

# SIO Lab

## Validation of X.509 Certificates

version 2.0

Author: André Zúquete, Vitor Cunha

Version log:

- 1.0: Initial version
- 1.1: Corrections in the functions' guidelines for certification\_path.py

## 1 Introduction

Public Key certificates comply with the X.509v3 standard. They are managed in the context of a PKI (Public Key Infrastructure) that defines the policies under which the certificates may be issued and used. The IETF PKIX Working Group<sup>1</sup> produces most standards regulating the use of X.509 certificates on the Internet. PKIX means X.509-based PKI.

In this guide, you will focus on the certificate validation processes. For creating Certification Authorities and certificates, explore the `xca`<sup>2</sup> tool (GUI) or look at the syntax of `openssl x509` (command line).

A certificate is considered valid for a **specific purpose** if the following conditions are verified:

1. If the certificate is used to validate a signature, its must have been performed during its validity interval;
2. If the certificate was issued (signed) by an entity (CA) in which the user trusts (directly or transitively);  
and
3. If the policies in the certificate allow its use for the intended specific purpose.

Here, we are going to handle the first two conditions and part of the third. For additional information on the subject of this guide, you may consult the Python Cryptography X509 documentation<sup>3</sup>.

You must obtain a X.509 Public Key Certificate to complete this guide. These certificates are widely available on HTTPS-enabled websites. You can obtain the certificate from any website. The following command exemplifies how to obtain the certificates from the university's main webpage:

```
openssl s_client -connect www.ua.pt:443 -showcerts < /dev/null
```

The example output should show you three certificates, that you can identify by the `BEGIN CERTIFICATE` and `END CERTIFICATE` lines:

```
Certificate chain
0 s:CN = www.ua.pt
  i:C = GR, O = Hellenic Academic and Research Institutions CA, CN = GEANT TLS RSA 1
  a:PKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
  v:NotBefore: May 13 13:10:00 2025 GMT; NotAfter: May 13 13:10:00 2026 GMT
-----BEGIN CERTIFICATE-----
MIIGazCCBN0gAwIBAgIQWxuqV9ciT75GTop4N6zDtTANBgkqhkiG9w0BAQsFADBg
MQswCQYDVQQGEwJHUjE3MDUGA1UECgwuSGVsbGVuaWMgQWNhZGVtaWMgYW5kIFJl
c2VhcmNoIEluc3RpdHVOaW9ucyBDQTlUEAwWPROVBTLQgVExtTIFJTQSAX
-----END CERTIFICATE-----
```

<sup>1</sup>[Public-Key Infrastructure \(X.509\) \(pkix\)](#)

<sup>2</sup>[XCA](#) (install with: `apt install xca`)

<sup>3</sup>[X.509 Reference](#)

If you copy and paste each certificate to a file, you can see more certificate details with the command:

Or with the XCA GUI:

You may be wondering why you got several certificates (in this case, 3) instead of just one. The reason is simple: for performance and simplicity purposes. As you will find out later in this guide, if you only have one certificate, you have to request each issuer's certificate in the chain, something that takes time and might be prone to errors (e.g., you might not be able to find where the issuer's certificate can be found). Therefore, it is usual for HTTPS servers to send the complete certification chain to fasten the chain creation validation and make it more robust.

X.509 Public Key Certificates are also present in the Portuguese Citizen Card, and you can create your own Certification Authority and certificates with XCA (useful for testing). The code and patterns you will develop in this guide will also apply in those contexts.

X.509 public key certificates are not perpetually valid. That is, a temporal validity interval will always define when the certificate may be used, and that is a basic validity check that must always be performed. The temporal validity acts as a boundary to invalidate any further use of the private key, even when it is lost.

2

We already provide you with a starting point for this program in `validity_check.py`. As you can verify, the given code already has a main function, which is handling the arguments given to the program and is calling another function, called `valid`. The function `valid` must receive a certificate, and will return `True` if the certificate is valid at this moment, or `False` otherwise. **You must complete the function `valid`.**

**Hint:** In Python, use the `datetime` module to get the current time. Also be aware that dates in certificates are in the UTC timezone.

**Note:** the validity of a certificate refers, indirectly, to the validity of operations performed with the corresponding private key. For instance, a signature produced with that private key must have a date during the validity period of the certificate.

To run the program, you just need to type in the terminal:

```
$ validity_check.py <certificate_file>
The given certificate was valid
```

You have an obsolete certificate in the files provided (`obsolete_certificate.pem`) to check your program.

This function you just created will be later used to verify the validity of all certificates in a chain.

## 3 Trust Anchor certificates

Trust anchor certificates are the certificates the user (or application) already trusts. These are typically root certificates (i.e., self-signed certificates). Trust anchor certificates are essential to validate certification paths (or certificate chains) because these define when you have reached trust through transitive properties. A valid certification path can only be trusted **iff a trust anchor certificate exists in that path**. You may encounter a valid certification path that is not trusted, meaning that no trust anchor is present. Trust management and anchor certificates are essential for establishing trust in X.509. Usually, trust anchor certificates are provided by the operating system or the application in use (e.g., Firefox, Chrome, etc.). Sometimes, the user may also provide new trust anchors, depending on the application and the security model.

### 3.1 Reading trust anchor certificates

The trust anchor certificates must be protected in a `keystore` or a restricted location (`/etc/ssl/certs`) to prevent unwanted additions or removals, be these intentional (e.g., adversarial) or unintentional (i.e., accidents). If an attacker is able to add a certificate to the `keystore`, it will be able to break the trust chain and impersonate other entities (Intercept HTTPS traffic or VPN traffic, install malicious updates). You can use the certificates in your Linux system as individual files containing certificates in the `PEM` format.

**Task:** Implement a small program that reads all system-trusted certificates into a dictionary (a `map`) of **trusted certificates**, using the `subject` as the dictionary key.

You may start by completing the code given in the file `trusted_certificates.py`. There, you have a function named `trusted`, that receives a directory name (such as the `/etc/ssl/certs`) as argument and must return a dictionary. This dictionary must have as keys the certificates subjects and as values the corresponding certificates.

To run the program, you just need to type in the terminal:

```
$ trusted_certificates.py /etc/ssl/certificates
153 valid trusted certificates found
```

**Attention:** Avoid loading certificates that have already expired (use the previously developed function).

**Hint:** Use the `os.scandir` object to scan for all certificates in `/etc/ssl/certs`.

## 4 Build a certification path

A certificate chain, or certification path, is the set of certificates that composes a chain of trust, from a trust anchor certificate to an end user certificate. Frequently, to validate a end-user certificate, it is necessary to build the certification path from a set of several candidate intermediary certificates. Other times, the entity to be validated (e.g., web server) provides the respective certification path with its certificate.

**Task:** For this exercise, take each **user-provided certificate** (from the Citizen's Card, XCA, downloaded, etc.) and recreate its validation chain in a list. The program shall receive as argument the certificate from which the chain is built, and shall obtain each issuer certificate from the certificate's **Authority Information Access** extension. This extension usually has two fields:

- The **OCSP**: we will take a look into it later in this guide.
- The **CA Issuers**: a list of URLs where you can find the issuer certificate. **This is where you must obtain the URL to obtain the issuer certificate**

You should stop when you get a root certificate (i.e., self-signed), or a trust anchor certificate. The root certificate can be missing from the list you create. That usually means the root certificate is already in your trusted certificates. Otherwise, the chain will be untrusted unless you already trust the end-certificate or any of its intermediate certificates.

The file `certification_path.py` already has a blueprint for the code you must create. In this script, you have two incomplete functions you need to complete.

The first is the `build_cert_path`, which receives a certificate as argument, from which you need to extract the location (URL) of the issuer's certificate. To get it, you need to obtain the URL from the **CA Issuers** field under the **Authority Information Access** extension of the certificate given by argument.

The second is the `get_issuer_cert`, which receives that URL as argument to fetch a certificate.

Therefore, you need to recursively call the `build_cert_path` function to get each chain's certificate issuer, until you reach a certificate in which you trust. At the end, this function must return a list with the chain certificates.

To run the program, you just need to type in the terminal (this exemple used the UA HTTPS server certificate):

```
$ certification_path.py <certificate_file>
Built path for CN=www.ua.pt
CN=GEANT TLS RSA 1,0=Hellenic Academic and Research Institutions CA,C=GR
CN=HARICA TLS RSA Root CA 2021,0=Hellenic Academic and Research Institutions CA,C=GR
The chain ended, this is a self-certified certificate
```

**Note:** some certificates might not have a **CA Issuers** field under **Authority Information Access**. In such cases, however, you may have that certificate already installed on your equipment. You can use the function created in the previous exercise to verify if that is the case (in such a case, that certificate can end the chain, as you reached a certificate which you trust).

**Hint:** Remember that the user and root certificates are loaded into a dictionary with the **subject** as the key. This greatly facilitates the process of getting the **issuer** of a new certificate, as it should exists in the dictionary, and can be retrieved by this value (the **issuer** field of a child certificate, corresponds to the **subject** field of a parent certificate).

**Note:** as mentioned in the introduction, sometimes you obtain from an HTTPS server multiple certificates. You can further improve this program to use those certificates instead of trying to request them.

## 5 Validate a certification path

Given a certification path, we can validate it by verifying each certificate and the relations between certificates.

**Task:** Create a program that validates the certification path of a given certificate. For validating the chain, you will need to, at least, validate its validity interval (from the first exercise), its purpose, its signature, and if it was already revoked or not:

- **Signature validation:** you need to verify if each certificate's signature is valid considering the public key of the issuer's certificate. In Python's cryptography, the signed data is present in the attribute `x509.tbs_certificate_bytes` (you have an example on how to verify the signature in the given URL).
- **Purpose validation:** you need to verify if a certificate which is issuing other certificates has that purpose. The most simple verification is to verify if an issuer certificate has that purpose. That information is found under the `ca` field in the `X509v3 Basic Constraints` certificate extension. If this field has the value `true`, the CA of that certificate can issue other certificates. If it is `false`, it can not. Note also that this extension is normally critical, and you can verify if this is true. **If you are using, for instance, a website certificate as the initial one in the chain, you do not need to do this validation for it, as it is not being used for issuing other certificates.** Nevertheless, you need to do that for the other certificates in the chain.
- **Key usage:** Another important purpose validation is the verification if the subject can use its private key to sign certificates. This is provided by the `Key Usage` extension, which is also normally critical. You should also verify this property for any CA certificates. There are other usages that you should, in some circumstances, verify. For instance, if the certificate is used in TLS, it should have the `Key Encipherment` and `Digital Signature` key usage, as well as the `TLS Web Server Authentication` (at least) extended key usage. In this exercise, however, you can ignore this validation, but bear in mind that it is an important validation, for example, when you connect to a website through HTTPS.
- **Revocation validation:** you need to verify if each certificate was revoked by its CA since it was issued. There are two ways of verifying that. You can check the [Certificate Revocation List \(CRL\)](#), which is a list of all the certificates revoked by that certificate's CA, or use the [Online Certificate Status Protocol \(OCSP\)](#) to directly verify if the certificate in question was revoked by directly "asking" CA. Note that some certificates do not support both types of validation. Therefore, your code should first try to verify if each certificate was revoked using OCSP (because it is usually faster), and if it fails (because, for example, the certificate does not support the OCSP method), you should try to use the CRL method. OCSP

The starting point for this exercise can be found at within the `certification_path_validation.py` file. The given code already has a `validate_chain` function which validates the chain according to the below verifications. Analyze this code and understand how it works. If you analyze carefully, you can check that all other functions (apart from the ones that you already created in the previous exercises) are just returning what is necessary to have a valid certificate. That is, you need to create the code for the functions `valid_signature`, `verify_constraint`, `key_usage`, `revoked`.

To run the program, you just need to type in the terminal (this exemple used the UA HTTPS server certificate):

```
$ certification_path_validation.py <certificate_file>
Validate path for CN=www.ua.pt
  CN=GEANT TLS RSA 1,0=Hellenic Academic and Research Institutions CA,C=GR
  Valid signature over issued certificate
  Basic Constraints
    Issued for a CA
  Key Usage:
    Critical extension
    The corresponding private key can sign
    The corresponding private key can sign public key certificates
    The corresponding private key can sign CRLs
  OCSP validation:
    OCSP status is valid
  CN=HARICA TLS RSA Root CA 2021,0=Hellenic Academic and Research Institutions CA,C=GR
  Valid signature over issued certificate
  Basic Constraints
    Issued for a CA
  Key Usage:
    Critical extension
    The corresponding private key can sign
    The corresponding private key can sign public key certificates
    The corresponding private key can sign CRLs
```

```
OCSP validation:
  No OCSP information
The chain ended, this is a self-certified certificate
```

**Task:** Add an extra parameter to this program to import a list of trusted roots in order to verify if any of the chain certificates is a trusted root.

## 6 Closing note

Establishing proper time is critical for X.509 validations. You cannot confidently validate the status at a past time unless you can establish fully trusted timestamps for each moment (i.e., you must trust all clocks before validating the paths). Otherwise, there is no other time like the present.