# SSL/TLS Certificates and The Public-Key Infrastructure (PKI)

LEONARDO TAMIANO

# TABLE OF CONTENTS


- Limitations of Asymmetric Cryptography
- x.509 Certificates
- The Public Key Infrastructure
- Hands-On 1: HTTPs Web Server
- Hands-On 2: TLS Exporters

# LIMITATIONS OF ASYMMETRIC CRYPTOGRAPHY

## **The Power of Asymmetric Cryptographic** (1/5)

---

**Asymmetric cryptography** allows two entities to share a common secret using an unsafe communication chanel.

# The Power of Asymmetric Cryptographic (2/5)

**Secret**

**Insecure Channel**

## **The Power of Asymmetric Cryptographic** (3/5)

There are various algorithms for exchaning data securely using asymmetric cryptography:

- Diffie-Hellman Key Exchange
- RSA Key Exchange

# The Power of Asymmetric Cryptographic (4/5)

## Diffie-Hellman Key Exchange

private: x
public: g, p, g^x mod p

g^x mod p

g^y mod p

private: y
public: g, p, g^y mod p

**Secret shared: g^(x*y) mod p**

# The Power of Asymmetric Cryptographic (5/5)

## RSA Key Exchange

private: k

**Q: Your public key?**

**(e, N)**

**k^e mod N**

private: d
public: (e, N)

**Secret shared: k**

Mathematically, both **DH** and **RSA** works out.

In the real world, there is a **biggggg problem** to overcome.

# Theory vs Practice

**Q**: Where is the problem?

**Q**: Where is the problem?

---

# The public key is not authenticated!

**The public key is not authenticated!**

---

More specifically, the **mathematical binding that exists private-key and public-key is not enough** for providing secure connection in case of **Man-in-the-Middle** (MitM) attacks.

# DH is not enough in MITM scenarios

private: x
public: g, p, g^x mod p

private: z
public: g, p, g^z mod p

private: y
public: g, p, g^y mod p

g^x mod p

g^z mod p

Secret shared: g^(x*z) mod p

g^z mod p

g^y mod p

Secret shared: g^(y*z) mod p

# RSA is not enough in MITM scenarios

private: k

private: d2
public: (e2, N2)

private: d1
public: (e1, N1)

Q: Your public key?

(e2, N2)

k^e2 mod N2

Q: Your public key?

(e1, N1)

k^e1 mod N1

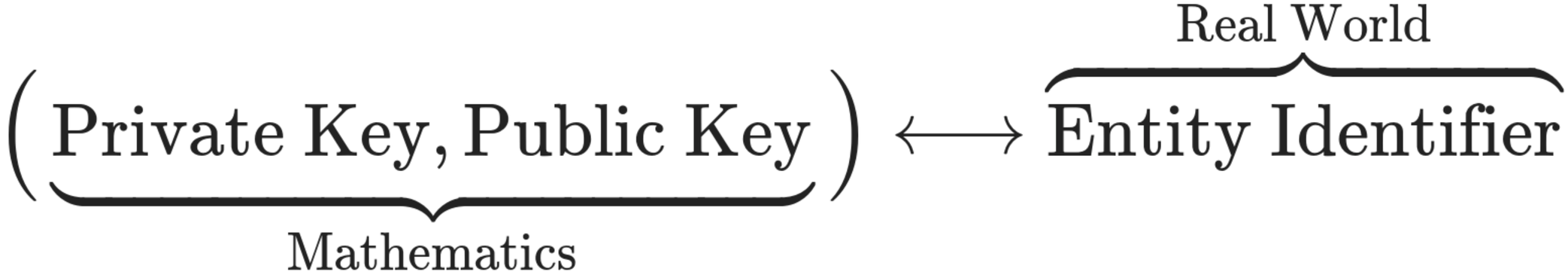
To overcome this problem we need a

# cryptographic certificate

# X.509 CERTIFICATES

To solve our problem we need to **bind** a **private/public key pair** to an **entity identifier**.

---



# Binding Mathematics to the real world

## Mathematics

```
Private parameters

D = 776146563016167143639407273325675675258876383492304426355111898870465959460471326350389760890316583821937947364795351484367169212580231655993152129616761344530050725986827108953721558434941732038437591573053924491754510126364959317607757682447820942988657535279905936373267774215782242998025429793627422113

P = 98156973181620679211853421049873645267348559832027798575924056293799669247321177371558696106863284345557873306563227111028207859649430558922828531760940890

Q = 12589468470705273263461973192074899819962797457208232192214149362393003845360955098692591656972152326355810945154251491667000321296926470121976969660208161

Public parameters

N = 125894684707052732634619731920748998199627974572082321922123574411904987661324496409285016232564691118238530592626869039820889711716151616920802900114105503684028781673723882053400476571357110578897839565426525836523516069897401769498818606567641270578863484599103723172886277118009531924346591366796118168388195224114860554824660954136393556092086822258328661660329

E = 65537
```

RSA parameters

## Real World

www.banking.com

# Binding Mathematics to the real world

| Mathematics | Real World |
| --- | --- |
| $\text{Private} \longleftrightarrow \text{Public}$ | $\text{Public} \longleftrightarrow \text{Entity Identifier}$ |
| • **Public key**: $(e, N)$<br>• **Private key**: $d$<br>$d \equiv e^{-1} \mod \Phi(N)$ | • **Public key**: $(e, N)$<br>• **Entity**: $\text{google.com}$<br>$(e, N) \longleftrightarrow \text{google.com}$ |

# Binding Mathematics to the real world

This binding is obtained using a **digital certificate**.

## Binding Mathematics to the real world

---

$$\text{Private Key} \underbrace{\longleftrightarrow}_{\text{Maths}} \text{Public Key} \underbrace{\longleftrightarrow}_{\text{Certs}} \text{Entity Identifier}$$

**Q**: What is a certificate?

**Q**: What is a certificate? (1/3)

---

**A**: A **certificate** is a **sequence of bytes** that contains:

## **Q**: What is a certificate? (1/3)

---

**A**: A **certificate** is a **sequence of bytes** that contains:

- A **public key**, depending on the cryptography scheme used (e.g. DH, RSA).

## **Q**: What is a certificate? (1/3)

---

**A**: A **certificate** is a **sequence of bytes** that contains:

- A **public key**, depending on the cryptography scheme used (e.g. DH, RSA).
- Some information regarding the **entity associated with the public key**.

## **Q**: What is a certificate? (2/3)

---

**A**: A **certificate** is a **sequence of bytes** that contains:

## **Q**: What is a certificate? (2/3)

---

**A**: A **certificate** is a **sequence of bytes** that contains:

- A **signature** which assures the validity of the binding between the entity and the public key.

## **Q**: What is a certificate? (2/3)

---

**A**: A **certificate** is a **sequence of bytes** that contains:

- A **signature** which assures the validity of the binding between the entity and the public key.
- Some information regarding the **entity that has signed the certificate**.

## **Q**: What is a certificate? (2/3)

---

**A**: A **certificate** is a **sequence of bytes** that contains:

- A **signature** which assures the validity of the binding between the entity and the public key.
- Some information regarding the **entity that has signed the certificate**.
- Some information regarding the **way the certificate should be used**

# **Q**: What is a certificate? (3/3)

Everytime you see a **certificate**, remember:

1. **Public key**
2. **Entity**
3. **CA Entity**
4. **CA Signature**
5. **Usage information**

## **Q**: What is a certificate? (3/3)

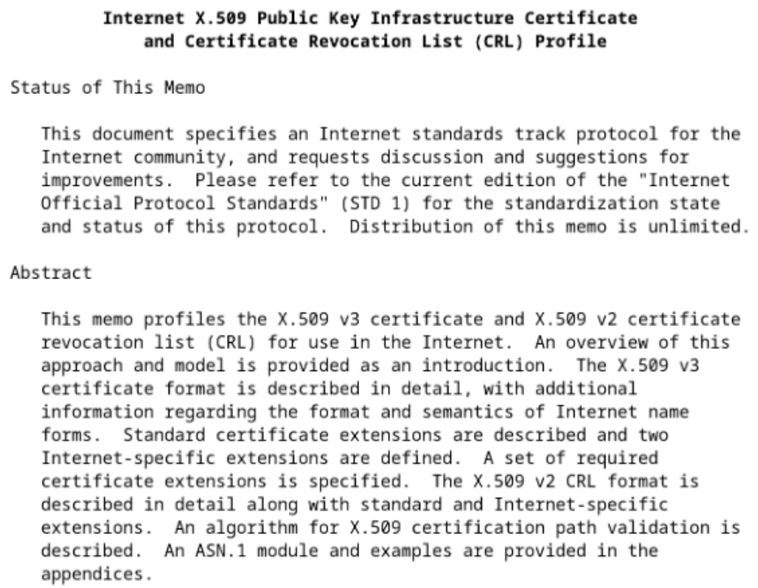
Everytime you see a **certificate**, remember:

1. **Public key**
2. **Entity**
3. **CA Entity**
4. **CA Signature**
5. **Usage information**

$$\mathrm{CA} \longrightarrow \mathrm{Certificate\ Authority}$$

# In **SSL/TLS** the certificate format used is the **X.509 standard** in RFC 5280 by the **PKIX group**.

---



**Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile**

Status of This Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Abstract

This memo profiles the X.509 v3 certificate and X.509 v2 certificate revocation list (CRL) for use in the Internet. An overview of this approach and model is provided as an introduction. The X.509 v3 certificate format is described in detail, with additional information regarding the format and semantics of Internet name forms. Standard certificate extensions are described and two Internet-specific extensions are defined. A set of required certificate extensions is specified. The X.509 v2 CRL format is described in detail along with standard and Internet-specific extensions. An algorithm for X.509 certification path validation is described. An ASN.1 module and examples are provided in the appendices.

https://www.rfc-editor.org/rfc/rfc5280

# Some fields within the **X.509** format

- **Version**
- **Serial Number**
- **Issuer**
- **Validity**
- **Subject**
- **Public Key**
- **Signature Algorithm**

Everytime we connect to an **HTTPs** server, we're also downloading a **chain of certificates**.


# OpenSSL commands

## Download **TLS certificate** given **TLS endpoint**

```
openssl s_client -connect leonardotamiano.xyz:443 -showcerts \
        </dev/null 2>/dev/null \
    | openssl x509 -outform PEM  > cert.pem
```

Several **formats** for saving **TLS certificates**:

- PEM (Privacy Enhanced Mail)
- CER
- DER
- PFX

# Example of a **PEM certificate**

```
-----BEGIN CERTIFICATE-----
MIIFLDCCBBSgAwIBAgISA8lPdFJehCKmEExu7mWqJMAoMA0GCSqGSIb3DQEBCwUA
MDIxCzAJBgNVBAYTAlVTMRYwFAYDVQQKEw1MZXQncyBFbmNyeXB0MQswCQYDVQQD
EwJSMzAeFw0yMjExMDkwMjMxNDNaFw0yMzAyMDcwMjMxNDJaMB4xHDAaBgNVBAMT
E2xlb25hcmRvdGFtaWFuby54eXowggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEK
AoIBAQC/ZvVG45Y8r0gr8huApsXswphzCwXyynlGM7whpe765noMePWEaeLYq5eu
FbadQKWXWHvLijlwDvApjhOd3FF+pjdO82aXSR6PURQM91sTdrF6xeKkrfv6C2QZ
SWFiiE17oam+O83InN7NWvXs+HNtoIwKLu3VfzEnll0AdtZA3MtBzcDLDtrBML4o
nmILI/0mFQ0l9iwRcDCfbdxFSiwTD68gN0jF7HadeQ7aVRhJ8W3Zucyla8KXGeh2
sMfj5IihIttUENYqw5AkFX8Hr0IzKN4ldm2J5cigFqiXriJpL1TLGqmsnt+3PoX5
gpgePdJV4nx873NlSwWKOhjtqkbpAgMBAAGjggJOMIICSjAOBgNVHQ8BAf8EBAMC
BaAwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMAwGA1UdEwEB/wQCMAAw
HQYDVR0OBBYEFJXXKkDnZZQogWRIoctZo61eVFoRMB8GA1UdIwQYMBaAFBQusxe3
WFbLrlAJQOYfr52LFMLGMFUGCCsGAQUFBwEBBEkwRzAhBggrBgEFBQcwAYYVaHR0
cDovL3IzLm8ubGVuY3Iub3JnMCIGCCsGAQUFBzAChhZodHRwOi8vcjMuaS5sZW5j
ci5vcmcvMB4GA1UdEQQXMBWCE2xlb25hcmRvdGFtaWFuby54eXowTAYDVR0gBEUw
QzAIBgZngQwBAgEwNwYLKwYBBAGC3xMBAQEwKDAmBggrBgEFBQcCARYaaHR0cDov
L2Nwcy5sZXRzZW5jcnlwdC5vcmcwggEEBgorBgEEAdZ5AgQCBIH1BIHyAPAAdgC3
Pvsk35xNunXyOcW6WPRsXfxCz3qfNcSeHQmBJe20mQAAAYRacZH3AAAEAwBHMEUC
ICNEQaHi9QBaLEgb+ehpt8soIRzyhPpGhE/SaQC6ex4vAiEAno7YntMpm+SK5dZg
Emgy0p+q2j51lLd7OAS0zBmg3HEAdgB6MoxU2LcttiDqOOBSHumEFnAyE4VNO9Ir
wTpXo1LrUgAAAYRacZIcAAAEAwBHMEUCIBR+CmQ7PsoDSxQ0ZIDueJP3Bu5PhGYz
Pj5mnp1aZGn+AiEA79a+Bv3LWD9RK07jSFOI7yXFyiJKIfjc/FJBpdxJ4IQwDQYJ
KoZIhvcNAQELBQADggEBAA08t5sZDF3ZL9jyYr/KCtRFbYgh/GgbCaFbxs9xvWoY
wMrxgTftBwxNDDTIdvBtnmLqaVvSDxJIbIKQAd0XNF2qduPUzEQ+yLazKLYJQxUw
sC+R/QP7+BFssF2WhYGbEKKNIiuDPjh45+dzYaI0lgUnAtNmLN+Vn8FEfIbo7KhU
yMF0zIfBRxO0q4qJJbRDH3x4mBlImwsiBUQVG+eZPsmhqyS0XB8iXYitXLVRtp1t
FcHmAt0R+R+t+4+2v0DiLGflSMIpwhh9Pk9RsrTfNIJfWia85/B/j66Ny2Gfj/bZ
TbzDsSK120xiVtRBT1j2dlxwrc9LBaeQ0zUapJwGL0M=
-----END CERTIFICATE-----
```

# **OpenSSL** command to read PEM certificate (1/5)

```
openssl x509 -in leonardotamiano.xyz -noout -text
```

# **OpenSSL** command to read PEM certificate (2/5)

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            03:c9:4f:74:52:5e:84:22:a6:10:4c:6e:ee:65:aa:24:c0:28
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Let's Encrypt, CN = R3
        Validity
            Not Before: Nov  9 02:31:43 2022 GMT
            Not After : Feb  7 02:31:42 2023 GMT
        Subject: CN = leonardotamiano.xyz
```

# **OpenSSL** command to read PEM certificate (3/5)

```
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:bf:66:f5:46:e3:96:3c:af:48:2b:f2:1b:80:a6:
                    c5:ec:c2:98:73:0b:05:f2:ca:79:46:33:bc:21:a5:
                    ee:fa:e6:7a:0c:78:f5:84:69:e2:d8:ab:97:ae:15:
                    b6:9d:40:a5:97:58:7b:cb:8a:39:70:0e:f0:29:8e:
                    13:9d:dc:51:7e:a6:37:4e:f3:66:97:49:1e:8f:51:
                    14:0c:f7:5b:13:76:b1:7a:c5:e2:a4:ad:fb:fa:0b:
                    64:19:49:61:62:88:4d:7b:a1:a9:be:3b:cd:c8:9c:
                    de:cd:5a:f5:ec:f8:73:6d:a0:8c:0a:2e:ed:d5:7f:
                    31:27:96:5d:00:76:d6:40:dc:cb:41:cd:c0:cb:0e:
                    da:c1:30:be:28:9e:62:0b:23:fd:26:15:0d:25:f6:
                    2c:11:70:30:9f:6d:dc:45:4a:2c:13:0f:af:20:37:
                    48:c5:ec:76:9d:79:0e:da:55:18:49:f1:6d:d9:b9:
                    cc:a5:6b:c2:97:19:e8:76:b0:c7:e3:e4:88:a1:22:
                    db:54:10:d6:2a:c3:90:24:15:7f:07:af:42:33:28:
                    de:25:76:6d:89:e5:c8:a0:16:a8:97:ae:22:69:2f:
                    54:cb:1a:a9:ac:9e:df:b7:3e:85:f9:82:98:1e:3d:
                    d2:55:e2:7c:7c:ef:73:65:4b:05:8a:3a:18:ed:aa:
                    46:e9
                Exponent: 65537 (0x10001)
```

# **OpenSSL** command to read PEM certificate (4/5)

```
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                95:D7:2A:40:E7:65:94:28:81:64:48:A1:CB:59:A3:AD:5E:54:5A:11
            X509v3 Authority Key Identifier:
                keyid:14:2E:B3:17:B7:58:56:CB:AE:50:09:40:E6:1F:AF:9D:8B:14:C2:C6
```

# **OpenSSL** command to read PEM certificate (5/5)

```
...
Signature Algorithm: sha256WithRSAEncryption
     0d:3c:b7:9b:19:0c:5d:d9:2f:d8:f2:62:bf:ca:0a:d4:45:6d:
     88:21:fc:68:1b:09:a1:5b:c6:cf:71:bd:6a:18:c0:ca:f1:81:
     37:ed:07:0c:4d:0c:34:c8:76:f0:6d:9e:62:ea:69:5b:d2:0f:
     12:48:6c:82:90:01:dd:17:34:5d:aa:76:e3:d4:cc:44:3e:c8:
     b6:b3:28:b6:09:43:15:30:b0:2f:91:fd:03:fb:f8:11:6c:b0:
     5d:96:85:81:9b:10:a2:8d:22:2b:83:3e:38:78:e7:e7:73:61:
     a2:34:96:05:27:02:d3:66:2c:df:95:9f:c1:44:7c:86:e8:ec:
     a8:54:c8:c1:74:cc:87:c1:47:13:b4:ab:8a:89:25:b4:43:1f:
     7c:78:98:19:48:9b:0b:22:05:44:15:1b:e7:99:3e:c9:a1:ab:
     24:b4:5c:1f:22:5d:88:ad:5c:b5:51:b6:9d:6d:15:c1:e6:02:
     dd:11:f9:1f:ad:fb:8f:b6:bf:40:e2:2c:67:e5:48:c2:29:c2:
     18:7d:3e:4f:51:b2:b4:df:34:82:5f:5a:26:bc:e7:f0:7f:8f:
     ae:8d:cb:61:9f:8f:f6:d9:4d:bc:c3:b1:22:b5:db:4c:62:56:
     d4:41:4f:58:f6:76:5c:70:ad:cf:4b:05:a7:90:d3:35:1a:a4:
     9c:06:2f:43
```

To deal with **SSL/TLS** we can use **pyOpenSSL**, a python wrapper around a subset of the famous OpenSSL library.

---

# Welcome to pyOpenSSL's documentation!

Release v22.2.0.dev (What's new?).

pyOpenSSL is a rather thin wrapper around (a subset of) the OpenSSL library. With thin wrapper we mean that a lot of the object methods do nothing more than calling a corresponding function in the OpenSSL library.

# Download certs data into JSON (1/4)

```
def establish_ssl_session(host, port=443):
    dst = (host.encode("utf-8"), port)
    s = socket.create_connection(dst)
    ctx = OpenSSL.SSL.Context(OpenSSL.SSL.SSLv23_METHOD)
    s = OpenSSL.SSL.Connection(ctx, s)
    s.set_connect_state()
    # -- sets the server-name-indication (sni) extension
    s.set_tlsext_host_name(dst[0])
    s.do_handshake()
    return s
```

(code/example_1_tls_certs.py)

# Download certs data into JSON (2/4)

```python
def get_tls_certs_chain(ssl_session):
    result = {}
    result["host"] = ssl_session.get_servername().decode("utf-8")
    result["tlsVersion"] = ssl_session.get_protocol_version_name()
    for (idx, cert) in enumerate(ssl_session.get_peer_cert_chain()):
        result[str(idx)] = dump_cert_data(cert)
    return result
```

(code/example_1_tls_certs.py)

# Download certs data into JSON (3/4)

```python
def dump_cert_data(x509):
    cert_data = {}
    cert_data["subject"] = x509.get_subject().commonName

    pubkey = x509.get_pubkey()
    pubkey_public_params = pubkey.to_cryptography_key().public_numbers()

    if pubkey.type() == OpenSSL.crypto.TYPE_EC:
        cert_data["pubkey"] = {
            "type": "Elliptic-Curve (EC)",
            "curve": pubkey_public_params.curve.name,
            "x": pubkey_public_params.x,
            "y": pubkey_public_params.y,
        }
    elif pubkey.type() == OpenSSL.crypto.TYPE_RSA:
        cert_data["pubkey"] = {
            "type": "RSA",
            "n": pubkey_public_params.n,
            "e": pubkey_public_params.e,
        }

    cert_data["issuer"] = x509.get_issuer().commonName

    return cert_data
```

(code/example_1_tls_certs.py)

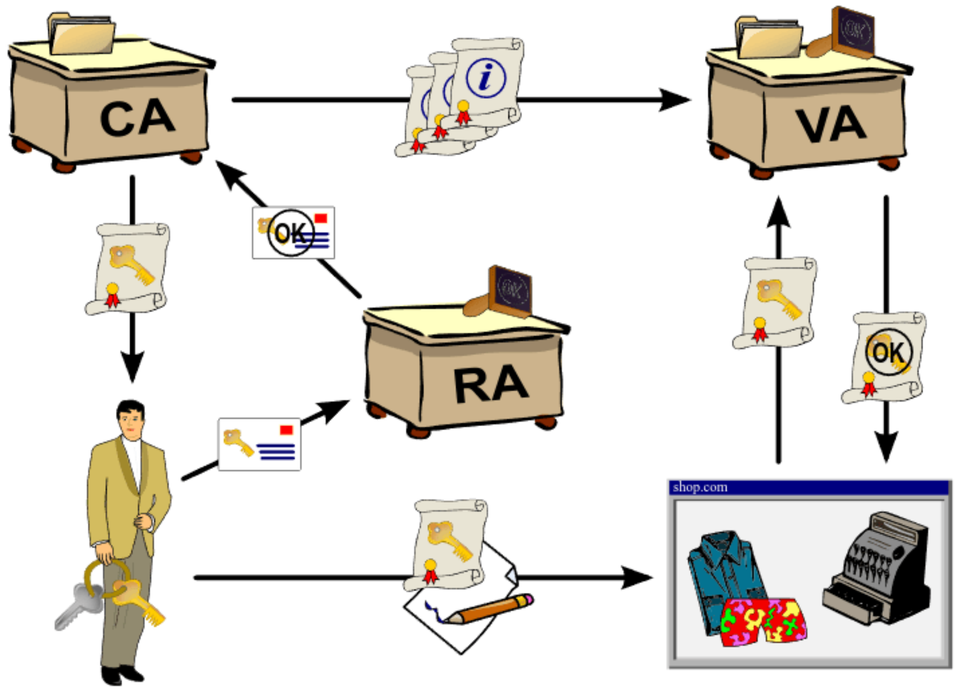
# Download certs data into JSON (4/4)

```
def main():
    if len(sys.argv) < 3:
        print(f"Usage: {sys.argv[0]} <host> <port>")
        exit()
    try:
        host = sys.argv[1]
        port = int(sys.argv[2])
    except ValueError as e:
        print(f"[ERR] - Failed to convert '{sys.argv[2]}' into an int")
        print(e)
        exit()

    ssl_session = establish_ssl_session(host, port)
    res = get_tls_certs_chain(ssl_session)
    print(json.dumps(res, indent=4))
```

# THE PUBLIC KEY INFRASTRUCTURE

Certificates, by themselves, **are not enough**.

We need an entire **infrastructure** which allows us to manage those certificates.




https://en.wikipedia.org/wiki/Public_key_infrastructure

## **PUBLIC-KEY INFRASTRUCTURE** (PKI)

---

With **PKI** we mean the sets of **protocols**, **policies** and the **cryptographic mechanism** that drive the management of **public key certificates**.

The PKI in the context of the **SSL/TLS** protocol is used **authenticate** server and client.

To understand the PKI we need to review:

- **Trust**
- **Certificate Authorities** (CAs)
- **The Hierarchy of the PKI**
- **PKI in SSL/TLS**

# TRUST IN THE PKI

A **certificate** binds a public key to an identity.

# But what is it that makes a certificate valid?

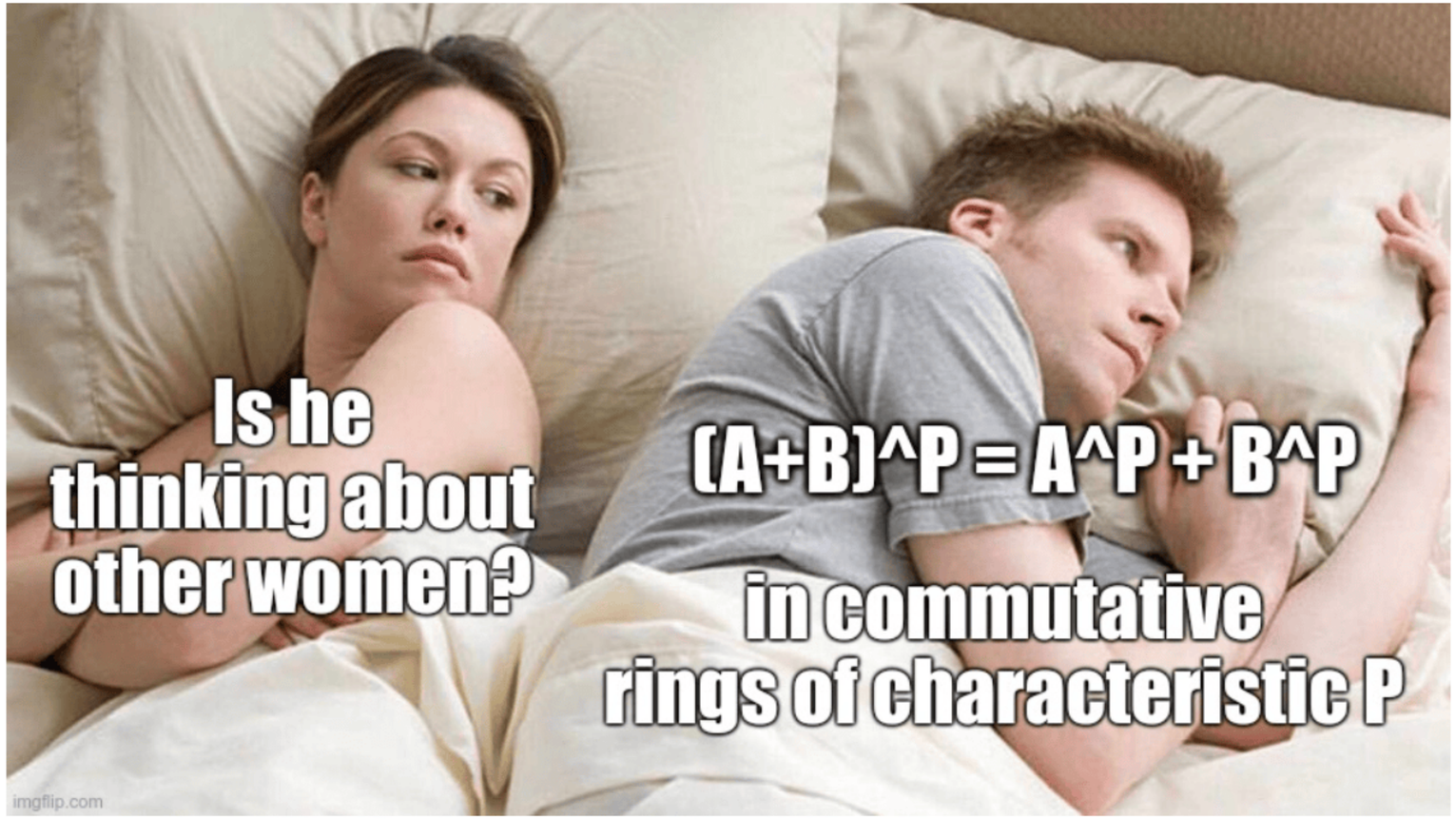
**Q**: What is it that makes a certificate valid?

**A:** The **signature** field of the certificate.

We can now talk about **trust**.

**TRUST**: tricky concept that carries significant emotional and rational meaning when discussed in general everyday life.

# Trust in personal relationship



## **Trust in the PKI** (1/5)

---

We have to put aside the common meaning of the word and only think about it in **narrower, more specific and technical terms**.

## **Trust in the PKI** (2/5)

In the context of PKI, we trust a certificate only if we **acknowledge** the entity that has signed the certificate.

# **Trust in the PKI** (3/5)

---

**Q**: How do we **acknowledge** the entity that has signed the certificate?

## **Trust in the PKI** (3/5)

---

**Q**: How do we **acknowledge** the entity that has signed the certificate?

**A**: By using an internal database, generally called a **trust store**.

## **Trust in the PKI** (4/5)

Certificates stored in the **trust store** are called

# root certificates

We trust anything that has been signed using a **private key** related to a **public key** contained in a **root certificate**.

## **Trust in the PKI** (5/5)

---

This **acknowledgement** can be done either directly or indirectly through a **chain of certificates**.

# CERTIFICATE AUTHORITIES

**Certificate Authorities**, also known as **CAs** are entities that **store**, **sign** and **issue** digital certificates such as **X.509** certificates.

In the **trust store** we store the **root certificates** of **trusted CAs**.

The idea is to **trust anything that is signed by a trusted CA**.

The role of each **CA** is to issue certificates only to the correct entities.

This can be done by introducing another entity known as a **RA**

$$\mathrm{RA} \longrightarrow \mathrm{Registration\ Authority}$$

## **Certificate Signing Requests** (1/2)

To generate a new certificate we have to issue a

$$\begin{aligned} \text{CSR} &\longrightarrow \text{Certificate} \\ &\longrightarrow \text{Signing} \\ &\longrightarrow \text{Requests} \end{aligned}$$

# **Certificate Signing Requests** (2/2)

Process of generating a new certificate:

# **Certificate Signing Requests** (2/2)

Process of generating a new certificate:

- Generation of personal private/public key pair.

# **Certificate Signing Requests** (2/2)

Process of generating a new certificate:

- Generation of personal private/public key pair.
- Generation of CSR which contains our public key pair as well as the entity identifier to be put in the certificate.

# **Certificate Signing Requests** (2/2)

Process of generating a new certificate:

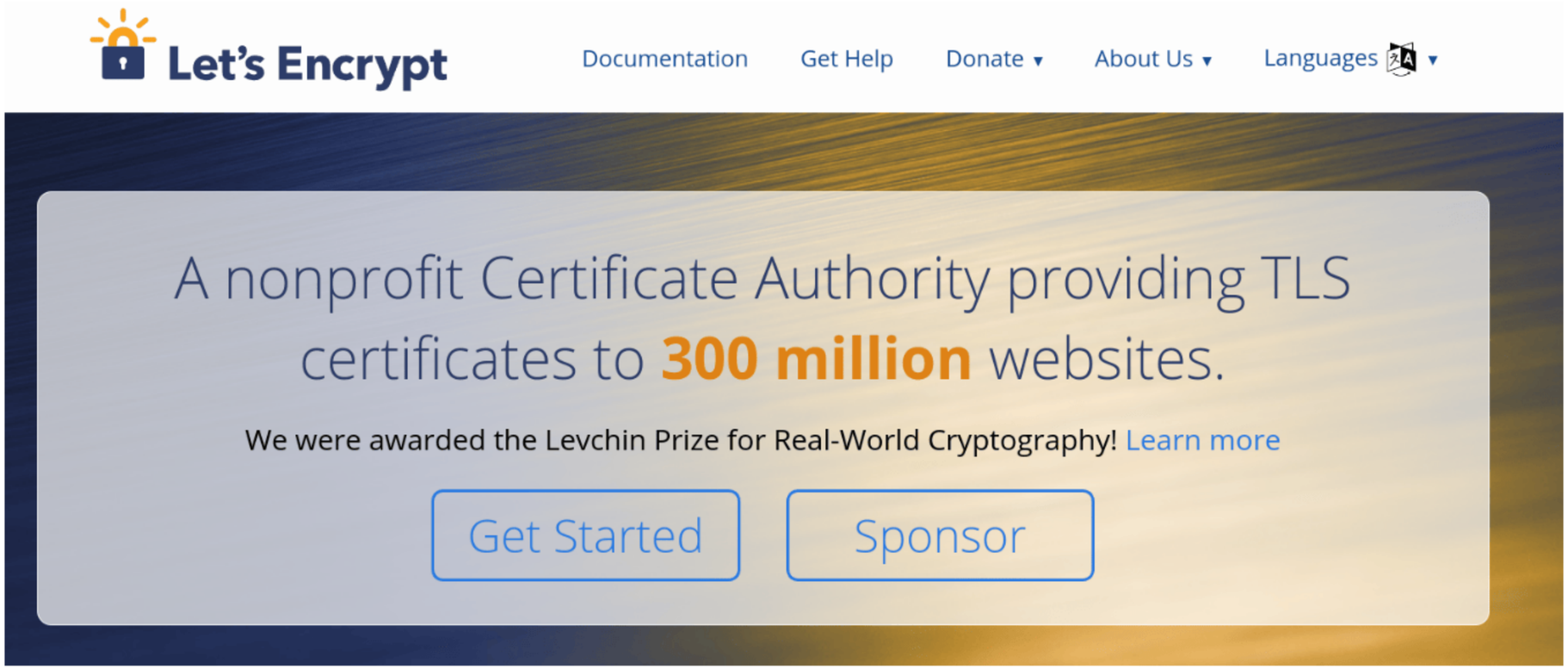
- Generation of personal private/public key pair.
- Generation of CSR which contains our public key pair as well as the entity identifier to be put in the certificate.
- The CSR is sent to a CA.

# **Certificate Signing Requests** (2/2)

Process of generating a new certificate:

- Generation of personal private/public key pair.
- Generation of CSR which contains our public key pair as well as the entity identifier to be put in the certificate.
- The CSR is sent to a CA.
- The CA, after having checked the authenticity of the entity, eventually by going through an **RA**, sends back the signed certificate.

# **Example of a CA:** Let's Encrypt



# THE HIERARCHY OF THE PKI

As it has already been established, everytime we connect to an **HTTPs server**, we download a

# chain of certificates

There are various types of certificates:

- Leaf certificates
- Intermediate CAs certificates
- Root CA certificates

We have that:

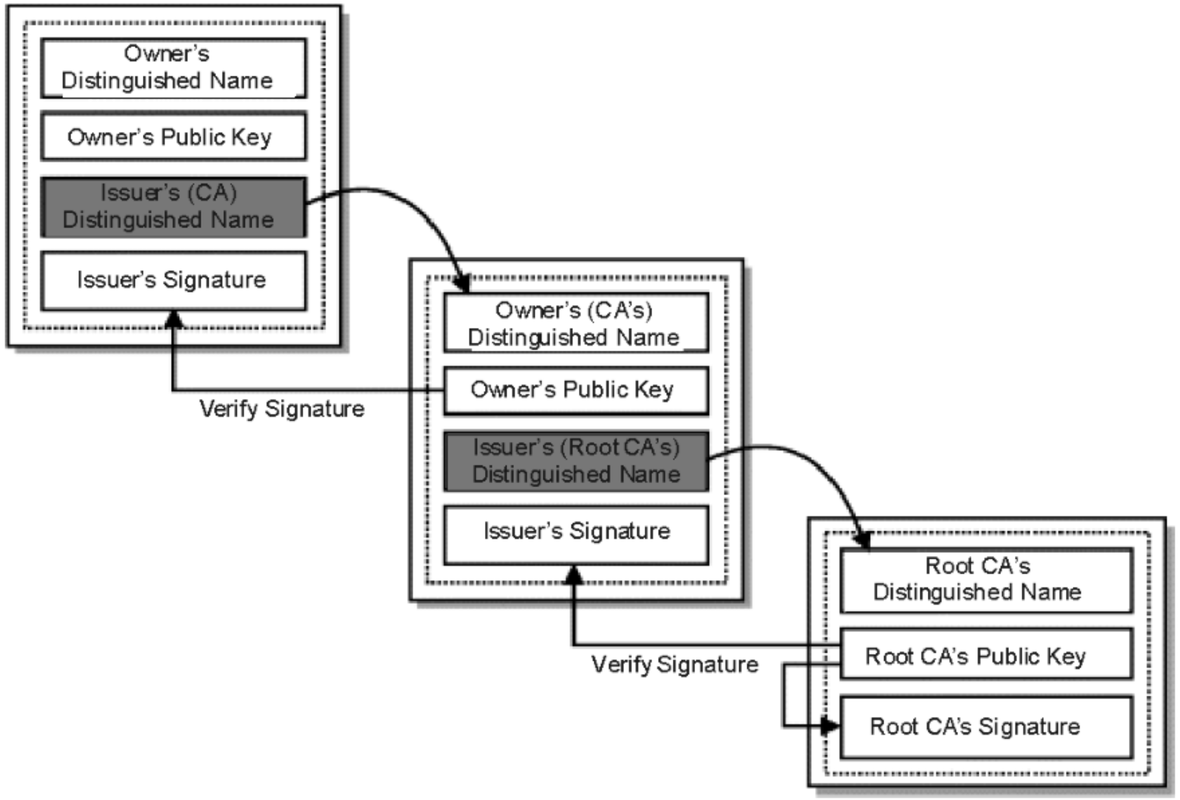
- **Leaf certificates** certified by **intermediate CAs certificates**.
- **Intermediate CAs certificates** certified by **root CA certificates**.
- **Root CA certificates certify their own identity** (!).

We have that:

- **Leaf certificates** certified by **intermediate CAs certificates**.
- **Intermediate CAs certificates** certified by **root CA certificates**.
- **Root CA certificates certify their own identity** (!).

**Remember**: We trust a root CA certificate only if it is physically saved in the **trust store**.

# Example of a 2 level chain



# Certificate Chains

List of certificates followed by one or more CA certificates, with the following properties:

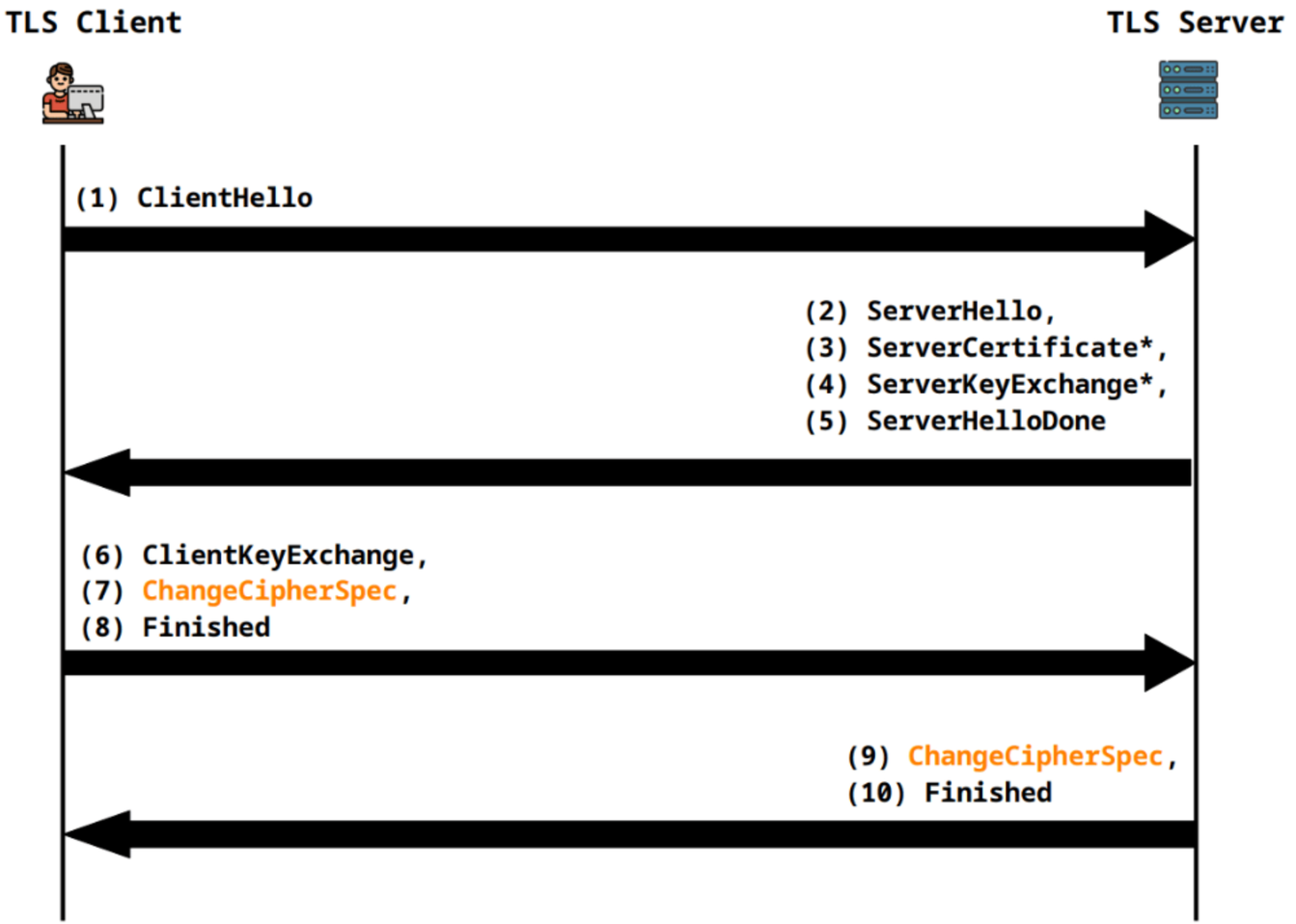
- The issuer of each certificate (except the last one) matches the subject of the next certificate in the list.
- Each certificate (except the last one) is supposed to be signed by the secret key corresponding to the next certificate in the chain.
- The last certificate in the list is a trust anchor: a certificate that you trust because it was delivered to you by some trustworthy procedure.

# PKI IN SSL/TLS

In **SSL/TLS** certificates are used during the TLS handshake to perform

- **server authentication**
- **client authentication**

# Server authentication ($\leq$ TLSv1.2)



# HANDS-ON 1: HTTPS WEB SERVER

Let us now understand how to setup a web server and add support for **HTTPs**.

---

In particular we will use the following software stack:

- **ubuntu 20.04**
- **nginx**
- **openssl**

For those interested at the end a **docker image** will be made available to play and tinker with.

# Starting point

```
user@47b491e10890:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.5 LTS
Release:        20.04
Codename:       focal
user@47b491e10890:~$ uname -a
Linux 47b491e10890 5.15.65-1-lts #1 SMP Mon,
   05 Sep 2022 15:40:39 +0000 x86_64 x86_64 x86_64 GNU/Linux
```

# Become **root**

```
user@47b491e10890:~$ su
Password:
root@47b491e10890:/home/user#
```

# Install **nginx** and **openssl**

```
apt-get install -y nginx openssl
```

Use **OpenSSL** to **generate RSA 2048 bit key** as well as a **self-signed certificate**

```
openssl req -x509 -nodes -days 365 \
        -newkey rsa:2048 \
        -keyout nginx-selfsigned.key \
        -out nginx-selfsigned.crt
```

During certificate generation the most important field is the **Common Name** field.

```
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:localhost
Email Address []:
```

The **Common Name** field has to be equal to the **Domain Name** of the server that is translated using the **DNS** protocol.

www.google.com

# Move files into `/etc/ssl/private/` folder.

```
mv /home/user/nginx-selfsigned.key /etc/ssl/private/nginx-selfsigned.key
mv /home/user/nginx-selfsigned.crt /etc/ssl/certs/nginx-selfsigned.crt
```

# Generate 2048 bit **Diffie-Hellman parameters**

```
openssl dhparam -out /etc/ssl/certs/dhparam.pem 2048
```

```
-----BEGIN DH PARAMETERS-----
MIIBCAKCAQEAwJbkDLiwMysR9epWPxz8nsuGYcidRyfxj5fldtKtEr3hs6+1niKp
057Qhky3rlwMBVxdJXtvTFf0U4IpFIxSD6E4o4m/Xb8VdyfMz8JEdwB1CPD+rmb8
FFPgelOdqGvVs8OQfUWSXwalLq5u7+319ZP1K32Q9awdW4gvI4iWMKQA+8zkEtrF
TsM4oZXcFsEH1yATLLnqubyYR3PWiB7mvyneEnEFwsSdAcyOoBIopiuKRWaqRxTG
v5XZajl0VIb4GiUouKYiSuYhtbeb+aF8Tyds0vsTu5fMuWt51UQOorXgp4QODzGg
lW/jXzBoSLneBqoXOOPq+qnvjKigptHGWwIBAg==
-----END DH PARAMETERS-----
```

(/etc/ssl/certs/dhparam.pem)

# Create config file for **OpenSSL integration**

```
ssl_protocols TLSv1.2;
ssl_prefer_server_ciphers on;
ssl_dhparam /etc/ssl/certs/dhparam.pem;
ssl_ciphers ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-SHA512:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA3
ssl_ecdh_curve secp384r1; # Requires nginx >= 1.1.0
ssl_session_timeout 10m;
ssl_session_cache shared:SSL:10m;
ssl_session_tickets off; # Requires nginx >= 1.5.9
resolver 8.8.8.8 8.8.4.4 valid=300s;
resolver_timeout 5s;
add_header X-Frame-Options DENY;
add_header X-Content-Type-Options nosniff;
add_header X-XSS-Protection "1; mode=block";
```

**(/etc/nginx/snippets/ssl-params.conf)**

# Edit **nginx virtual host** configuration file

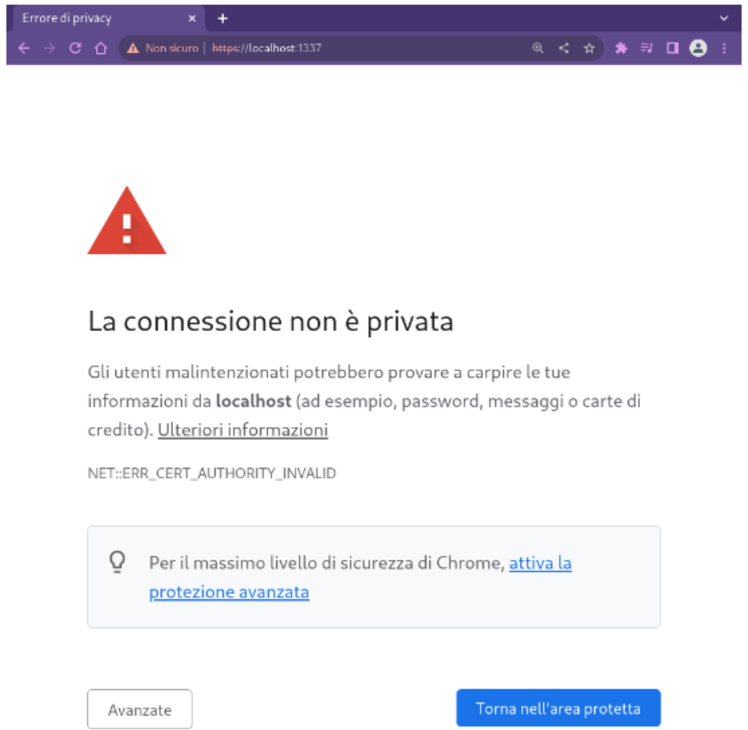
```
server {
        listen 80 ssl;
        listen [::]:80 ssl;

        ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
        ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;
        include snippets/ssl-params.conf;

        root /var/www/html;
        index index.html index.htm index.nginx-debian.html;
        server_name _;
        location / {
                try_files $uri $uri/ =404;
        }
}
```

**(/etc/nginx/sites-available/default)**

# Restart service

```
service nginx restart
```

# Now we can connect to our new HTTPs server.




# View TLS handshake with `curl`

```
$ curl -v -k https://localhost:1337
   Trying 127.0.0.1:1337...
 Connected to localhost (127.0.0.1) port 1337 (#0)
 ALPN: offers h2
 ALPN: offers http/1.1
 TLSv1.3 (OUT), TLS handshake, Client hello (1):
 TLSv1.3 (IN), TLS handshake, Server hello (2):
 TLSv1.2 (IN), TLS handshake, Certificate (11):
 TLSv1.2 (IN), TLS handshake, Server key exchange (12):
 TLSv1.2 (IN), TLS handshake, Server finished (14):
 TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
 TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
 TLSv1.2 (OUT), TLS handshake, Finished (20):
 TLSv1.2 (IN), TLS handshake, Finished (20):
 SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
 ALPN: server accepted http/1.1
 Server certificate:
  subject: C=AU; ST=Some-State; O=Internet Widgits Pty Ltd
  start date: Dec  2 19:41:12 2022 GMT
  expire date: Dec  2 19:41:12 2023 GMT
  issuer: C=AU; ST=Some-State; O=Internet Widgits Pty Ltd
  SSL certificate verify result: self signed certificate (18), continuing anyway.
```

# The following `Dockerfile` may be useful

```
# syntax=docker/dockerfile:1
FROM ubuntu:20.04

RUN mkdir /var/run/sshd
RUN chmod 0755 /var/run/sshd

RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y nginx openssl curl nano ssh passwd

RUN useradd --create-home --shell /bin/bash user
RUN echo 'user:password' | chpasswd
RUN echo 'root:password' | chpasswd

EXPOSE 22
EXPOSE 80

WORKDIR /home/user

CMD ["/usr/sbin/sshd", "-D"]
```

**(code/example_2_nginx_with_https/Dockerfile)**

# Build docker image

```
docker image build -t cns_nginx_with_https .
```

# Run docker

```
docker run -p 1337:80 -p 1338:22 cns_nginx_with_https
```

# HANDS-ON 2: TLS EXPORTERS

**RFC 5246** defines a mechanism for exporting **keying material** from TLS to an application or protoocl residing at an upper layer.

This material can then be used for generating secure data between clients and servers.

# Request for Comments: 5705

https://www.rfc-editor.org/rfc/rfc5705

**Keying Material Exporters for Transport Layer Security (TLS)**

Abstract

A number of protocols wish to leverage Transport Layer Security (TLS) to perform key establishment but then use some of the keying material for their own purposes. This document describes a general mechanism for allowing that.

The **TLS exporter** works within a TLS session and takes three input parameters:

- A label string
- A length value
- A context (optional)

Value computed by **TLS exporter** without context

---

$$\begin{aligned}&\mathrm{PRF}(\mathrm{master\_secret},\\&\qquad\mathrm{label},\\&\qquad\mathrm{client\_random} + \mathrm{server\_random})[\mathrm{length}]\end{aligned}$$

Value computed by **TLS exporter** with context

---

$$\begin{aligned}&\mathrm{PRF}(\mathrm{master\_secret},\\&\qquad\mathrm{label},\\&\qquad\mathrm{client\_random} + \mathrm{server\_random},\\&\qquad\mathrm{context\_length} + \mathrm{context\_value})[\mathrm{length}]\end{aligned}$$

Where **PRF** is the **TLS Pseudorandom Function** in use for the current session.

The output is a pseudorandom bit string of length bytes generated from the **master_secret** of the session.

We can test out this functionality with **pyOpenSSL** using the `export_keying_material` method.

**`export_keying_material`**(*label*, *olen*, *context=None*)

Obtain keying material for application use.

| | |
|---|---|
| **Param:** | label - a disambiguating label string as described in RFC 5705 |
| **Param:** | olen - the length of the exported key material in bytes |
| **Param:** | context - a per-association context value |
| **Returns:** | the exported key material bytes or None |

# Exported Keying Material with pyOpenSSL (1/3)

```
def establish_ssl_session(host, port=443):
    dst = (host.encode("utf-8"), port)
    s = socket.create_connection(dst)
    ctx = OpenSSL.SSL.Context(OpenSSL.SSL.SSLv23_METHOD)
    s = OpenSSL.SSL.Connection(ctx, s)
    s.set_connect_state()
    # -- sets the server-name-indication (sni) extension
    s.set_tlsext_host_name(dst[0])
    s.do_handshake()
    return s
```

(code/example_3_export_keying_material.py)

# Exported Keying Material with pyOpenSSL (2/3)

```
def get_keying_material(ssl_session, length, label=b"EXPORTER_EXPERIMENTAL", context=None):
    returned_bytes = ssl_session.export_keying_material(label, length, context)
    return returned_bytes
```

# Exported Keying Material with pyOpenSSL (3/3)

```python
def main():
    if len(sys.argv) < 4:
        print(f"Usage: {sys.argv[0]} <host> <port> <length>")
        exit()
    try:
        host = sys.argv[1]
        port = int(sys.argv[2])
        length = int(sys.argv[3])
    except ValueError as e:
        print(f"[ERR] - Failed to convert argument into an int")
        print(e)
        exit()

    ssl_session = establish_ssl_session(host, port)
    keying_material = get_keying_material(ssl_session, length)
    if not keying_material:
        print("[ERR] - Failed to export keying material")
        exit()
    print(f"From TLS connection to ({host}, {port}) computed following keying material:")
    print(f"  Bytes: {keying_material}")
    print(f"  Hex: {keying_material.hex()}")
```

(code/example_3_export_keying_material.py)