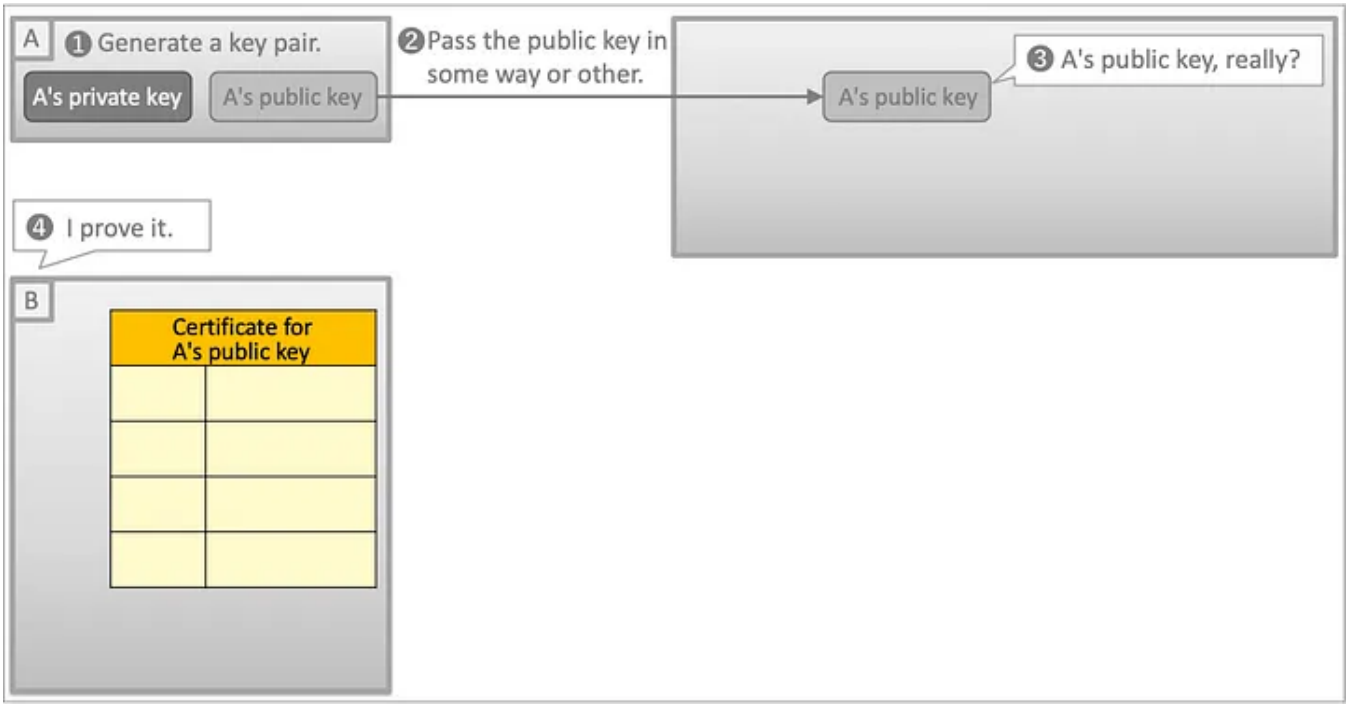
It depends on how the public key has been passed, but what if the other party cannot be sure that the public key is the legitimate one?



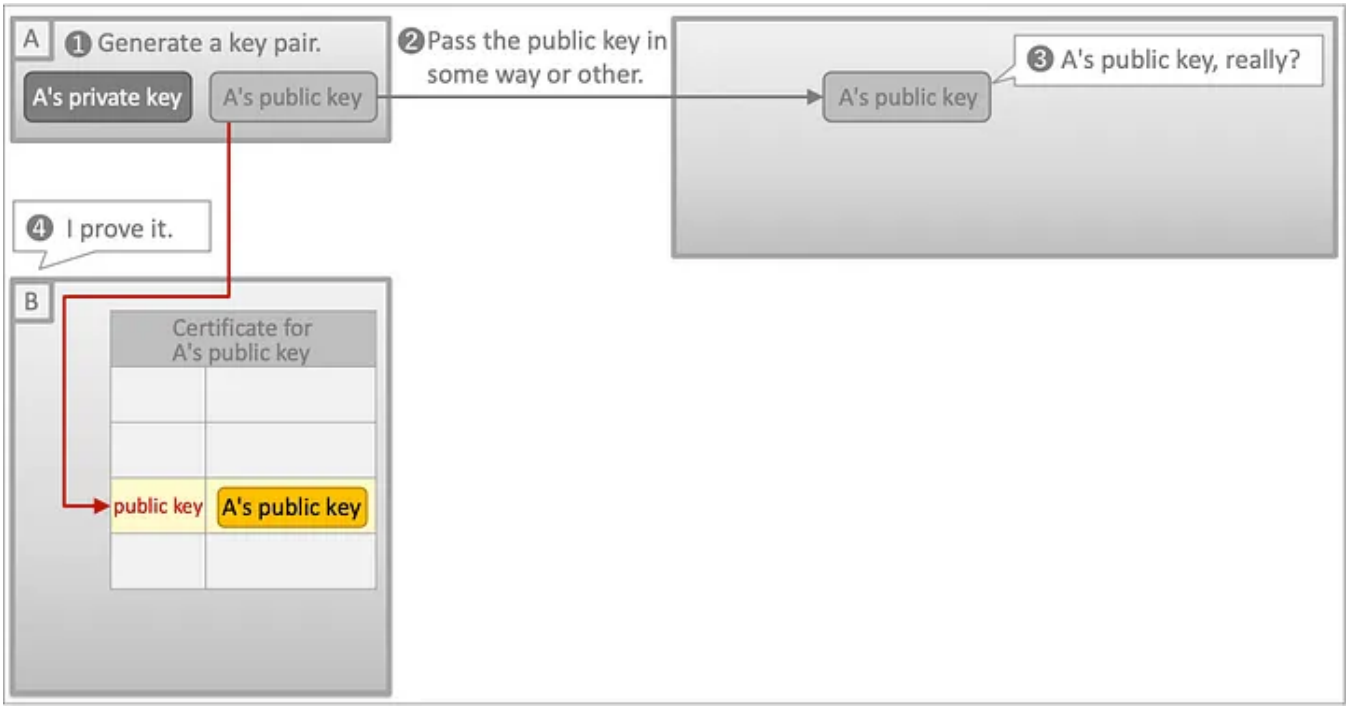
A third party may say she can prove it.



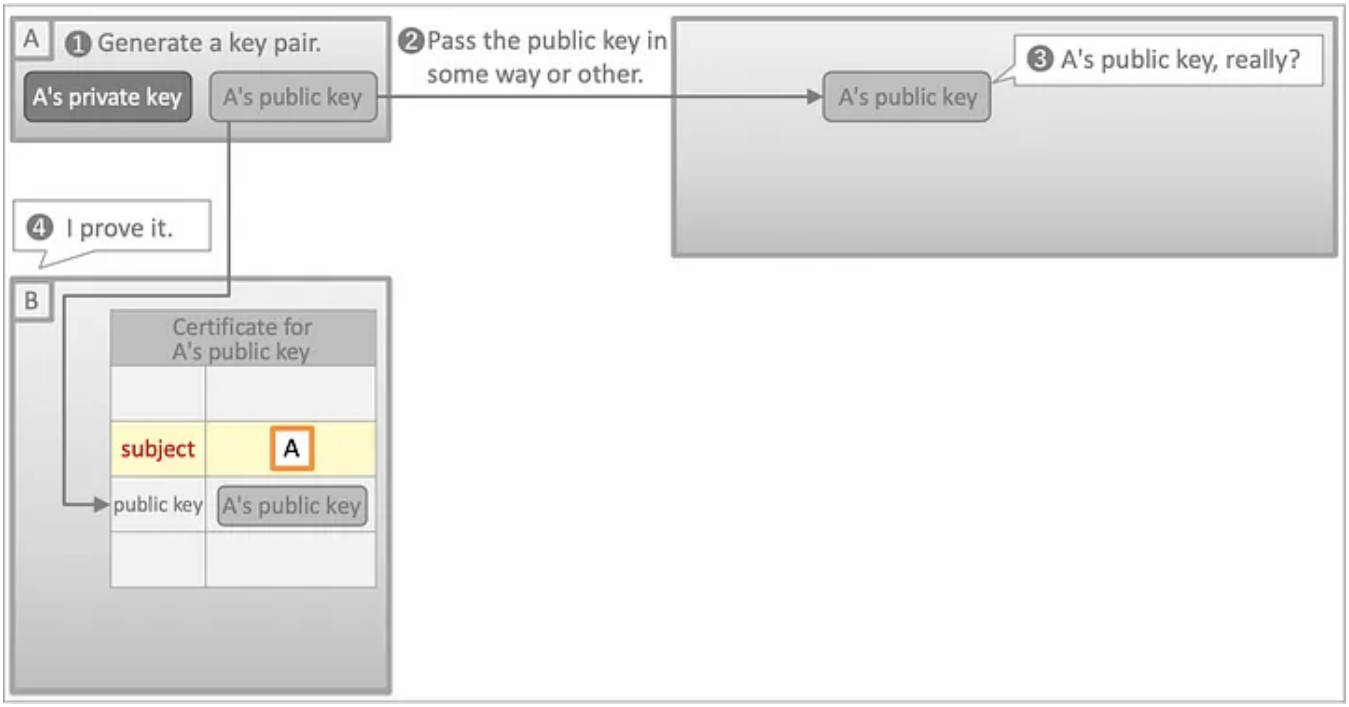
She starts to prepare a document which proves the authenticity of the public key, namely, a **certificate** for the public key.



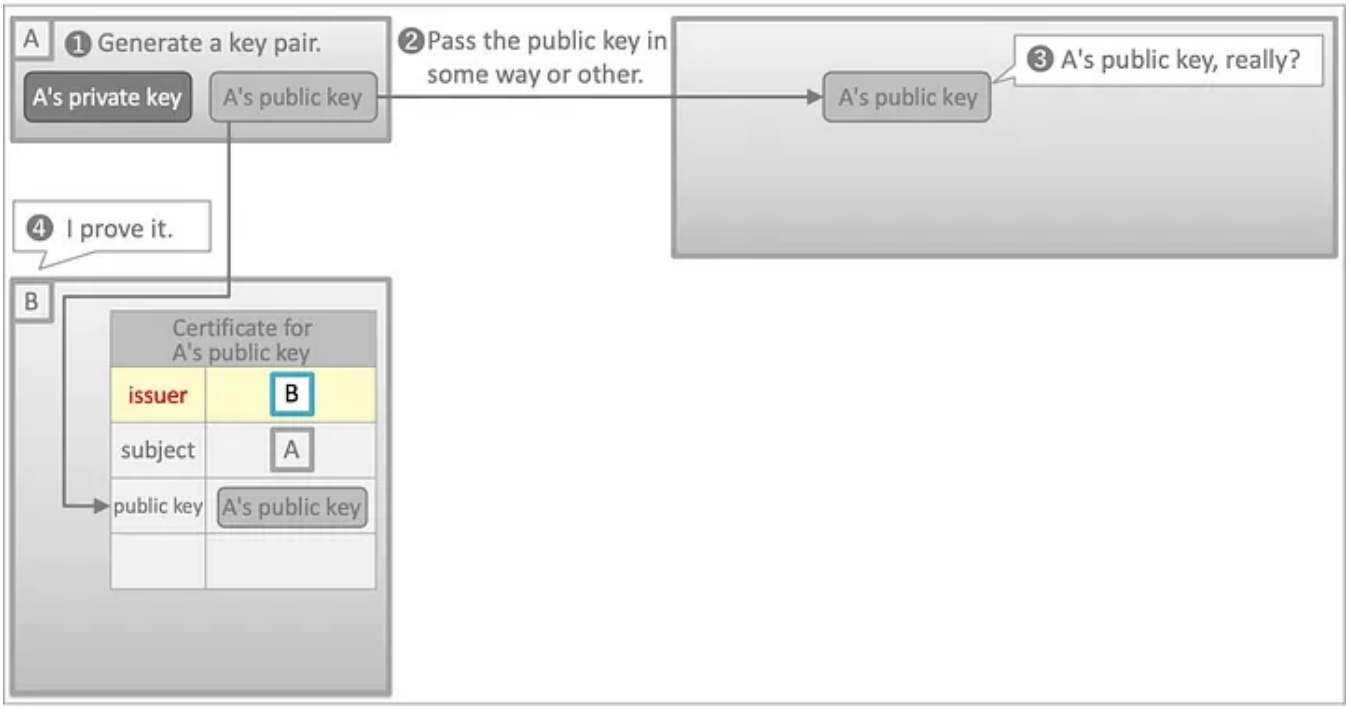
To begin with, the public key, which is the target of assurance, is put on the certificate.



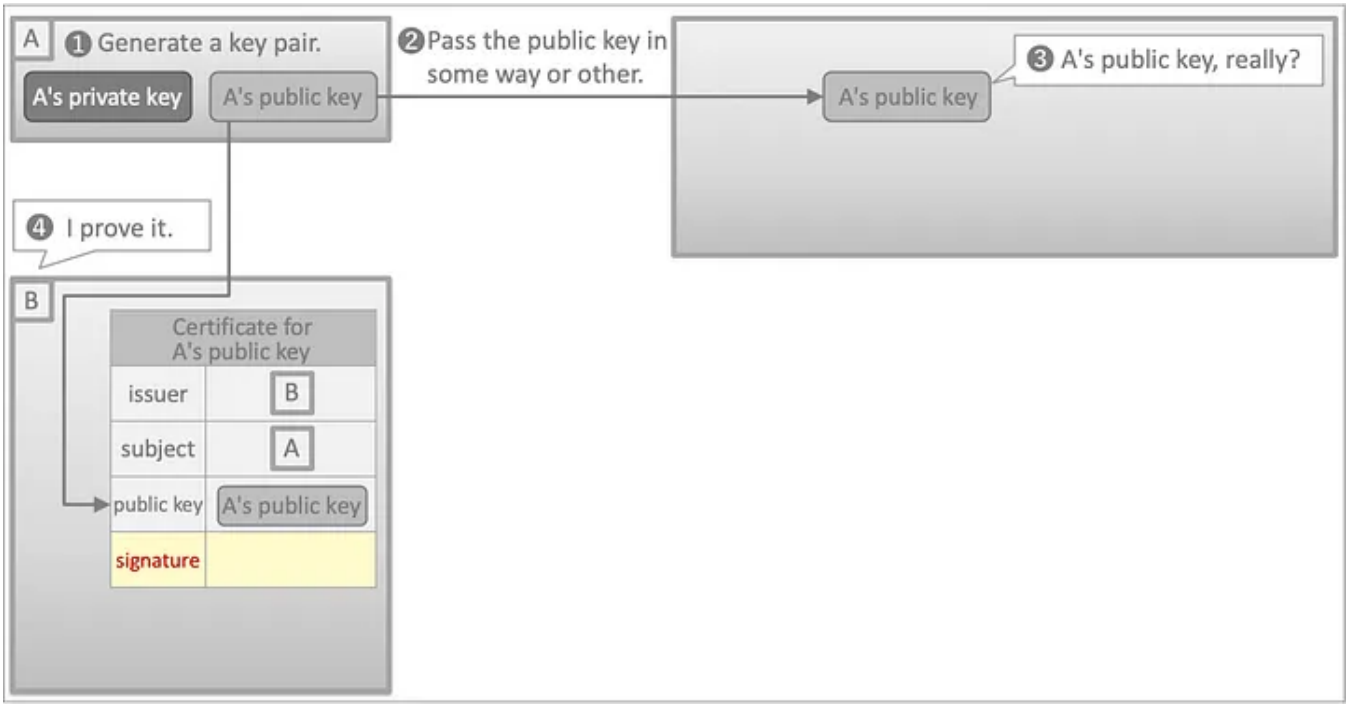
Next, information about the party that has the private key (hereinafter, **subject**) is put.



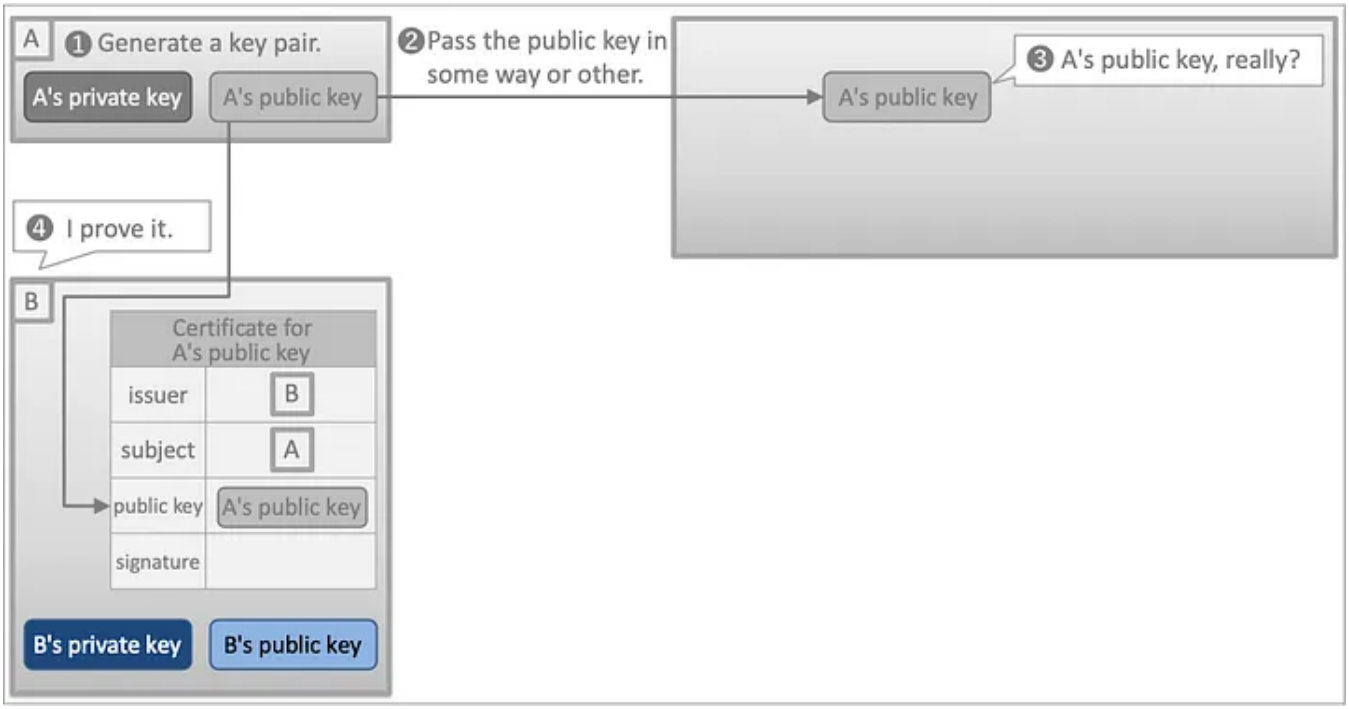
Then, information about the **issuer** of the certificate (herself) is put.



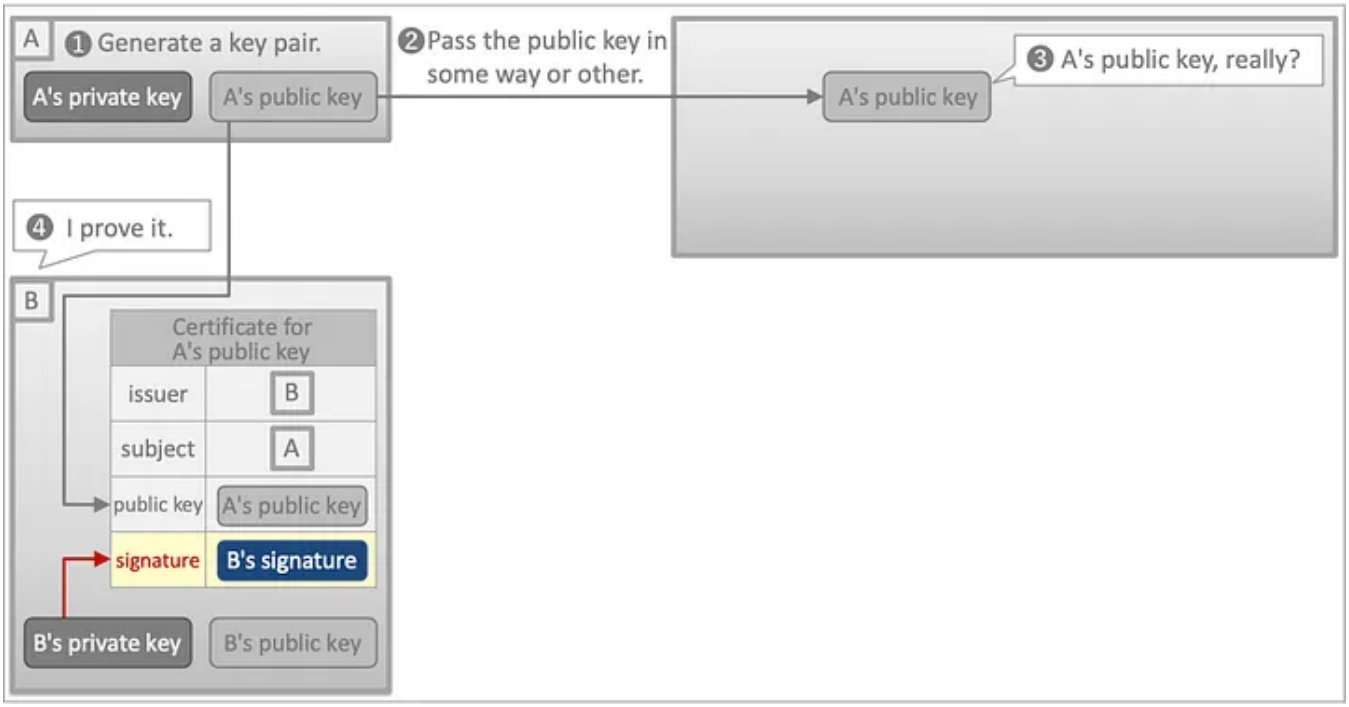
Finally, she wants to attach digital signature in order to assure the content of the certificate.



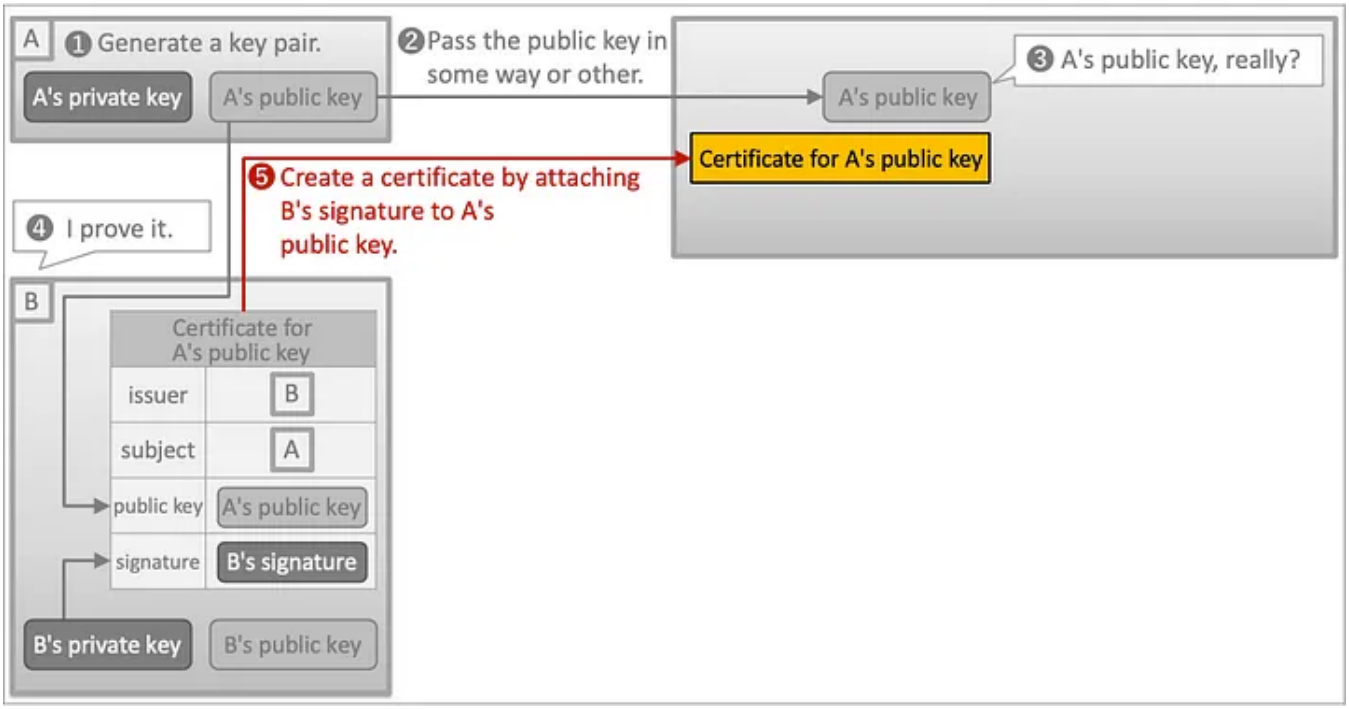
For digital signature, she generates a pair of private key and public key.



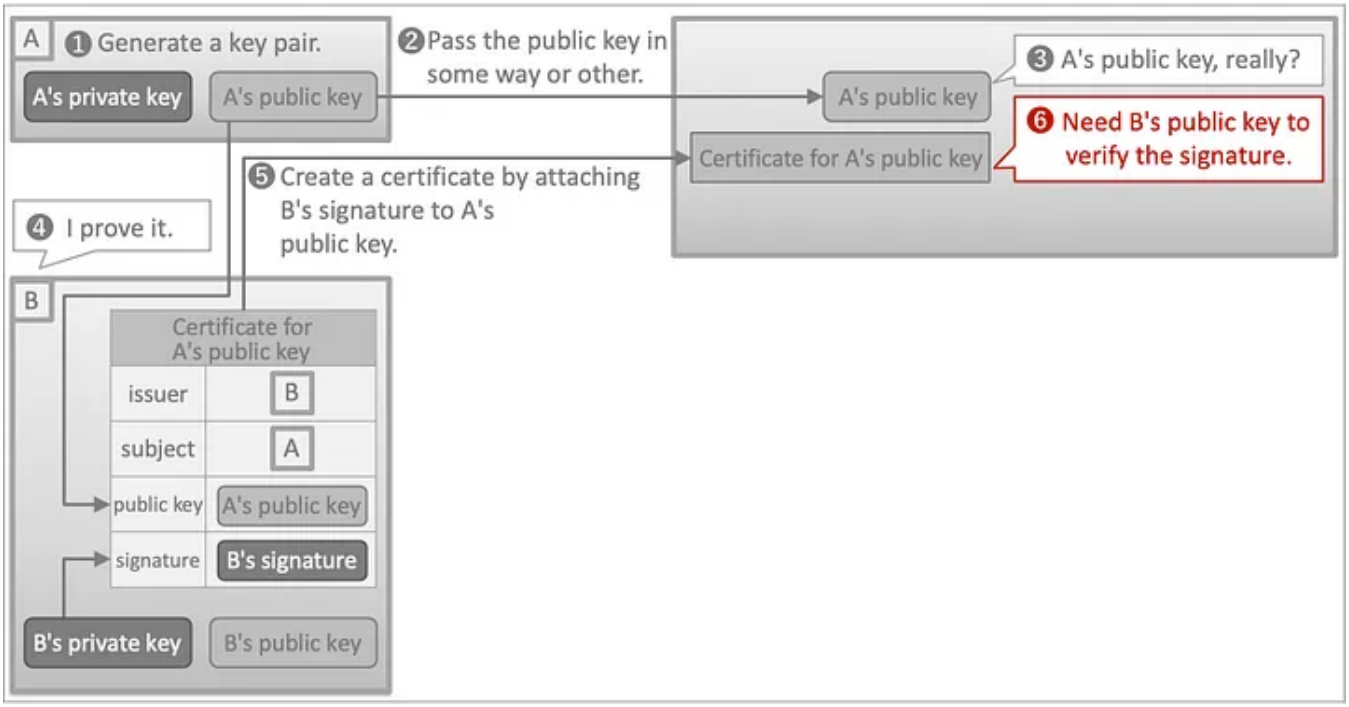
Then, she signs the certificate with the private key. This makes the certificate complete.



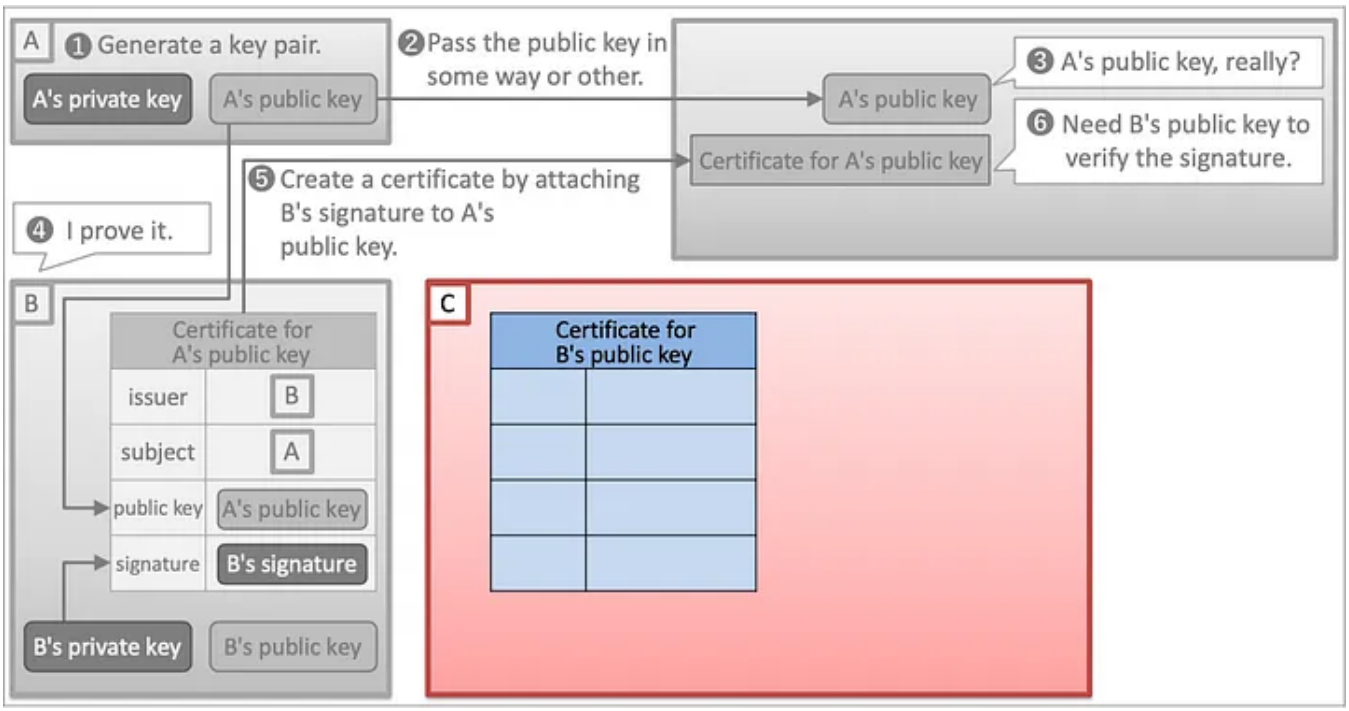
Instead of receiving the public key directly, the other party receives the certificate that includes the public key and assures the origin of the public key.



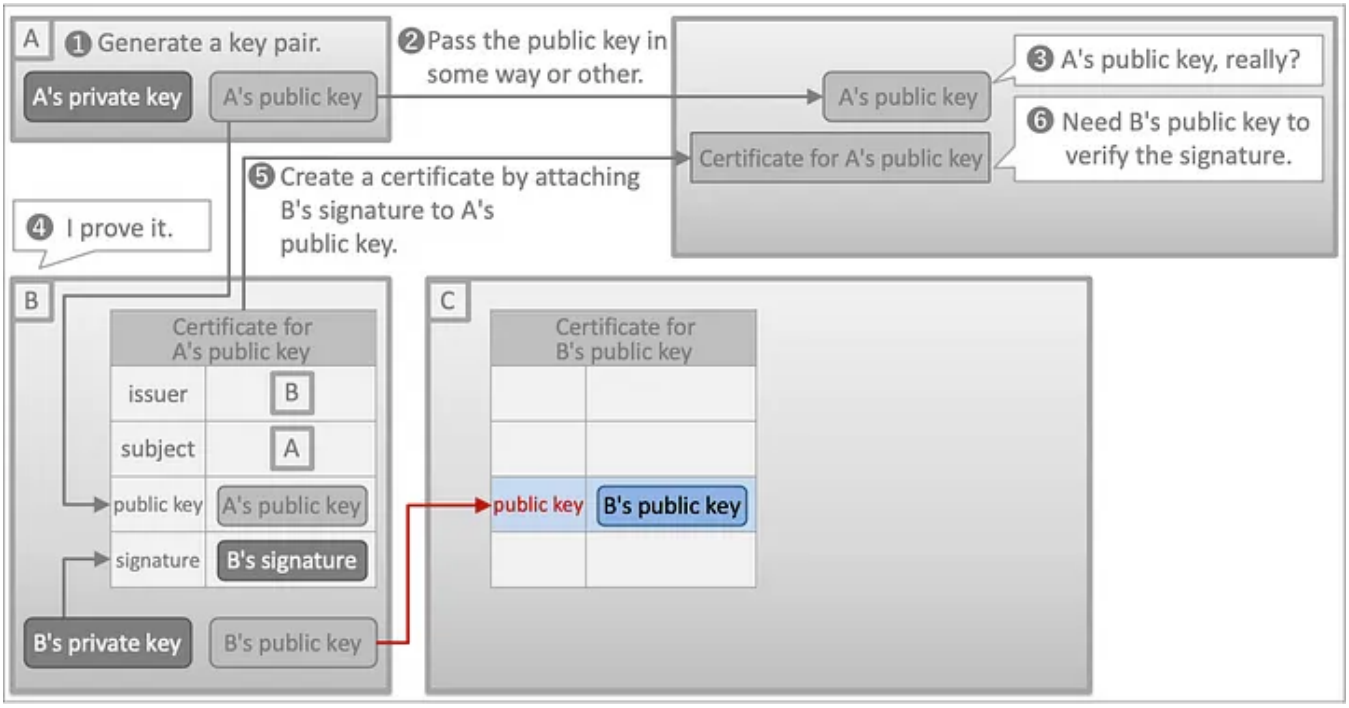
As a result, the other party could get the public key he originally wanted (“A’s public key” in the diagram). However, he realizes that he needs one more public key to verify the digital signature of the certificate (“B’s public key” in the diagram).



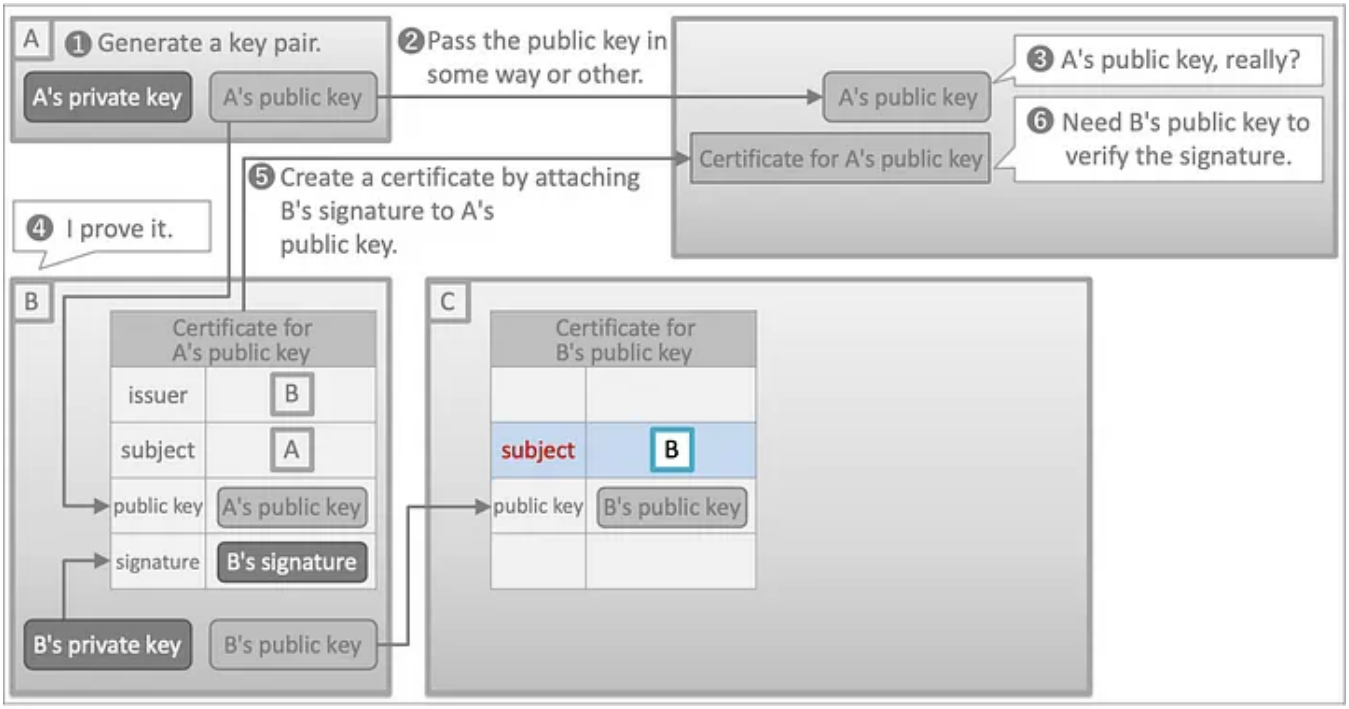
Another third party shows up and says that she provides a certificate for the public key which is necessary to verify the digital signature. She starts to prepare the certificate.



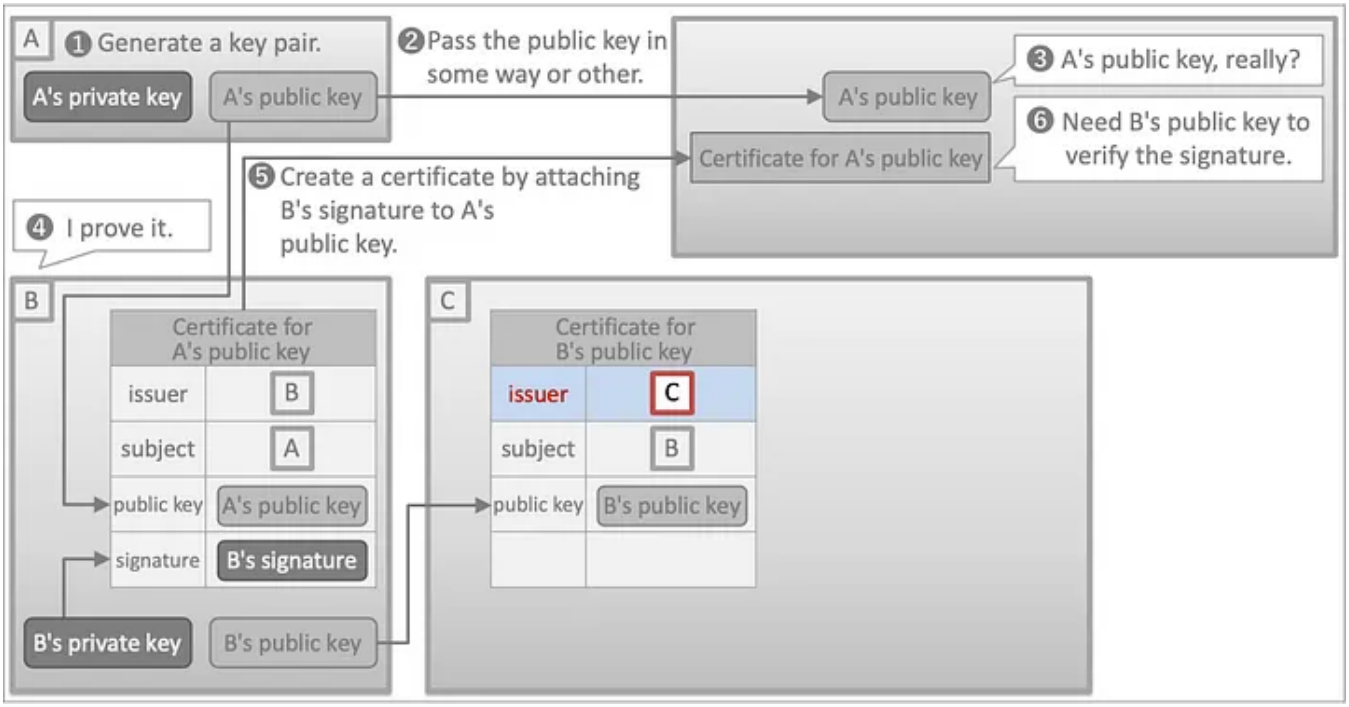
To begin with, the public key, which is the target of assurance, is put on the certificate.



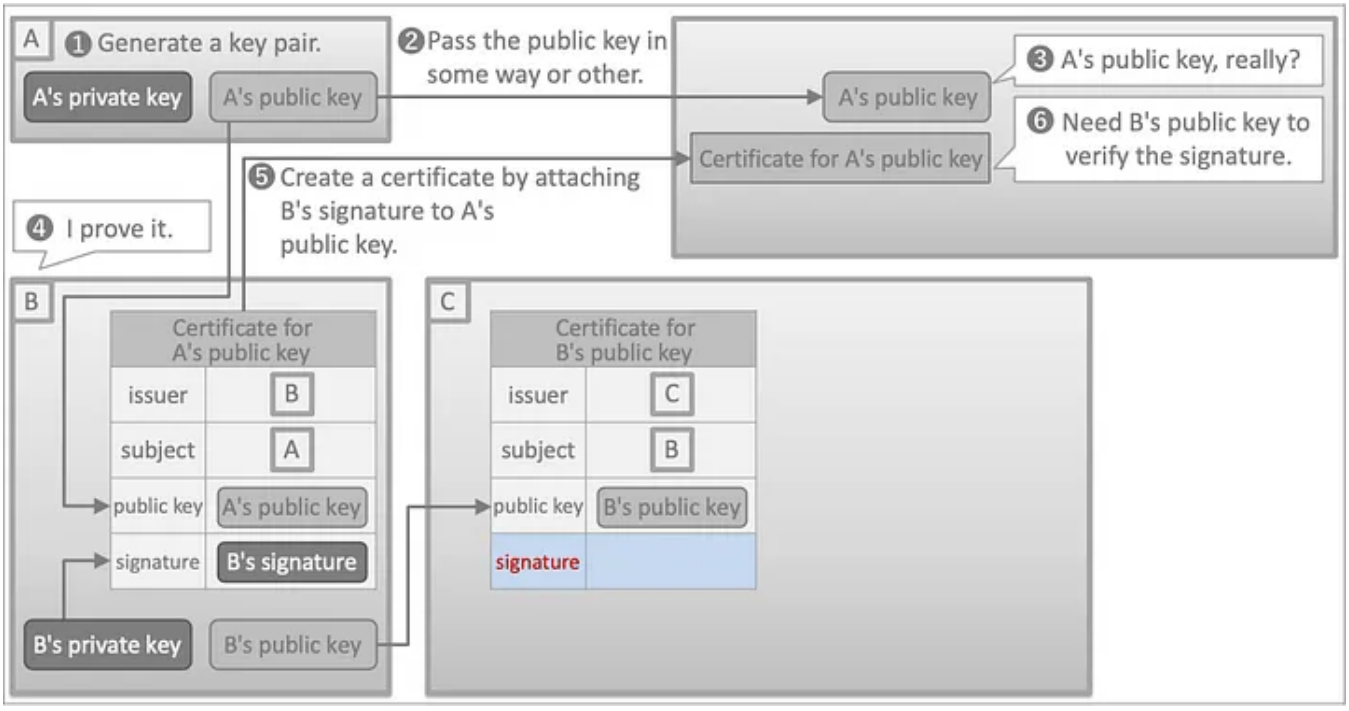
Next, information about the subject is put.



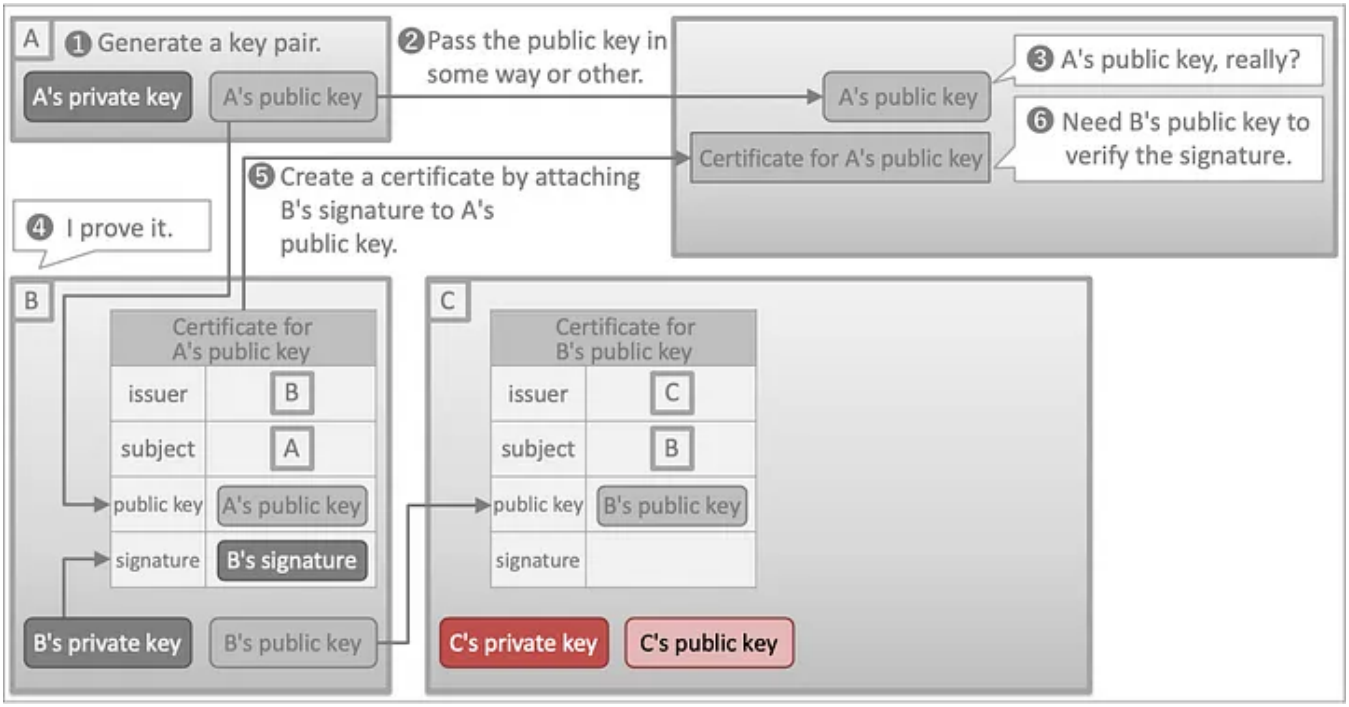
Then, information about the issuer of the certificate (herself) is put.



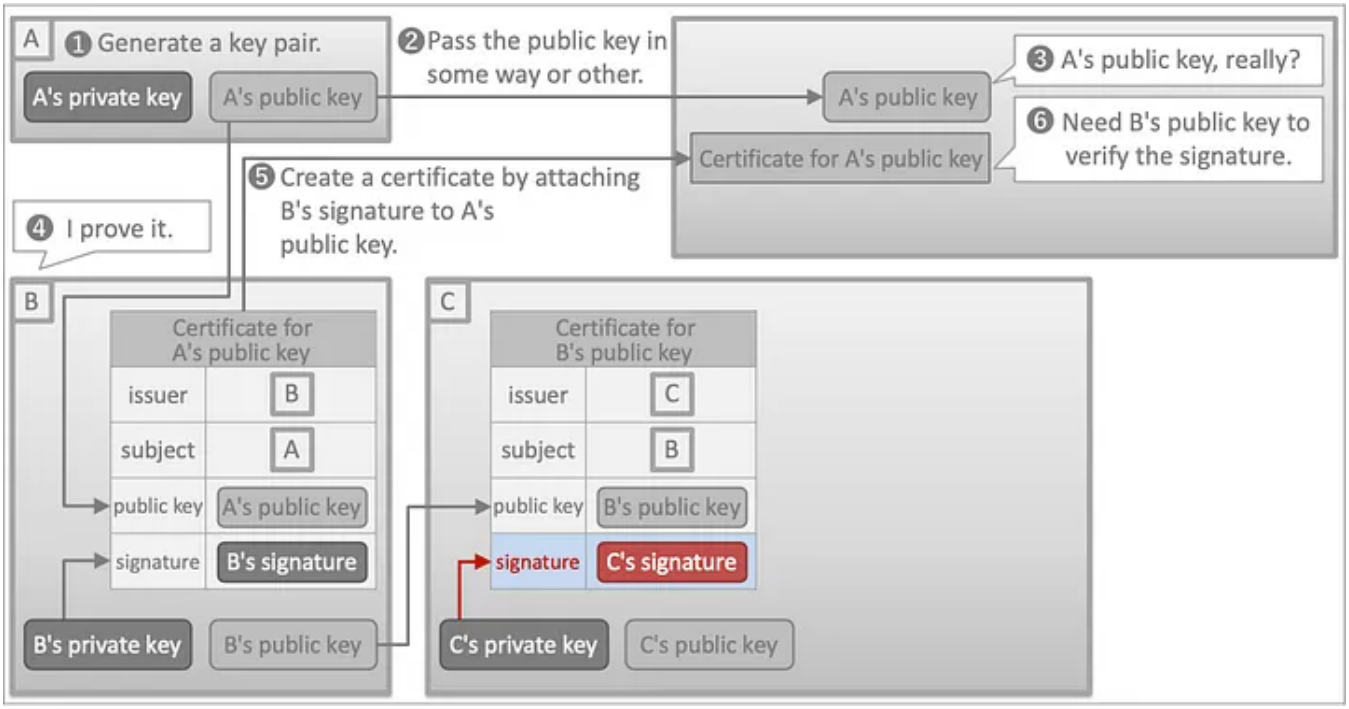
Finally, she wants to attach digital signature in order to assure the content of the certificate.



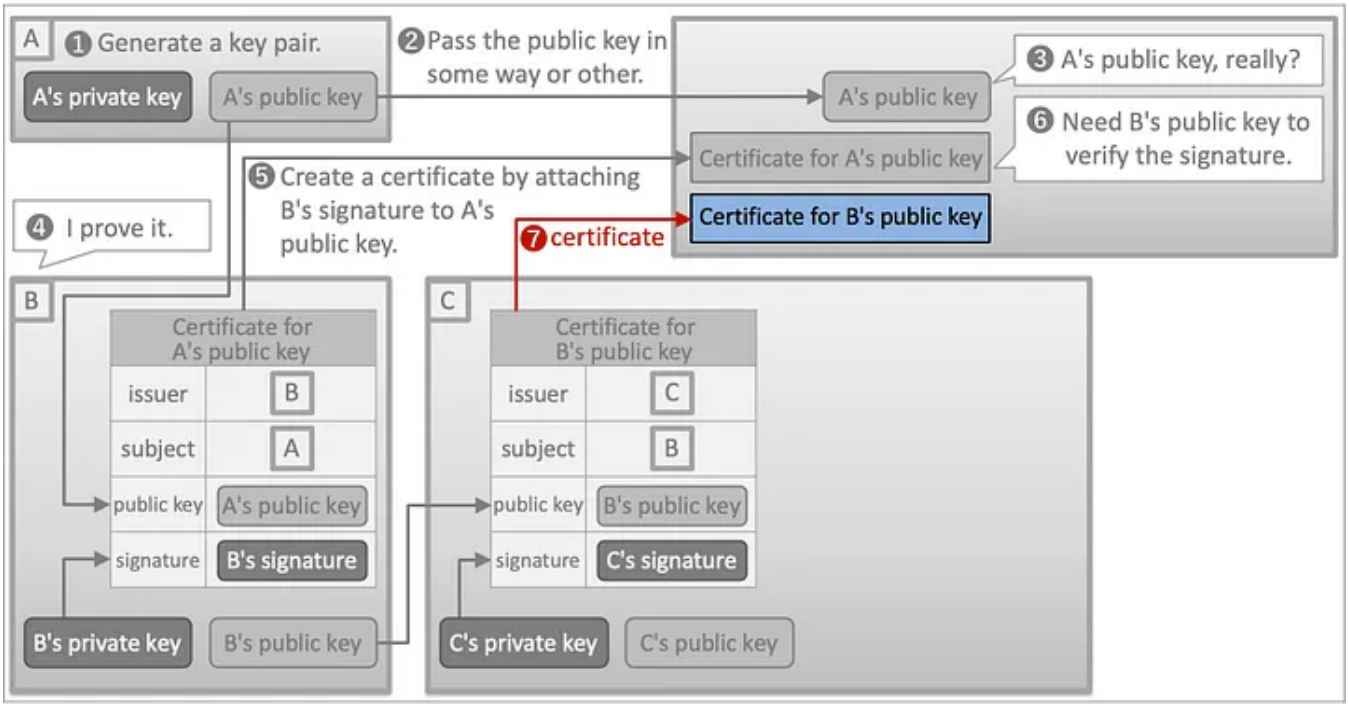
For digital signature, she generates a pair of private key and public key.



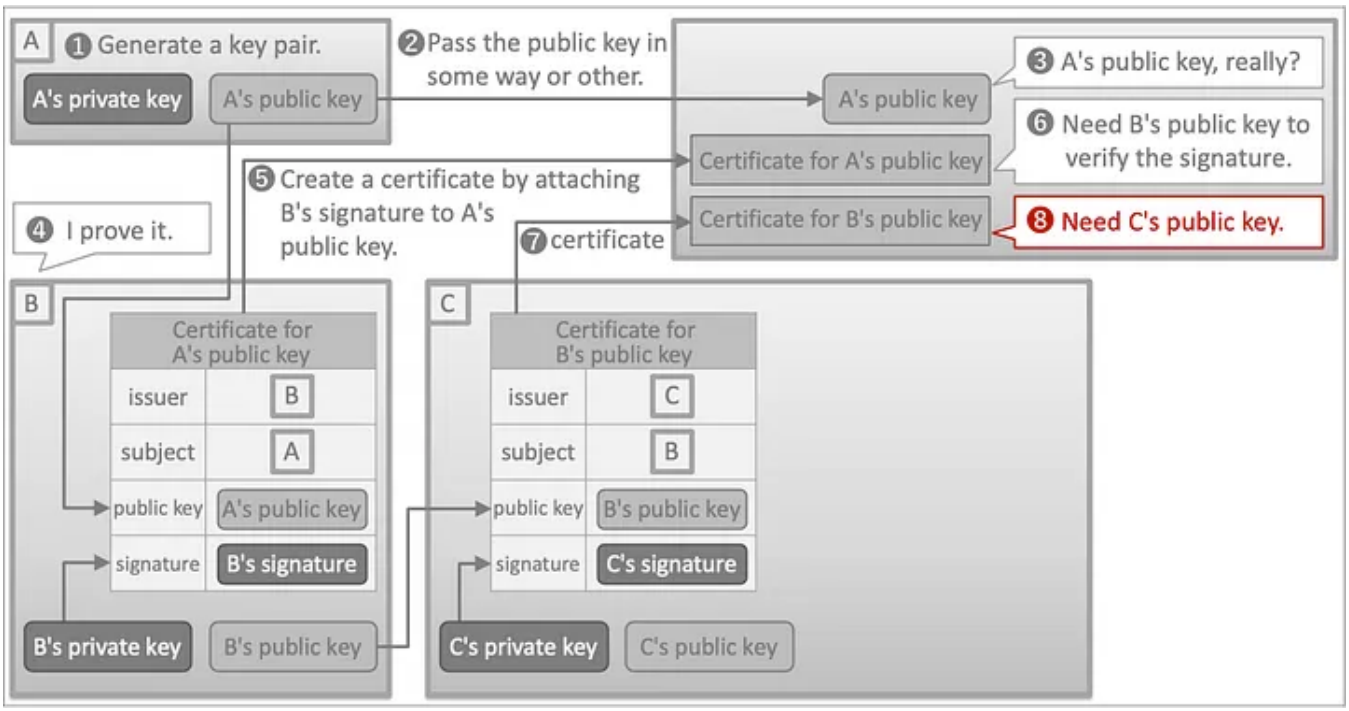
Then, she signs the certificate with the private key. This makes the certificate complete.



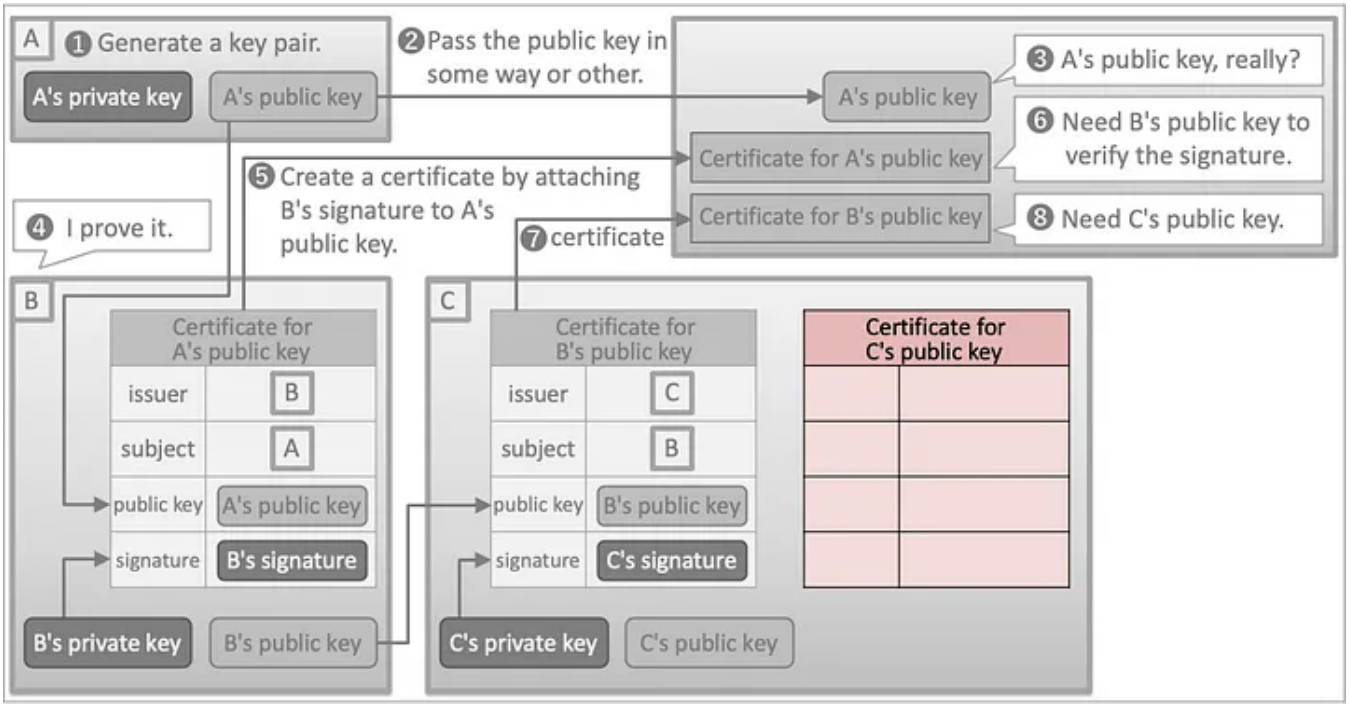
The complete certificate is sent to the other party.



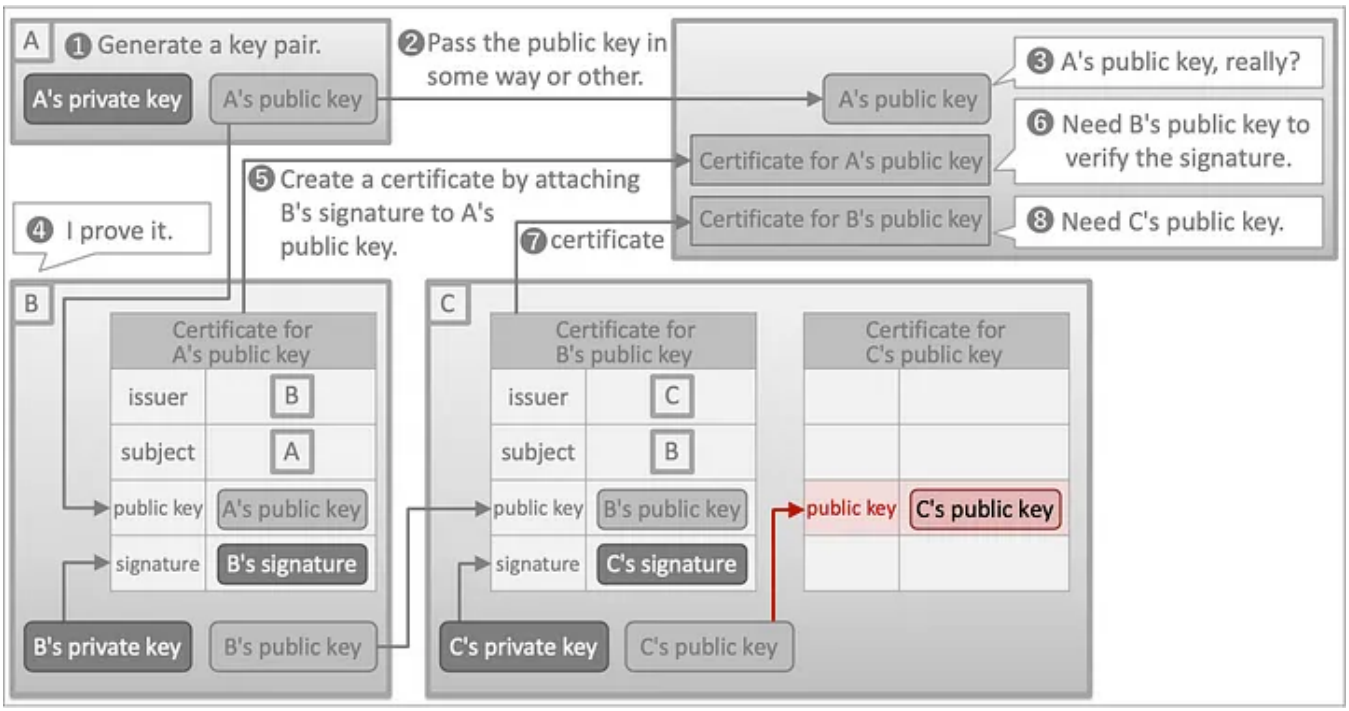
However, here we encounter the same problem as we saw previously. The other party needs one more public key (“C’s public key” in the diagram) to verify the digital signature of the certificate he just received.



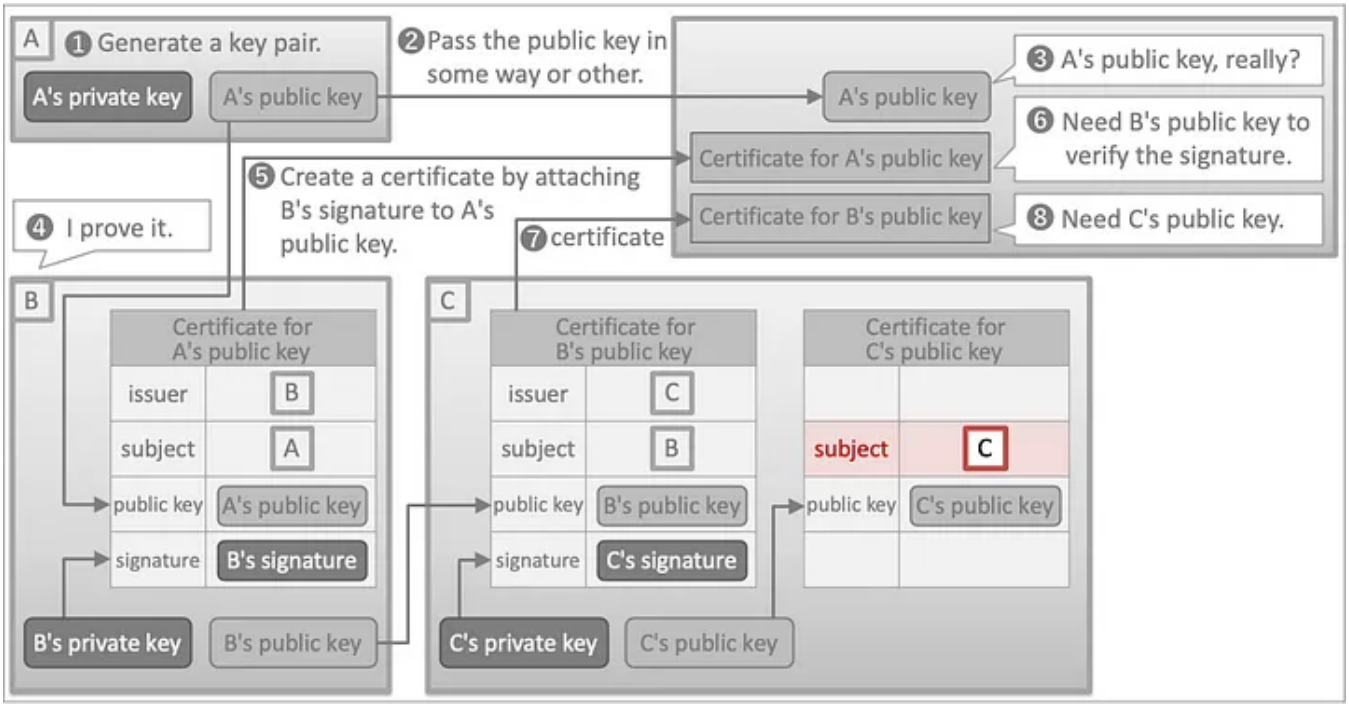
It sounds like endless, but this time, the subject of the public key herself (“C” in the diagram) says that she provides a certificate for the public key and starts to prepare the certificate.



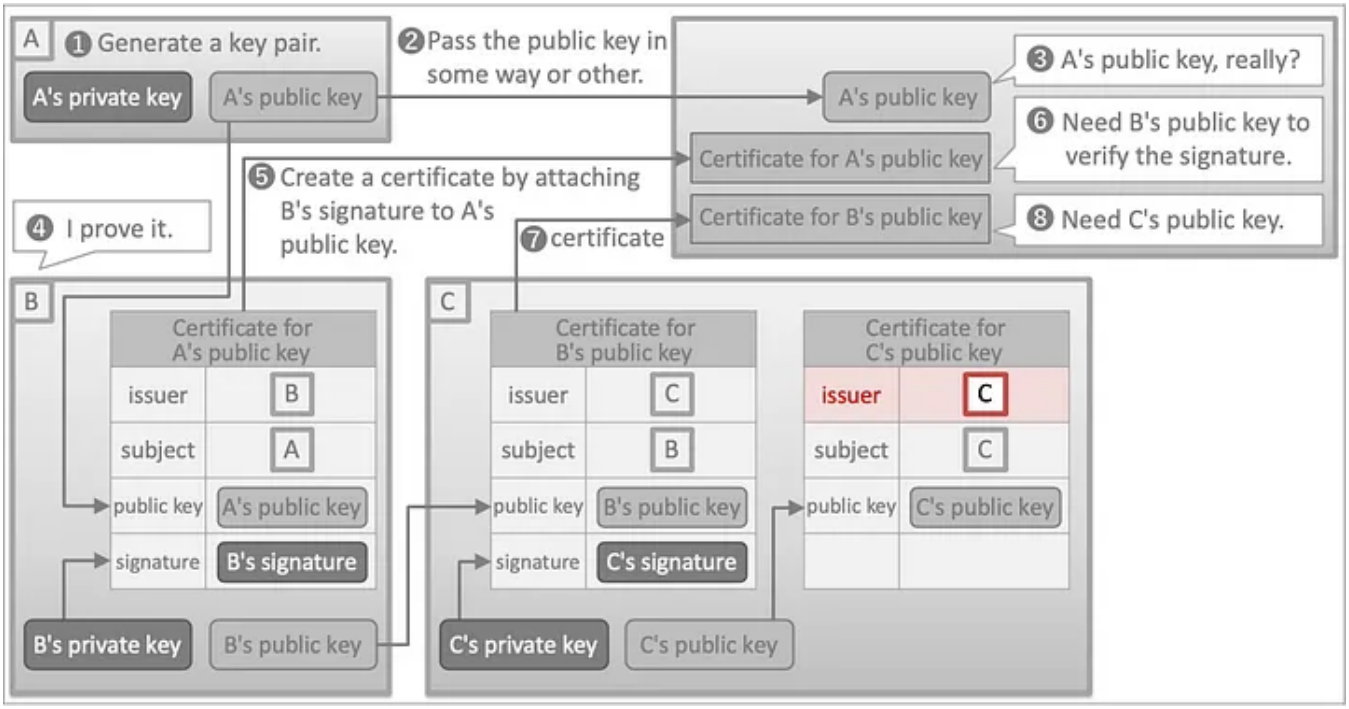
The target public key is put on the certificate,



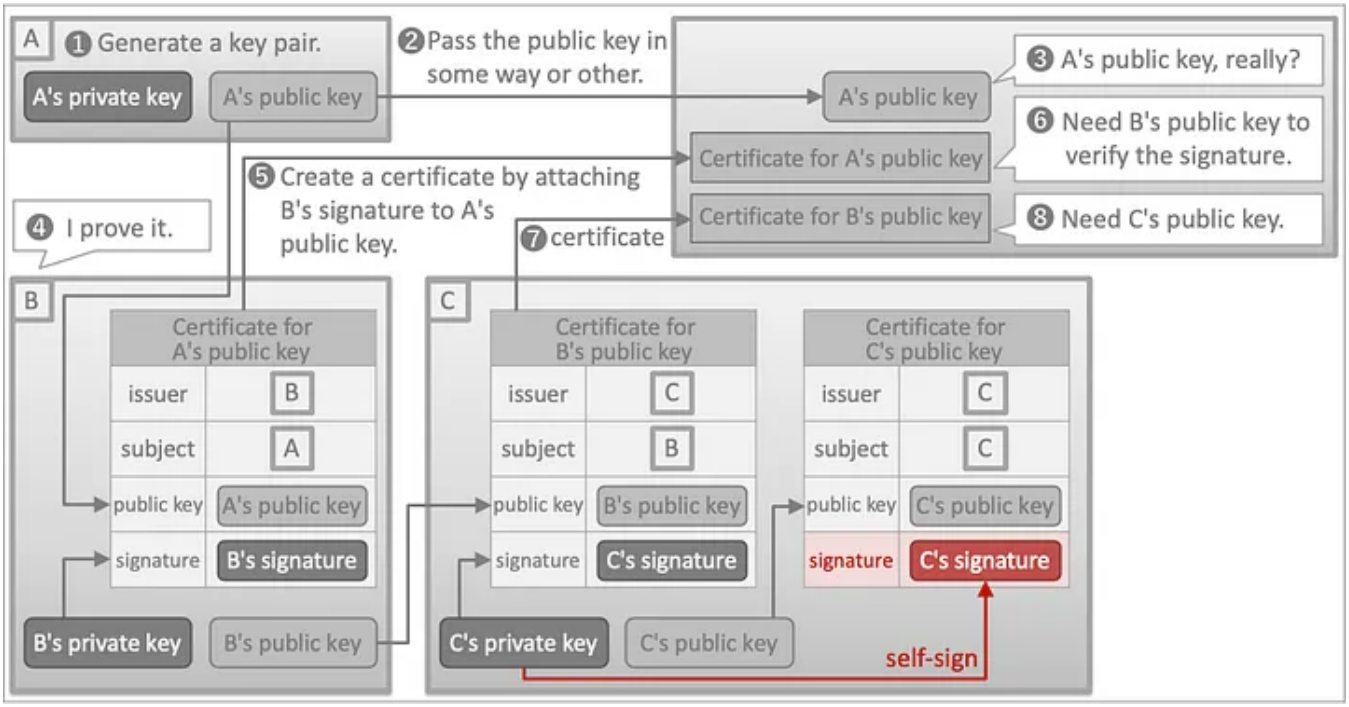
then information about the subject,



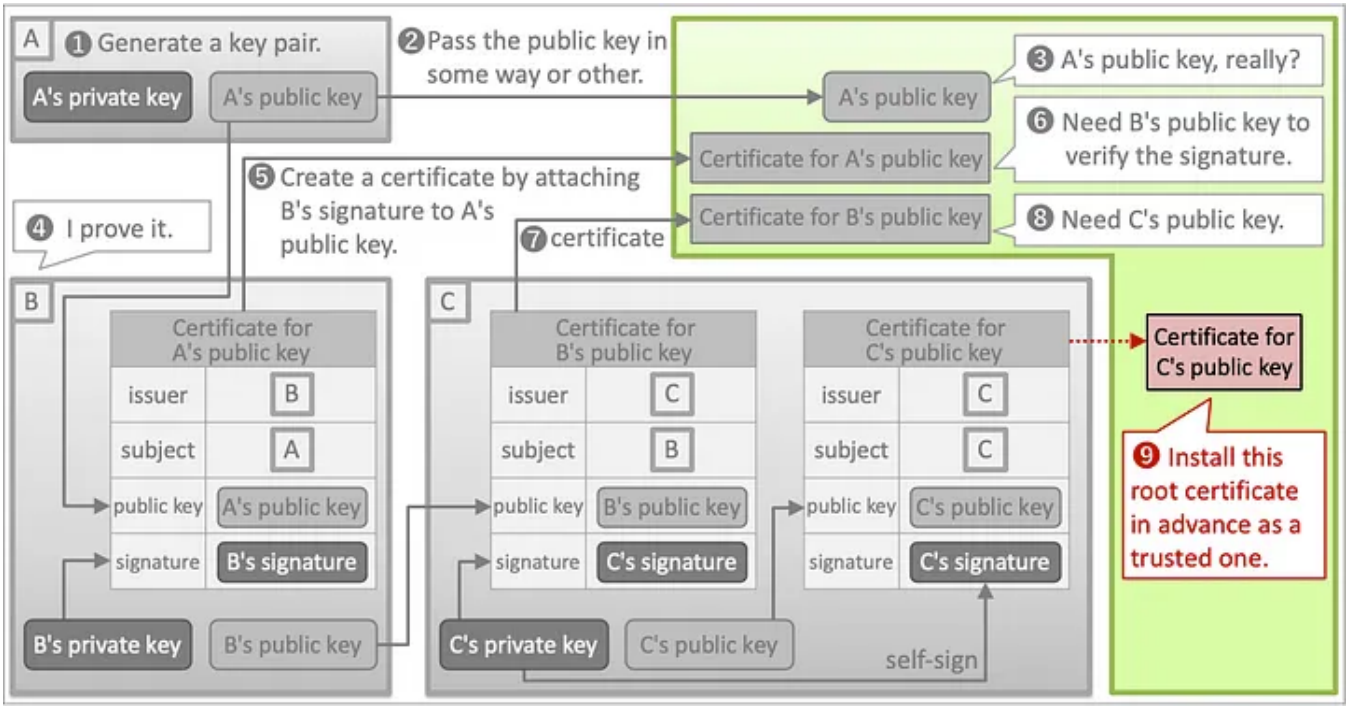
and information about the issuer (herself).



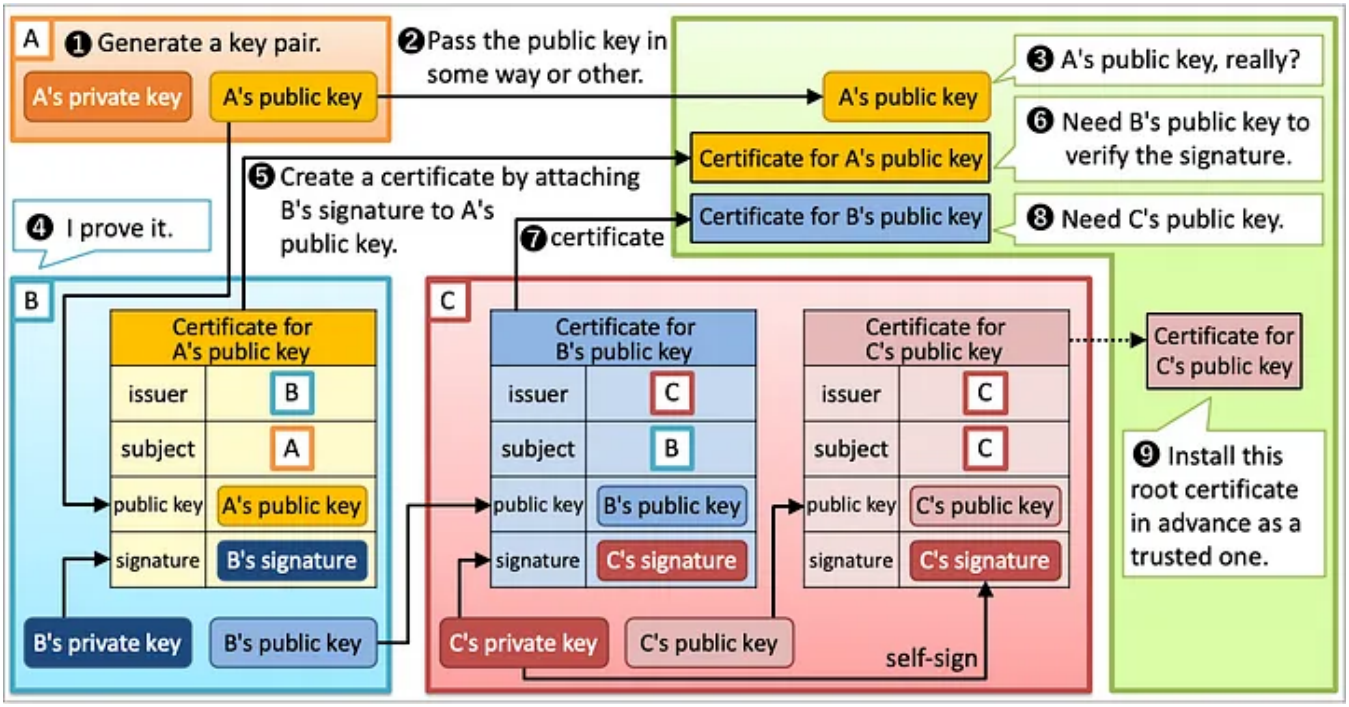
The final step is to sign the certificate. The point is that the private key used for the digital signature is the one that is paired with the target public key. The operation of signing the public key with the paired private key like this is called “**self-sign**”. And, a certificate made by self-signing is called “**self-signed certificate**”. The issuer and the subject of a self-signed certificate are identical.



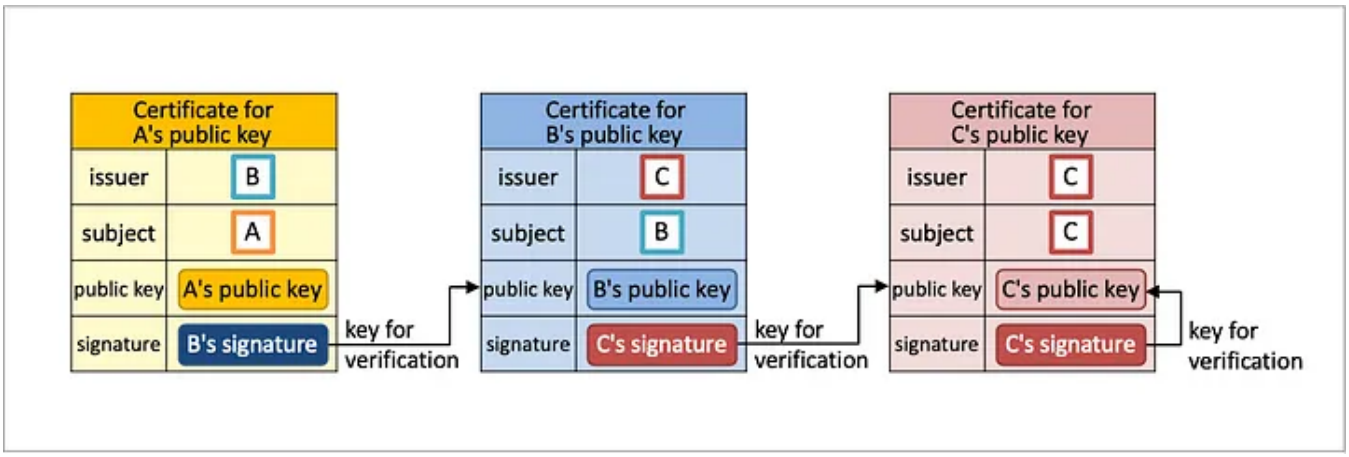
As the self-signed certificate is the origin, it has to be installed in advance as a trusted certificate.



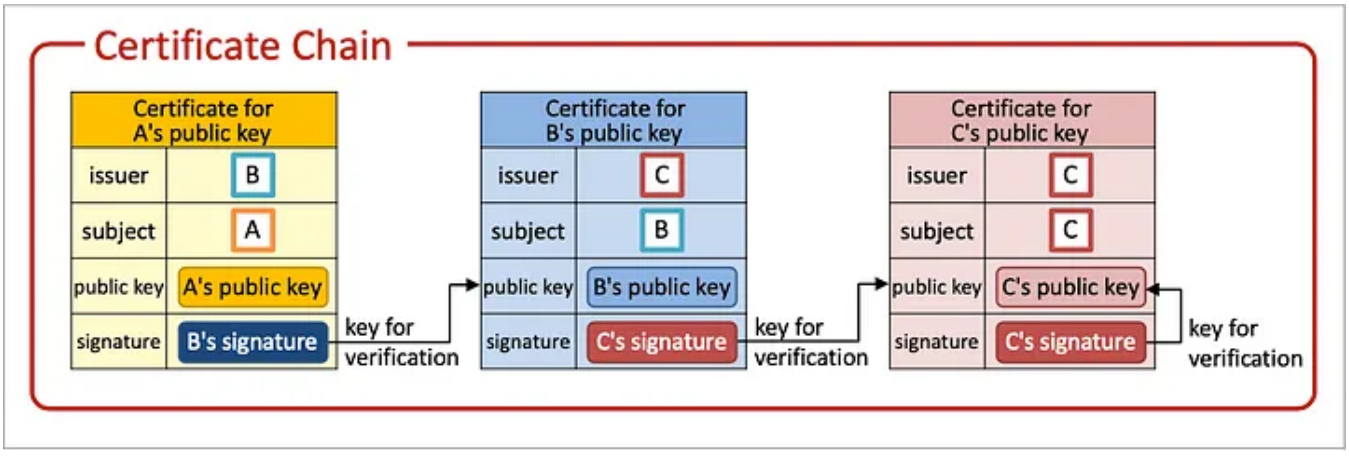
The following is the entire diagram.



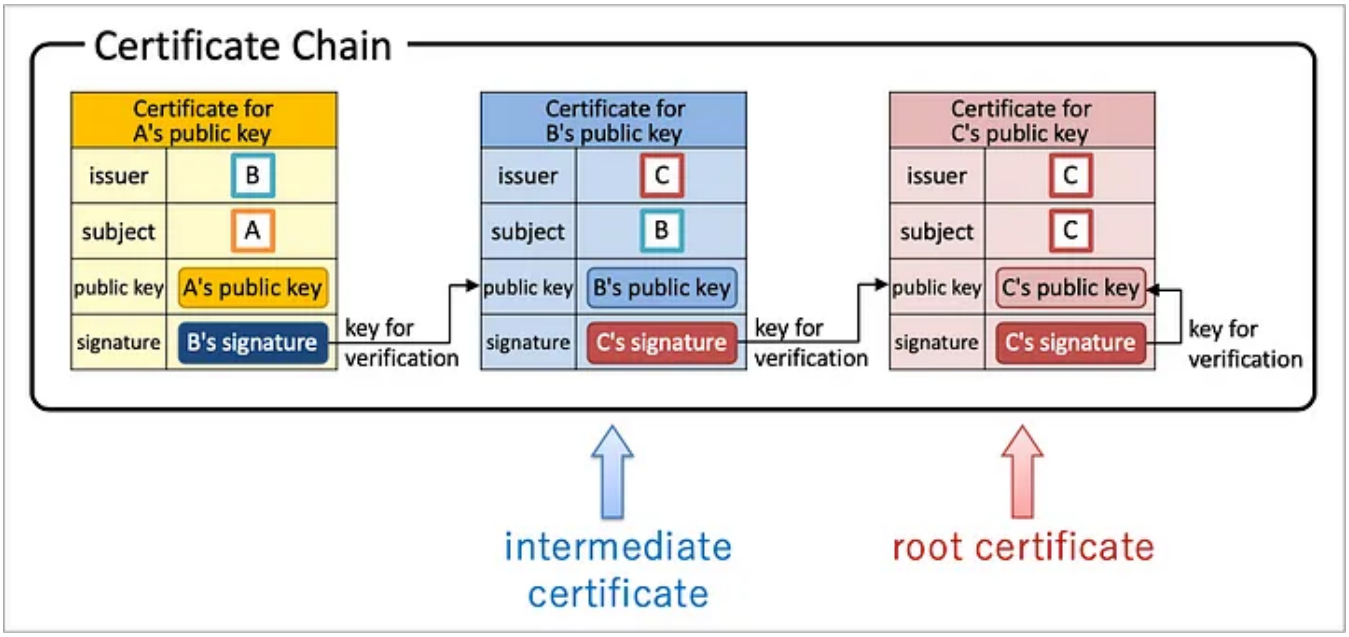
If we focus on certificates in the entire diagram,



we find that certificates are connected like a chain. This chain is called "**certificate chain**".



The self-signed certificate which is the origin of the chain is called "**root certificate**", and the in-between certificate is called "**intermediate certificate**". The number of intermediate certificates can be 2 or more.



3. Certificate Structure

The structure of certificates is defined in RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.

The following is a part of the certificate structure excerpted from Section 4.1 of RFC 5280.

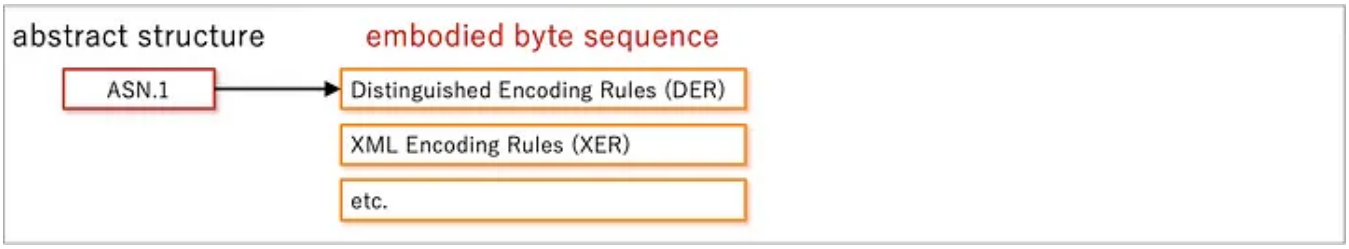
```
Certificate ::= SEQUENCE {                                  entire certificate
    tbsCertificate       TBSCertificate,                      certificate content
    signatureAlgorithm   AlgorithmIdentifier,                 signature algorithm
    signatureValue       BIT STRING  }                        signature

TBSCertificate  ::=  SEQUENCE  {                            certificate content
    version         [0]  EXPLICIT Version DEFAULT v1,         version
    serialNumber         CertificateSerialNumber,             serial number
    signature            AlgorithmIdentifier,                 signature algorithm
    issuer               Name,                                issuer
    validity             Validity,                            valid period
    subject              Name,                                subject
    subjectPublicKeyInfo SubjectPublicKeyInfo,                public key
    issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,  issuer ID
                         -- If present, version MUST be v2 or v3
    subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,  subject ID
                         -- If present, version MUST be v2 or v3
    extensions      [3]  EXPLICIT Extensions OPTIONAL         extensions
                         -- If present, version MUST be v3
    }
```

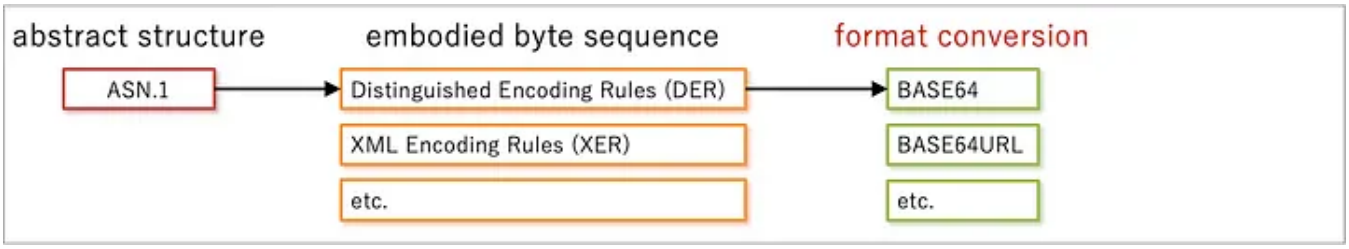
This structure definition is written in **ASN.1** (Abstract Syntax Notation One). Specifications related to the notation are defined in the X.680 series.

ASN.1 is a notation to represent data structures in an **abstract** way and it does not define how to embody abstract structures into a specific byte sequence.

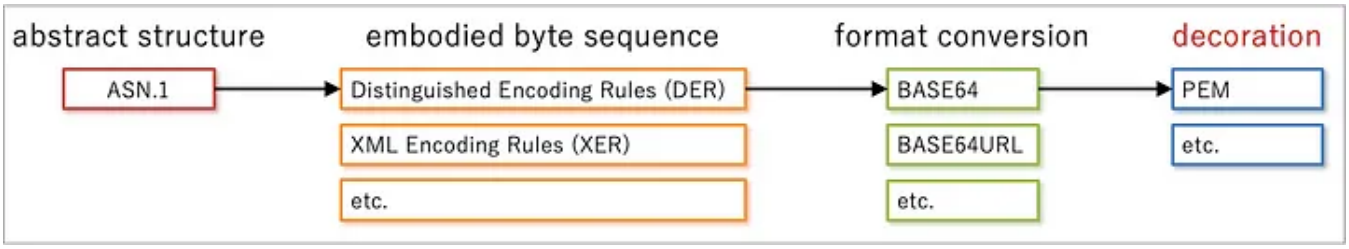
Therefore, another different specification is necessary to build a byte sequence from a data structure represented in ASN.1. For the purpose, there already exist many specifications such as **DER** (Distinguished Encoding Rules), XER (XML Encoding Rules), JER (JSON Encoding Rules) and so on.



After building a byte sequence, the format of the byte sequence may be changed. For example, because a byte sequence made based on DER becomes binary data, **BASE64** (RFC 4648) may be applied to convert the binary data into text data.

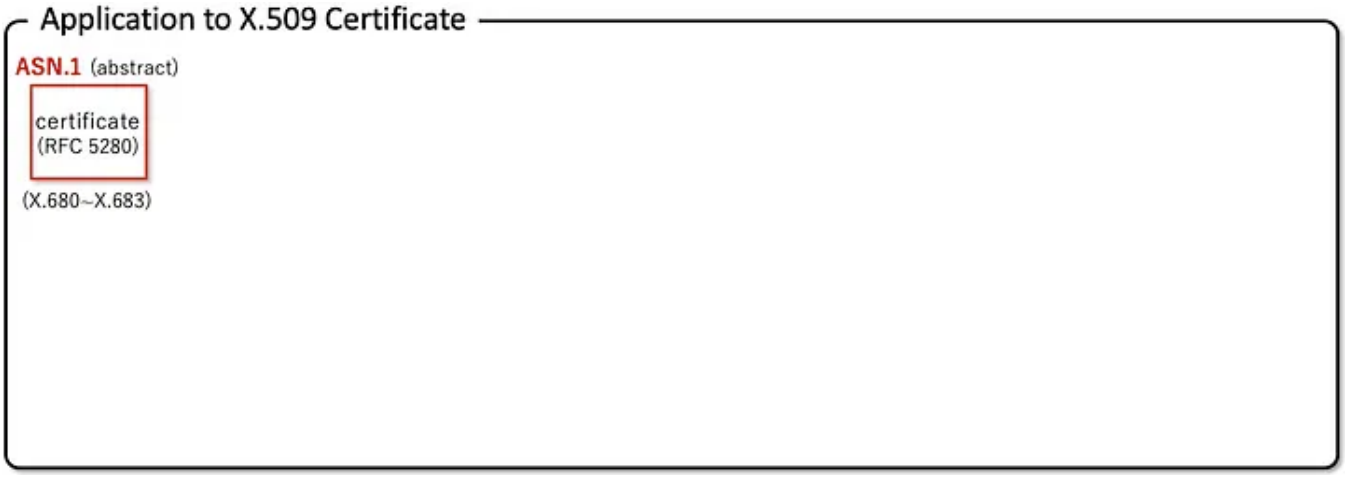


In some cases, data may be decorated. For example, it is possible to add information about what BASE64 data represents by using the rule called **PEM** (RFC 7468). For example, information telling that BASE64 data represents an X.509 certificate can be added.

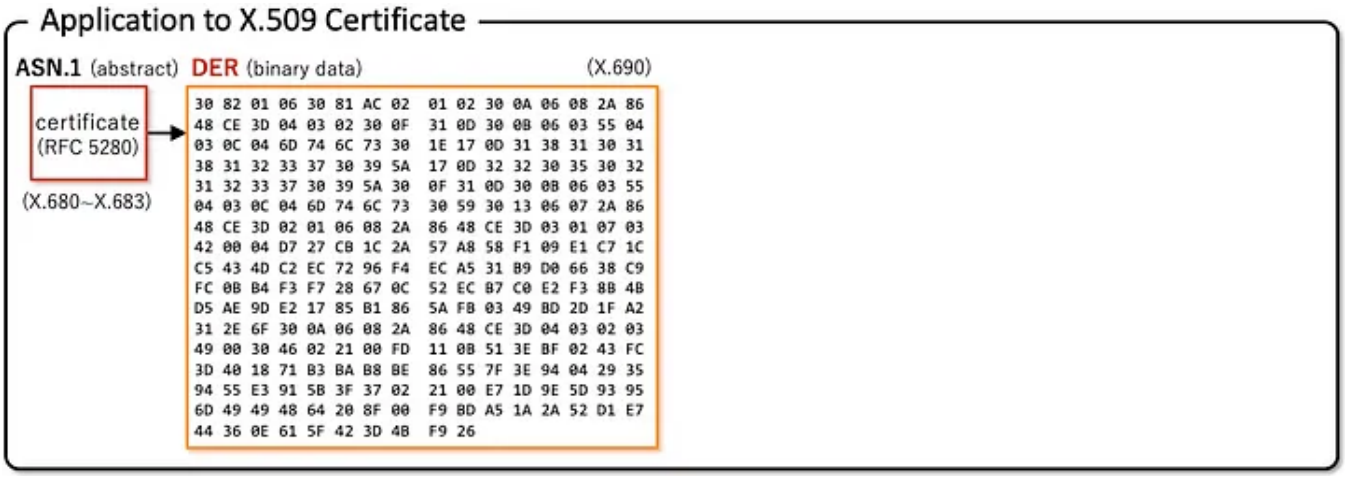


Let's apply ASN.1, DER, BASE64 and PEM mentioned above to the case of X.509 certificate.

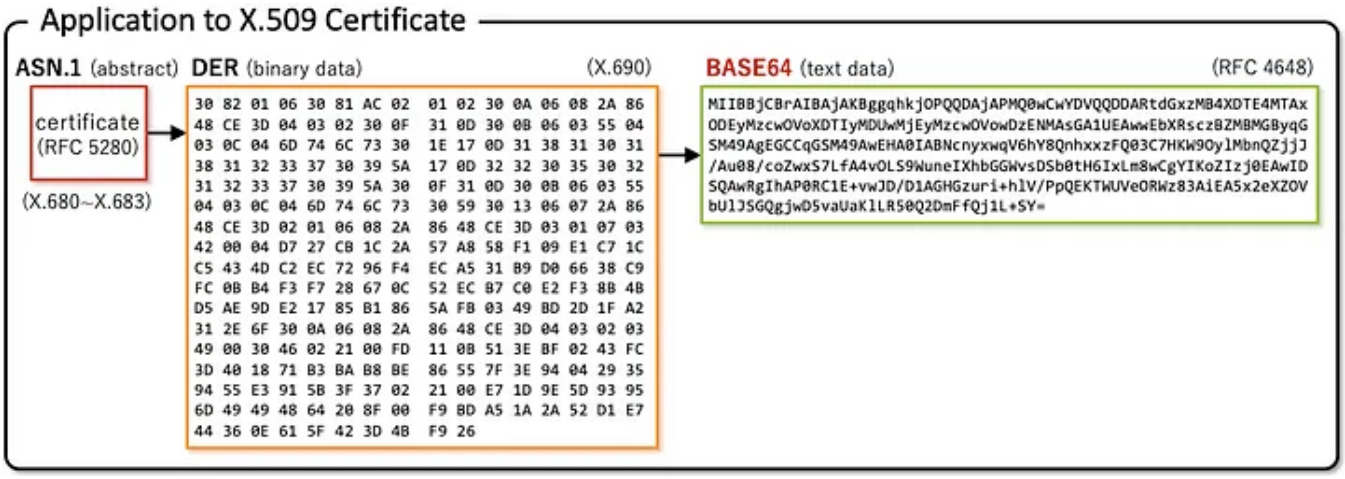
To begin with, prepare information that a certificate should have (e.g. information about the subject). Fields that a certificate should/may have are described in RFC 5280 in the form of ASN.1.



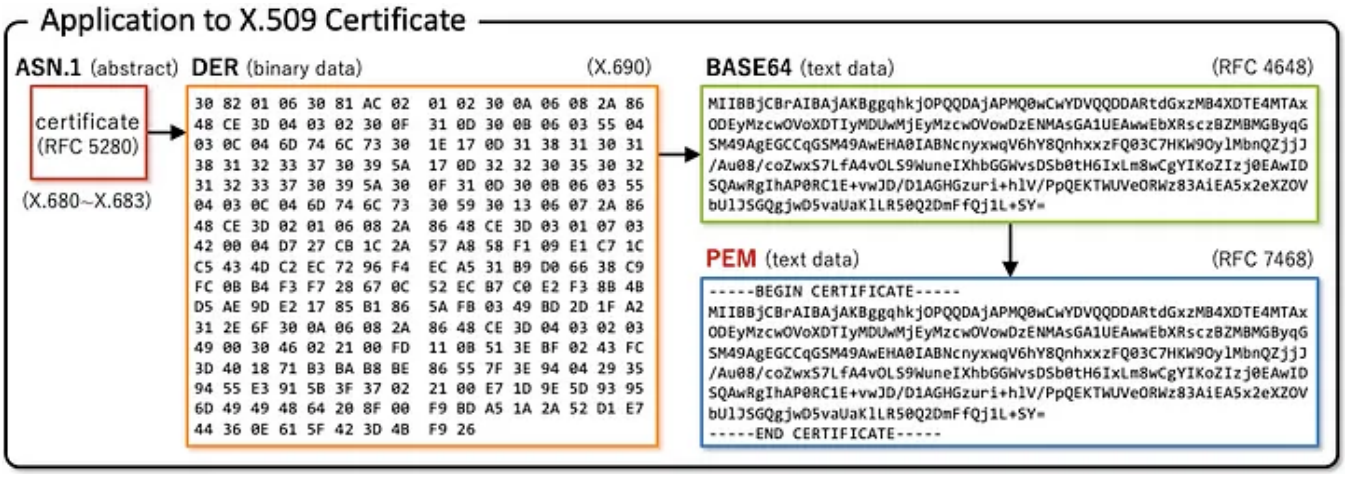
Next, convert the information into a byte sequence using DER. The generated output becomes binary data.



Convert the binary data into text data using BASE64.



Add information by using the rule of PEM. In this example, `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----` are placed at the top and the bottom respectively as the BASE64 represents a certificate.



3.1. Subject

Let's pick up the `subject` field from among the fields contained in a certificate to learn the data structure.

```
TBSCertificate  ::=  SEQUENCE  {
    version         [0]  EXPLICIT Version DEFAULT v1,
    serialNumber         CertificateSerialNumber,
    signature            AlgorithmIdentifier,
    issuer               Name,
    validity             Validity,
    subject              Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                         -- If present, version MUST be v2 or v3
    subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                         -- If present, version MUST be v2 or v3
    extensions      [3]  EXPLICIT Extensions OPTIONAL
                         -- If present, version MUST be v3
    }
```

The `subject` field contains the **distinguished name** of the entity that is associated with the public key. The way to represent a distinguished name as a string is defined in RFC 4514: Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names.

The following are examples of distinguished names listed in the specification.

- `UID=jsmith,DC=example,DC=net`
- `OU=Sales+CN=J. Smith,DC=example,DC=net`
- `CN=James \"Jim\" Smith\, III,DC=example,DC=net`
- `CN=Before\0dAfter,DC=example,DC=net`

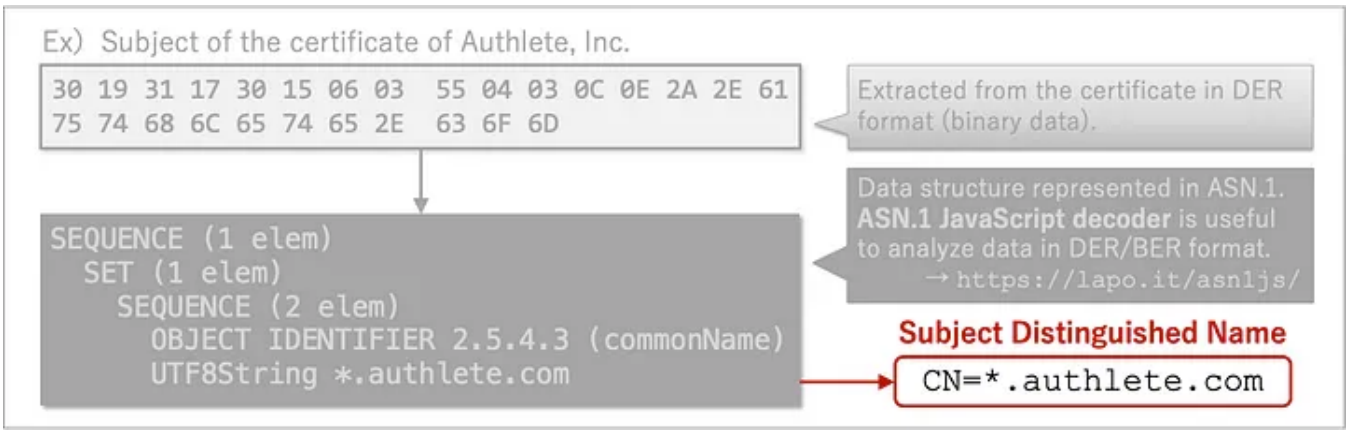
A distinguished name consists of attributes. `UID`, `OU` and `CN` in the examples above are attribute names.

An important usage of the `subject` field to note is that traditionally **a certificate for a website contains the hostname of the server as the value of the `CN` (common name) attribute of the subject distinguished name.**

Let's see an example. The following is the byte sequence representing the `subject` field of the certificate for the website of Authlete, Inc.

Decoding the binary data with ASN.1 JavaScript decoder shows the data structure below.

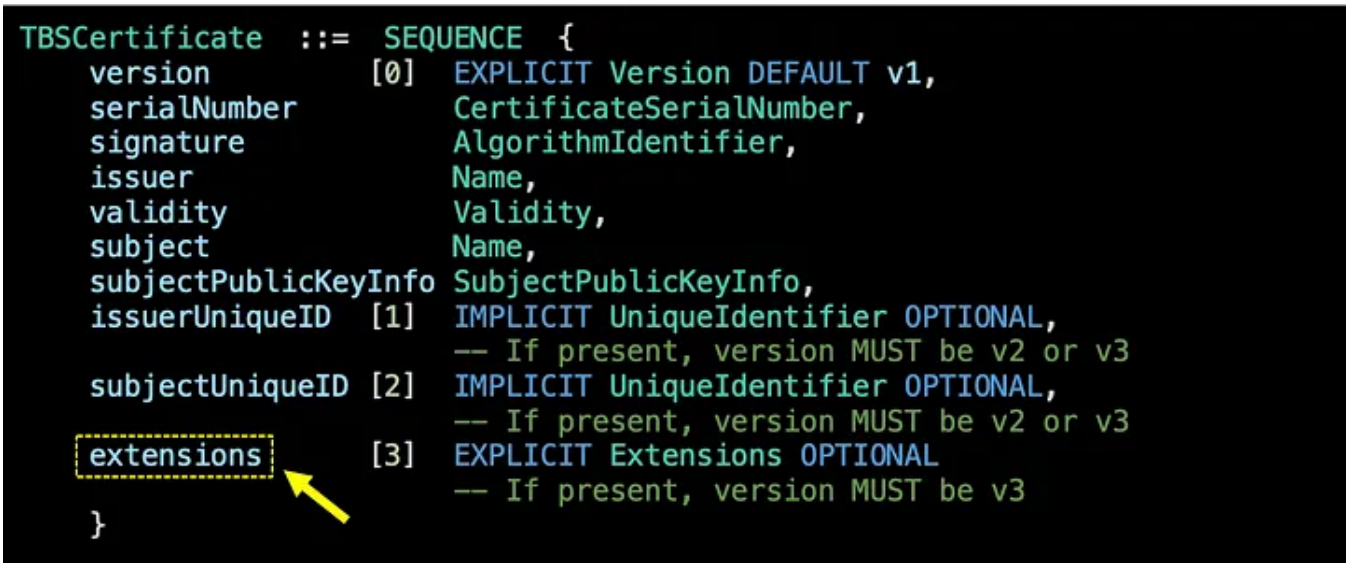
The output indicates that the `subject` field holds a distinguished name whose value is `CN=*.authlete.com`.



3.2. Subject Alternative Name

The traditional way that uses the `CN` attribute of the `subject` field cannot use other identifiers than the hostname. Also, there is no agreed rule whereby to include multiple subjects. Here the **Subject Alternative Name** (**SAN**) extension comes in.

A certificate has a place to hold extended data at.



By putting a SAN extension here, multiple subjects can be specified and other identifiers than the hostname can be used. The SAN extension is defined in Section 4.2.1.6 of RFC 5280 as below.

```
SubjectAltName ::= GeneralNames                                  Subject Alternative Name

GeneralNames ::= SEQUENCE SIZE (1..MAX) OF GeneralName

GeneralName ::= CHOICE {
    otherName                   [0] OtherName,
    rfc822Name                  [1] IA5String,                   Email Address
    dNSName                     [2] IA5String,                   DNS Name
    x400Address                 [3] ORAddress,
    directoryName               [4] Name,
    ediPartyName                [5] EDIPartyName,
    uniformResourceIdentifier   [6] IA5String,                   URI
    iPAddress                   [7] OCTET STRING,                IP Address
    registeredID                [8] OBJECT IDENTIFIER }
```

Let's see an example. The following is the byte sequence representing the SAN extension of the certificate for the website of Authlete, Inc.

Ex) Subject Alternative Name of the certificate of Authlete, Inc.

```
                            30 27 06 03 55 1D 11 04
20 30 1E 82 0E 2A 2E 61   75 74 68 6C 65 74 65 2E
63 6F 6D 82 0C 61 75 74   68 6C 65 74 65 2E 63 6F
6D
```

Extracted from the certificate in DER format (binary data).

Decoding the binary data with ASN.1 JavaScript decoder shows the data structure below.

Ex) Subject Alternative Name of the certificate of Authlete, Inc.

```
                            30 27 06 03 55 1D 11 04
20 30 1E 82 0E 2A 2E 61   75 74 68 6C 65 74 65 2E
63 6F 6D 82 0C 61 75 74   68 6C 65 74 65 2E 63 6F
6D
```

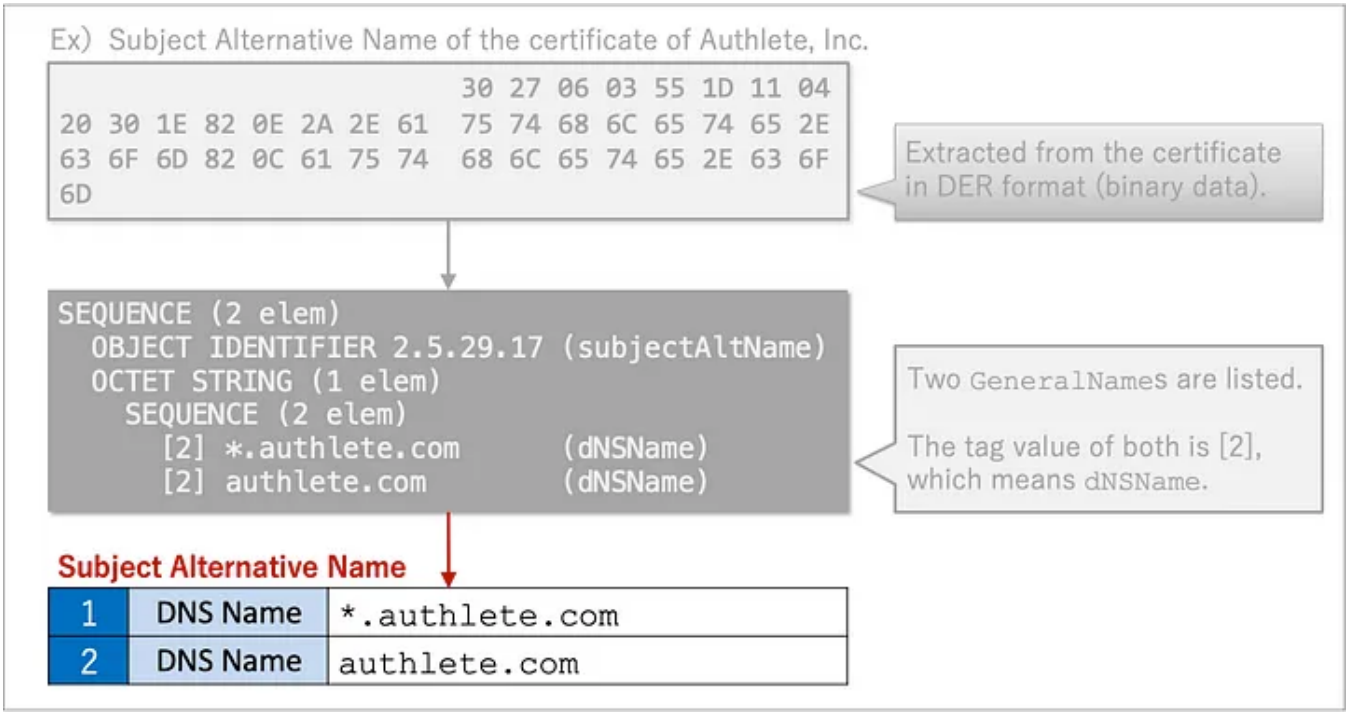
Extracted from the certificate in DER format (binary data).

```
SEQUENCE (2 elem)
  OBJECT IDENTIFIER 2.5.29.17 (subjectAltName)
  OCTET STRING (1 elem)
    SEQUENCE (2 elem)
      [2] *.authlete.com      (dNSName)
      [2] authlete.com        (dNSName)
```

Two GeneralNames are listed.

The tag value of both is [2], which means dNSName.

The output indicates that the SAN extension includes two DNS names, `*.authlete.com` and `authlete.com`.



4. Self-Signed Certificate

Let's deepen the knowledge by generating a self-signed certificate.

4.1. openssl Command

To begin with, check the version of the `openssl` command.

```
$ /usr/bin/openssl version -a
LibreSSL 2.6.5
built on: date not available
platform: information not available
options:  bn(64,64) rc4(16x,int) des(idx,cisc,16,int) blowfish(idx)
compiler: information not available
OPENSSLDIR: "/private/etc/ssl"
```

As Mac uses LibreSSL instead of OpenSSL since macOS High Sierra, the default `openssl` command on Mac will show `LibreSSL ?.?.?` as shown above.

LibreSSL's `openssl` command does not support new command line options added to OpenSSL's `openssl` command, so install OpenSSL before you try to type commands shown in the sebsequent sections.

```
$ brew install openssl
$ /usr/local/opt/openssl/bin/openssl version -a
OpenSSL 1.1.1g  21 Apr 2020
built on: Tue Apr 21 13:28:37 2020 UTC
platform: darwin64-x86_64-cc
options:  bn(64,64) rc4(16x,int) des(int) idea(int) blowfish(ptr)
compiler: clang -fPIC -arch x86_64 -O3 -Wall -DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_CP
UID_OBJ -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL
_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM -DMD5
_ASM -DAESNI_ASM -DVPAES_ASM -DGHASH_ASM -DECP_NISTZ256_ASM -DX25519_ASM -DPOLY130
5_ASM -D_REENTRANT -DNDEBUG
OPENSSLDIR: "/usr/local/etc/openssl@1.1"
ENGINESDIR: "/usr/local/Cellar/openssl@1.1/1.1.1g/lib/engines-1.1"
Seeding source: os-specific
```

4.2. Private Key Generation

Let's generate a private key.

```
$ openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 > private_key.pem
```

- `genpkey` : generates a private key. The `openssl` command provides `genrsa` and `gendsa` subcommands to generate RSA/DSA private keys, but they are superseded by `genpkey` .
- `-algorithm EC` : uses Elliptic Curve algorithm. Note that the `-algorithm` option must appear before `-pkeyopt` options when used.
- `-pkeyopt ec_paramgen_curve:P-256` : uses the P-256 curve which is a parameter specific to Elliptic Curve algorithm. P-256 is one of the curves recommended by NIST (National Institute of Standards and Technology). Refer to "D.2.3. Curve P-256" in Digital Signature Standard (DSS) if you are interested.

The command above will generate a file (`private_key.pem`) whose content looks like below. The format of the file is PEM. To indicate that the BASE64 part represents a private key, the file has `-----BEGIN PRIVATE KEY-----` and `-----END PRIVATE KEY-----` at the top and the bottom, respectively.

```
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgwSmbZ2gTKFbyy+q6
jXhuGXFuz8mCqjUhRYvQ/EEgqqihRANCAAS4QckKxSuC89ZfQyTWP+M01QUcJRRS
HG/oYhFEU5FOoSJGgxZgJeUFUglE3HiT/NCsdj8COWDCTqP9Iz2oEDnz
-----END PRIVATE KEY-----
```

4.3. Private Key Content

Let's confirm the content of the generated private key.

```
$ openssl pkey -text -noout -in private_key.pem
Private-Key: (256 bit)
priv:
    c1:29:9b:67:68:13:28:56:f2:cb:ea:ba:8d:78:6e:
    19:71:6e:cf:c9:82:aa:35:21:45:8b:d0:fc:41:20:
    aa:a8
pub:
    04:b8:41:c9:0a:c5:2b:82:f3:d6:5f:43:24:d6:3f:
    e3:34:d5:05:1c:25:14:52:1c:6f:e8:62:11:44:53:
    91:4e:a1:22:46:83:16:60:25:e5:05:52:09:44:dc:
    78:93:fc:d0:ac:76:3f:02:39:60:c2:4e:a3:fd:23:
    3d:a8:10:39:f3
ASN1 OID: prime256v1
NIST CURVE: P-256
```

- `pkey` : is the subcommand for key operations.
- `-text` : displays information in plain text.
- `-noout` : suppresses the encoded output.
- `-in private_key.pem` : specifies the input file.

The output above implies that the private key includes the paired public key (you can find `pub` in the output). Converting the format of the key from PEM to **JWK** (RFC 7517) will show the fact more clearly. Here I use eckles for conversion.

```
$ npm install -g eckles
$ eckles private_key.pem > private_key.jwk
$ cat private_key.jwk
{
  "kty": "EC",
  "crv": "P-256",
  "d": "wSmbZ2gTKFbyy-q6jXhuGXFuz8mCqjUhRYvQ_EEgqqg",
  "x": "uEHJCsUrgvPWX0Mk1j_jNNUFHCUUUhxv6GIRRFORTqE",
  "y": "IkaDFmAl5QVSCUTceJP80Kx2PwI5YMJOo_0jPagQOfM"
}
```

The `d`, `x` and `y` in the JSON are parameters specific to Elliptic Curve algorithm. The `d` parameter is included only in the private key while the `x` and `y` parameters are included in both the private and public keys. Removing the `d` parameter from the JWK will change the private key to the paired public key.

If you extract the public key from the private key in PEM format,

```
$ openssl pkey -pubout -in private_key.pem > public_key.pem
$ cat public_key.pem
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEuEHJCsUrgvPWX0Mk1j/jNNUFHCUU
Uhxv6GIRRFORTqEiRoMWYCXlBVIJRNx4k/zQrHY/Ajlgwk6j/SM9qBA58w==
-----END PUBLIC KEY-----
```

and convert the format of the public key from PEM to JWK,

```
$ eckles public_key.pem > public_key.jwk
$ cat public_key.jwk
{
  "kty": "EC",
  "crv": "P-256",
  "x": "uEHJCsUrgvPWX0Mk1j_jNNUFHCUUUhxv6GIRRFORTqE",
  "y": "IkaDFmAl5QVSCUTceJP80Kx2PwI5YMJOo_0jPagQOfM"
}
```

you can confirm that the public key does not contain the `d` parameter.

4.4. Certificate Generation

Let's generate a self-signed certificate.

```
$ openssl req -x509 -key private_key.pem -subj /CN=client.example.com > certificate.pem
```

- `req -x509` : generates a self-signed X.509 certificate. The primary usage of the `req` subcommand is to generate CSR (Certificate Signing Request), but the subcommand generates a self-signed certificate instead of CSR when the `-x509` option is given.
- `-key private_key.pem` : specifies the private key for signing and the target public key.

- `-subj /CN=client.example.com` : specifies a subject distinguished name. If the `-subj` option is not given, the `openssl` command shows a prompt for you to input a subject distinguished name interactively.
- If no valid period is given, 30 days is used as the default value. The `-days` option can be used to specify a valid period in days.

The command above will generate a file (`certificate.pem`) whose content looks like below. The format of the file is PEM. To indicate that the BASE64 part represents a certificate, the file has `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----` at the top and the bottom, respectively.

```
-----BEGIN CERTIFICATE-----
MIIBjzCCATWgAwIBAgIUdRbH+ElI8iLjnR9eJwCpIU8e8xYwCgYIKoZIzj0EAwIw
HTEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMB4XDTIwMDYyNzExMzgwM1oX
DTIwMDcyNzExMzgwM1owHTEbMBkGA1UEAwwSY2xpZW50LmV4YW1wbGUuY29tMFkw
EwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEuEHJCsUrgvPWX0Mk1j/jNNUFHCUUUhxv
6GIRRFORTqEiRoMWYCXlBVIJRNx4k/zQrHY/Ajlgwk6j/SM9qBA586NTMFEwHQYD
VR0OBBYEFJaMKA22eKiMXGvSojeoLGChcABcMB8GA1UdIwQYMBaAFJaMKA22eKiM
XGvSojeoLGChcABcMA8GA1UdEwEB/wQFMAMBAf8wCgYIKoZIzj0EAwIDSAAwRQIg
N+a6RbvOz+32qOOgKnbQB8sSVeD0gvRoRK13ZuducX4CIQDkgzc1BHoQJ6Pby3aO
byBmhjJS/LhhROgzecP/JMDxqA==
-----END CERTIFICATE-----
```

4.5. Certificate Content

Let's confirm the content of the generated certificate.

```
$ openssl x509 -text -noout -in certificate.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            75:16:c7:f8:49:48:f2:22:e3:9d:1f:5e:27:00:a9:21:4f:1e:f3:16
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN = client.example.com
        Validity
            Not Before: Jun 27 11:38:03 2020 GMT
            Not After : Jul 27 11:38:03 2020 GMT
        Subject: CN = client.example.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:b8:41:c9:0a:c5:2b:82:f3:d6:5f:43:24:d6:3f:
                    e3:34:d5:05:1c:25:14:52:1c:6f:e8:62:11:44:53:
                    91:4e:a1:22:46:83:16:60:25:e5:05:52:09:44:dc:
                    78:93:fc:d0:ac:76:3f:02:39:60:c2:4e:a3:fd:23:
                    3d:a8:10:39:f3
                ASN1 OID: prime256v1
                NIST CURVE: P-256                    Public Key Info
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                96:8C:28:0D:B6:78:A8:8C:5C:6B:D2:A2:37:A8:2C:60:A1:70:00:5C
            X509v3 Authority Key Identifier:
                keyid:96:8C:28:0D:B6:78:A8:8C:5C:6B:D2:A2:37:A8:2C:60:A1:70:00:5C

            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:20:37:e6:ba:45:bb:ce:cf:ed:f6:a8:e3:a0:2a:76:
         d0:07:cb:12:55:e0:f4:82:f4:68:44:ad:77:66:e7:6e:71:7e:
         02:21:00:e4:83:37:35:04:7a:10:27:a3:db:cb:76:8e:6f:20:
         66:86:32:52:fc:b8:61:44:e8:33:79:c3:ff:24:c0:f1:a8
```

- `x509` : is the subcommand for X.509 certificate operations.
- `-text` : displays information in plain text.
- `-noout` : suppresses the encoded output.
- `-in certificate.pem` : specifies the input file.

Note that the issuer and the subject are identical because the input certificate is self-signed. Information about the target public key is included in the Subject Public Key Info block.

Finally

The well-known usage of X.509 certificates is TLS (Transport Layer Security), but X.509 certificates are used in other various places. For example, in the OAuth world,

RFC 8705 defines methods to utilize X.509 certificates for client authentication and certificate-bound access tokens. Please visit "**OAuth 2.0 Client Authentication**" and "**OAuth Access Token Implementation**" for further technical details.

I hope this article could help you understand the basics of X.509 certificates. Finally, I highly recommend "Bulletproof SSL and TLS" (by Ivan Ristić) for deeper dive.

Thank you for reading.

X509 Certificate Openssl

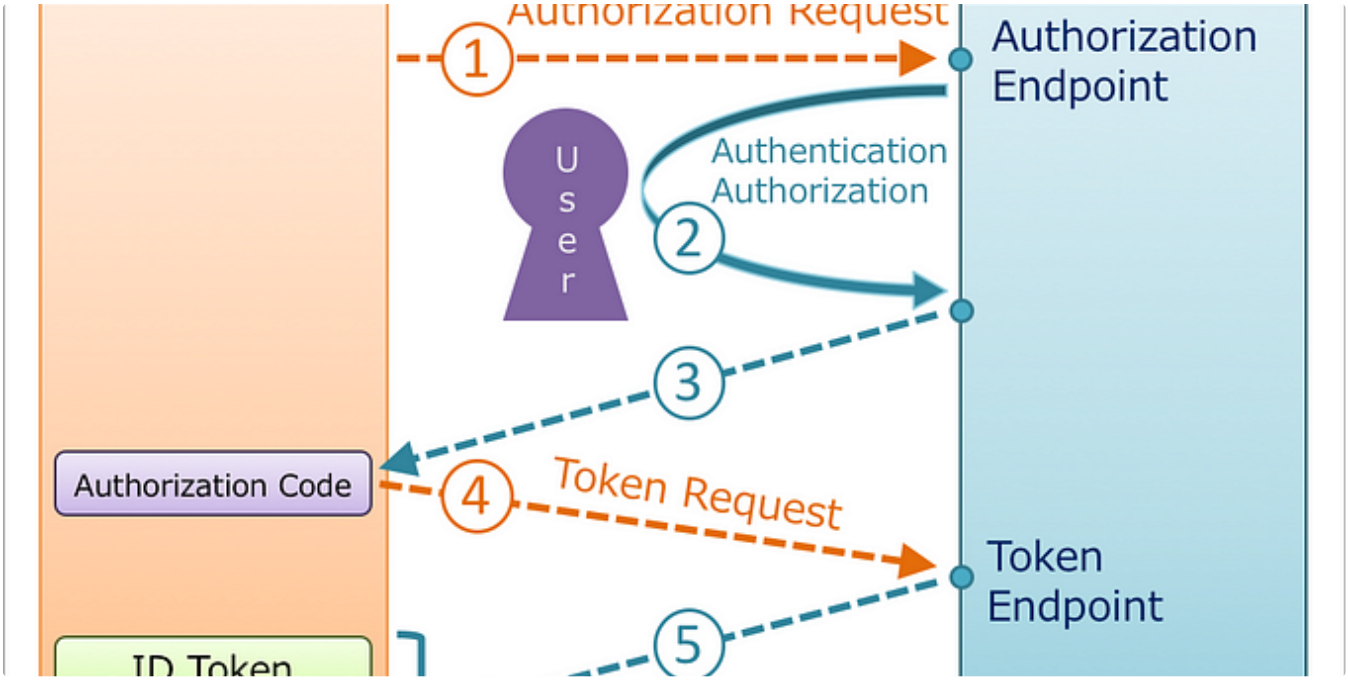
Follow

Written by Takahiko Kawasaki

2.5K Followers

Co-founder and representative director of Authlete, Inc., working as a software engineer since 1997.
https://www.authlete.com/

More from Takahiko Kawasaki



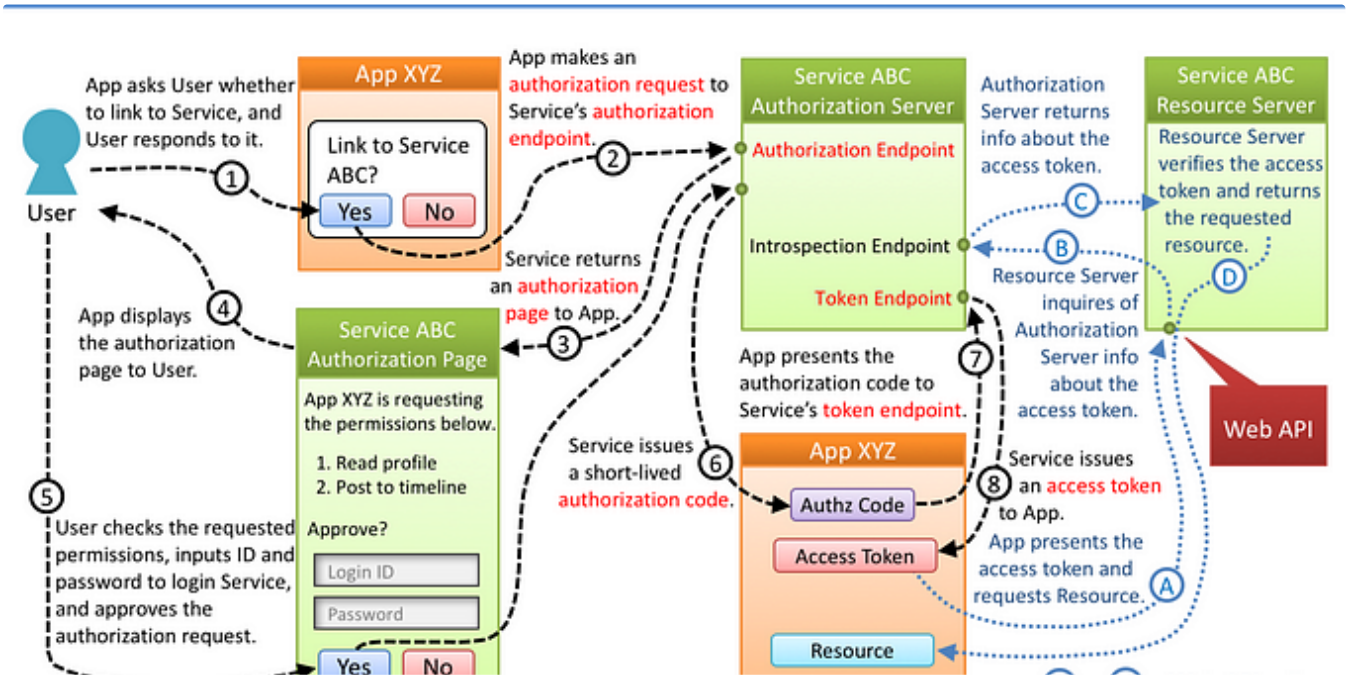
Takahiko Kawasaki

Diagrams of All The OpenID Connect Flows

Introduction

8 min read · Oct 30, 2017

2.4K 11



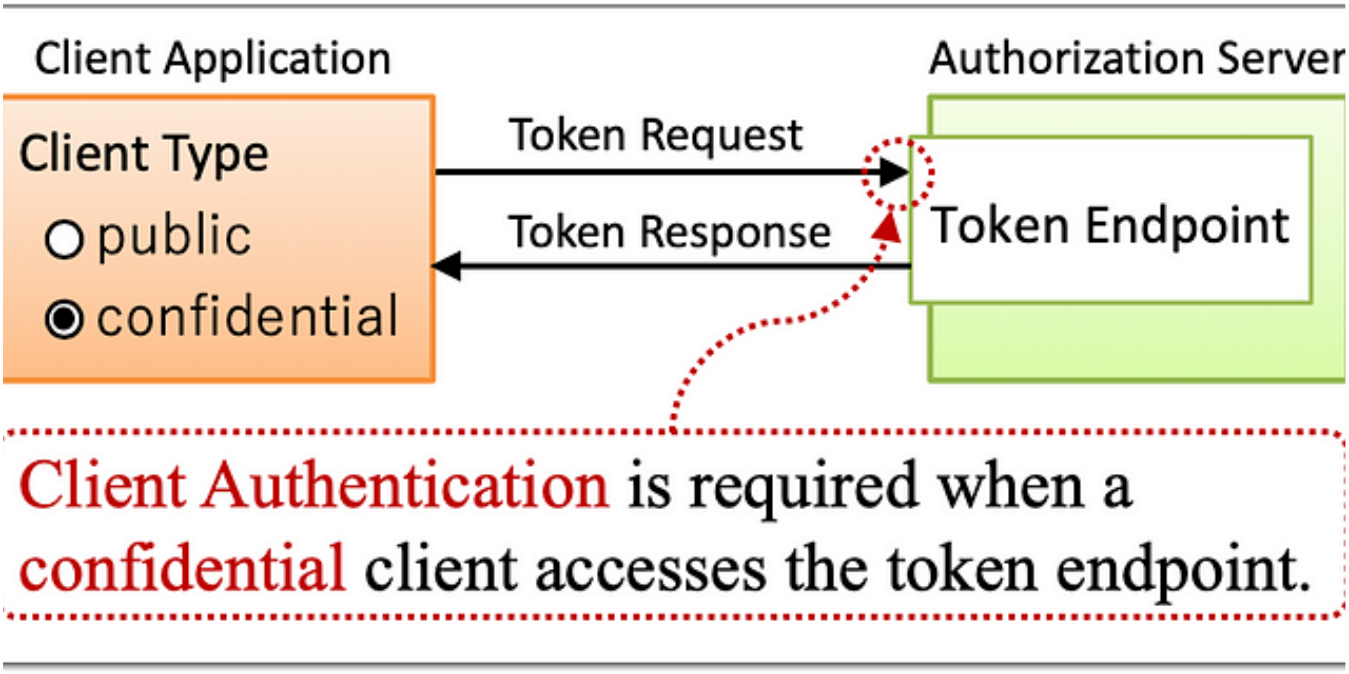
Takahiko Kawasaki

Diagrams And Movies Of All The OAuth 2.0 Flows

Diagrams and movies of all the 4 authorization flows defined in RFC 6749 (The OAuth 2.0 Authorization Framework) and one more flow to...

6 min read · May 26, 2017

2.5K 14



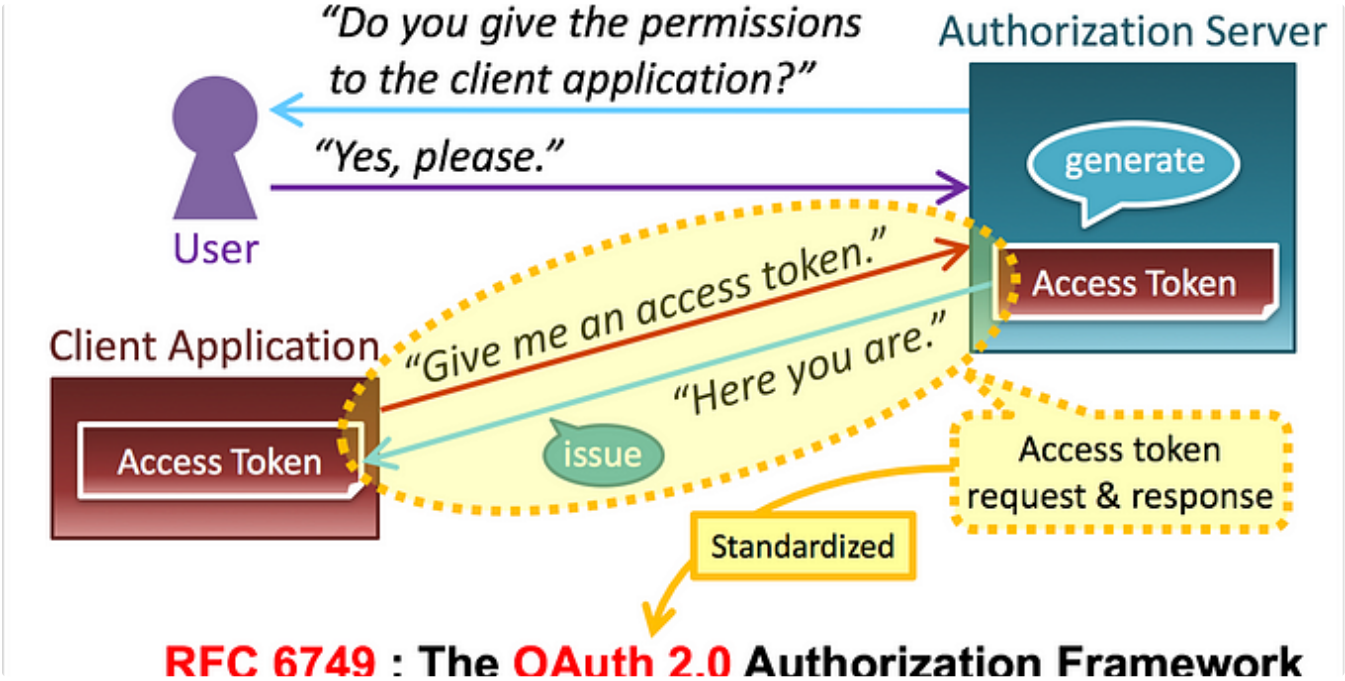
Takahiko Kawasaki

OAuth 2.0 Client Authentication

This article explains “OAuth 2.0 client authentication”.

12 min read · Jul 18, 2019

652 3



Takahiko Kawasaki

The Simplest Guide To OAuth 2.0

For the past three years, I've repeated to explain OAuth 2.0 to those who don't have a technical background, mainly to investors as a...

6 min read · Aug 1, 2017

6.9K 38

See all from Takahiko Kawasaki

Recommended from Medium

Prof Bill Buchanan OBE in ASecuritySite: When Bob Met Alice

SSL/TLS: The Dinosaur Protocols? Meet Message Layer Security (MLS)

Go, on, ask a cybersecurity professional to explain how SSL/TLS works and then ask them to explain PKI (Public Key Infrastructure) and...

5 min read · Aug 22, 2023

477 11

Viraj Shetty

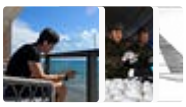
OAuth or OpenID Connect — Know the difference

There is a lot of confusion regarding the difference between OAuth 2 and OpenID Connect. This article attempts to explain the difference by...

6 min read · Oct 13, 2023

81 2

Lists



Staff Picks

550 stories · 616 saves

Stories to Help You Level-Up at Work

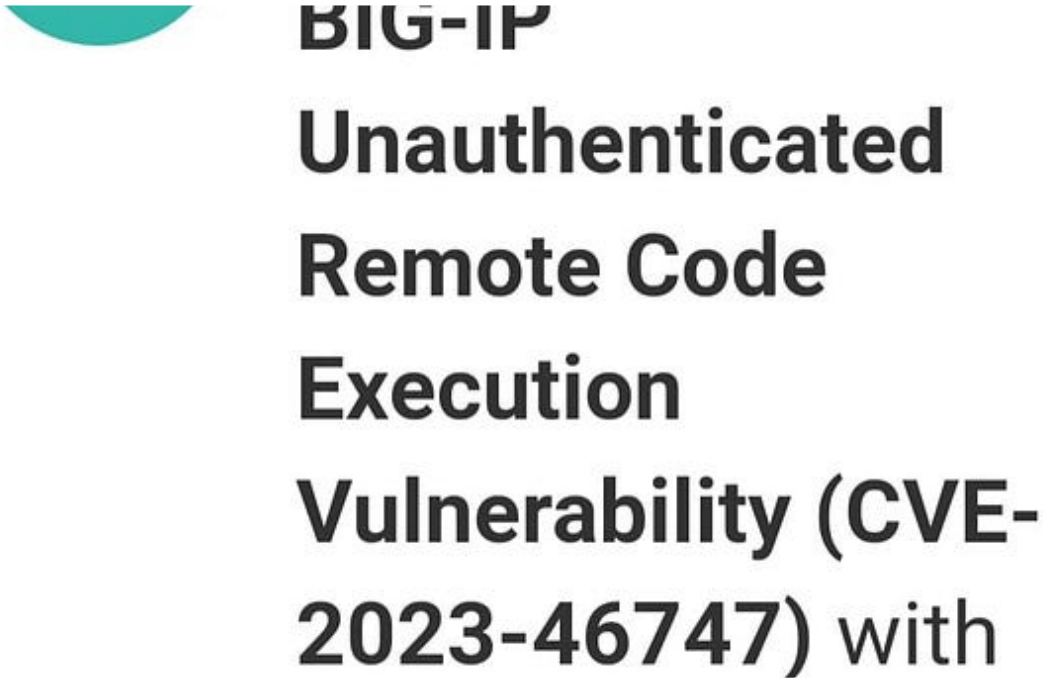
19 stories · 406 saves

Self-Improvement 101

20 stories · 1169 saves

Productivity 101

20 stories · 1070 saves



MS17-010

How I Discovered an RCE Vulnerability in Tesla, Securing a $10,000 Bounty

Myself: I am Raguraman , Security Researcher | Bug Hunter | CTF Player | Secured @ Tesla,Apple,Amazon,Oracle & more

4 min read · Dec 24, 2023

1.5K 19

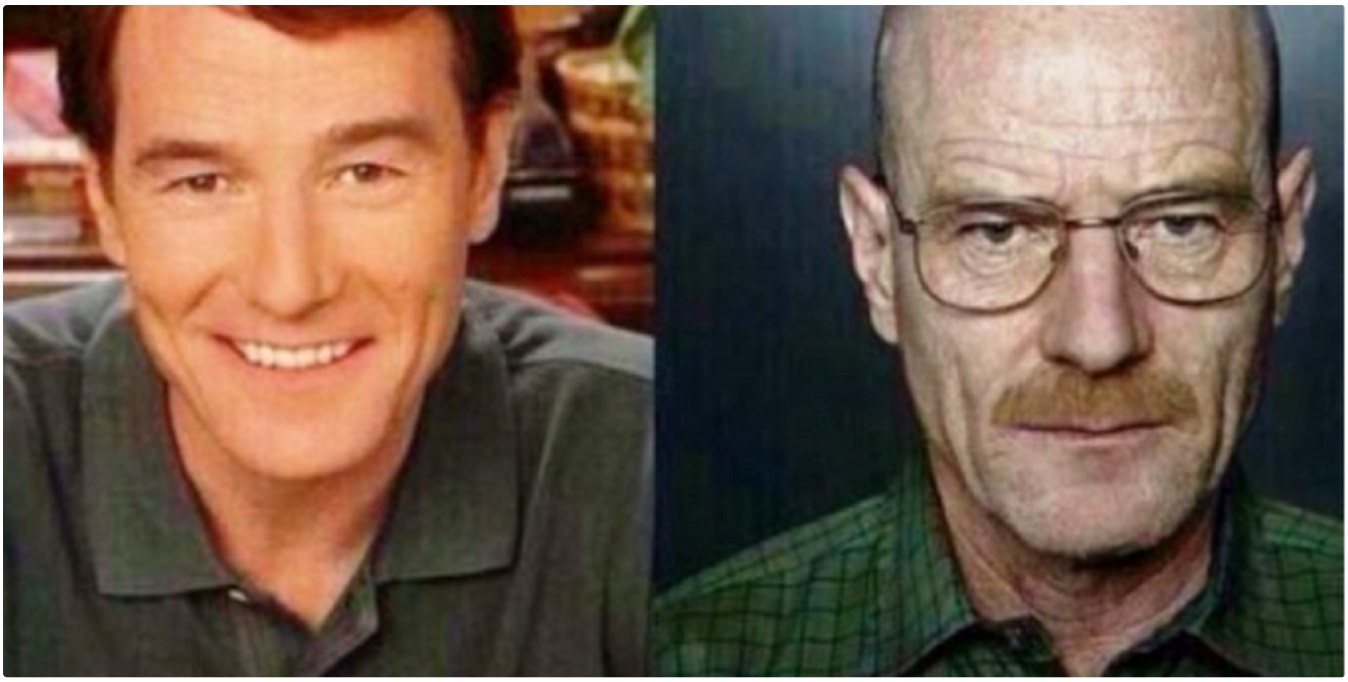
PRASAD

Bypass SSL Pinning for Flutter

What is Flutter?

7 min read · Dec 15, 2023

42 2



David Goudet

This is Why I Didn't Accept You as a Senior Software Engineer

An Alarming Trend in The Software Industry

5 min read · Jul 25, 2023

7.6K 75

Manish Singh

How I Passed OSCP 2023 in Just 8 Hours with 110 Points Without Using Metasploit

Hey everyone, If you've ever been curious about how to pass OffSec Certified Professional (OSCP) exam and get certified so this blog is for...

10 min read · Aug 17, 2023

466 7