network/tls

Exemples

socket

El primer que farem es generar un certificat pel servidor:

```
box@tls:~$ openssl req -new -x509 -days 365 -nodes -out cert.pem \
> -keyout key.pem \
> -subj "/C=ES/ST=Barcelona/O=XTEC/CN=xtec.dev"
...+......+...+......+......+...+++++++++++++++++++++++++++++++++++++++++++++++++
...+......++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*....
....+.....+.+........+.+......+.....+......+...++++++++++++++++++++++++++++++++
.....+.+......+..+..........+..+.+.....+...+.+.........+...........+............
...........+....+..+....+...+.....+......+.++++++++++++++++++++++++++++++++++++
+...+.....+.+...........++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+..+.+................+...+.+..............+.....+...+.......+..............+
.....+......+...+..+...+......+..................+..+....+...........+.+..+.+..
++++++++++++++++++++++++++++++++++
-----
box@tls:~$
box@tls:~$ ls *.pem
cert.pem  key.pem
box@tls:~$
```

`CN` és el nom de l'anfitrió.

Ara creem un servidor:

server.py client.py

server.py > ...

```python
from socket import socket, AF_INET, SOCK_STREAM
from ssl import SSLContext, PROTOCOL_TLS_SERVER

context = SSLContext(PROTOCOL_TLS_SERVER)
context.load_cert_chain("cert.pem","key.pem")

with socket(AF_INET, SOCK_STREAM) as server:
    server.bind(('0.0.0.0',8443))
    server.listen(1)
    with context.wrap_socket(server, server_side=True) as tls:
        connection, address = tls.accept()
        print(f'Connected by {address}\n')

        data = connection.recv(1024)
        print(f'Client says: {data}')

        connection.sendall(b"You're welcome")
```

I el client que ha d'acceptar el certificat del servidor:

server.py client.py

client.py > ...

```python
from socket import create_connection
from ssl import SSLContext, PROTOCOL_TLS_CLIENT

context = SSLContext(PROTOCOL_TLS_CLIENT)
context.load_verify_locations("cert.pem")

with create_connection(('127.0.0.1',8443)) as client:
    with context.wrap_socket(client, server_hostname="xtec.dev") as tls:
        print(f'Using {tls.version}\n')
        tls.sendall(b"Hello, world!")

        data = tls.recv(1024)
        print(f"Server says {data}")
```

Pots veure que SSL és completament indiferent al protocol subjacent.

Es limitat a crear un wrapper al voltant del socket i enviar i rebre tot el que li passen, això sí, encriptat i protegit:

```
box@tls:~$ python3 server.py
Connected by ('127.0.0.1', 51754)

Client says: b'Hello, world!'
box@tls:~$
```

```
box@tls:~$ python3 client.py
Using <bound method SSLSocket.version of <ssl.SSLSo
F_INET, type=SocketKind.SOCK_STREAM, proto=6, laddr
7.0.0.1', 8443)>>

Server says b"You're welcome"
box@tls:~$
```

Si modifiques el client, i no acceptes el certificat del servidor ws produirà un error perquè el certificat és autofirmat.

És com si tu mateix et firmessis el DNI:

client.py > ...

```
1  from socket import create_connection
2  from ssl import SSLContext, PROTOCOL_TLS_CLIENT
3
4  context = SSLContext(PROTOCOL_TLS_CLIENT)
5  #context.load_verify_locations("cert.pem")
6
7  with create_connection(('127.0.0.1',8443)) as client:
```

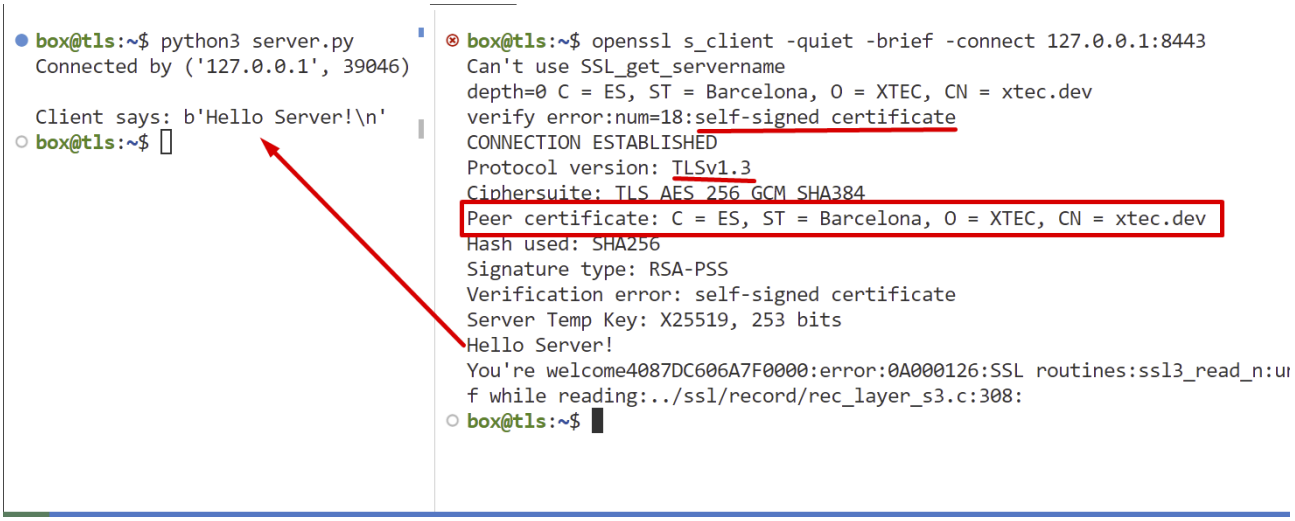
COMENTEM LA LINIA

La connexió no es pot establir:

```
  File "/home/box/server.py", line 11, in <m
odule>
    connection, address = tls.accept()
  File "/usr/lib/python3.10/ssl.py", line 13
88, in accept
    newsock = self.context.wrap_socket(newso
ck,
  File "/usr/lib/python3.10/ssl.py", line 51
3, in wrap_socket
    return self.sslsocket_class._create(
  File "/usr/lib/python3.10/ssl.py", line 10
71, in _create
    self.do_handshake()
  File "/usr/lib/python3.10/ssl.py", line 13
42, in do_handshake
    self._sslobj.do_handshake()
ssl.SSLError: [SSL: TLSV1_ALERT_UNKNOWN_CA]
tlsv1 alert unknown ca (_ssl.c:1007)
box@tls:~$
```

```
box@tls:~$ python3 client.py
Traceback (most recent call last):
  File "/home/box/client.py", line 8, in <module>
    with context.wrap_socket(client, server_hostname="xtec.dev") as
  File "/usr/lib/python3.10/ssl.py", line 513, in wrap_socket
    return self.sslsocket_class._create(
  File "/usr/lib/python3.10/ssl.py", line 1071, in _create
    self.do_handshake()
  File "/usr/lib/python3.10/ssl.py", line 1342, in do_handshake
    self._sslobj.do_handshake()
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] cert
e verify failed: self-signed certificate (_ssl.c:1007)
box@tls:~$
```

També ens podem connectar amb el client `openssl`:



Python fa servir la llibreria OpenSSL que tenim instal.lada al sistema operatiu.

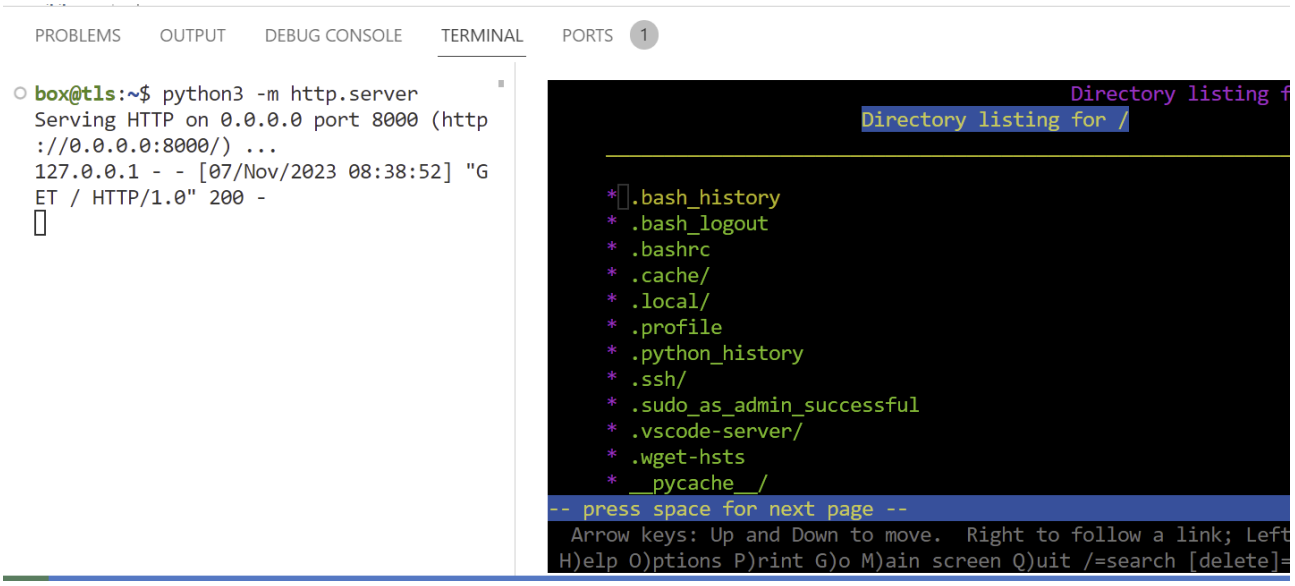
Per tant el resultat és l'esperat.

HTTP

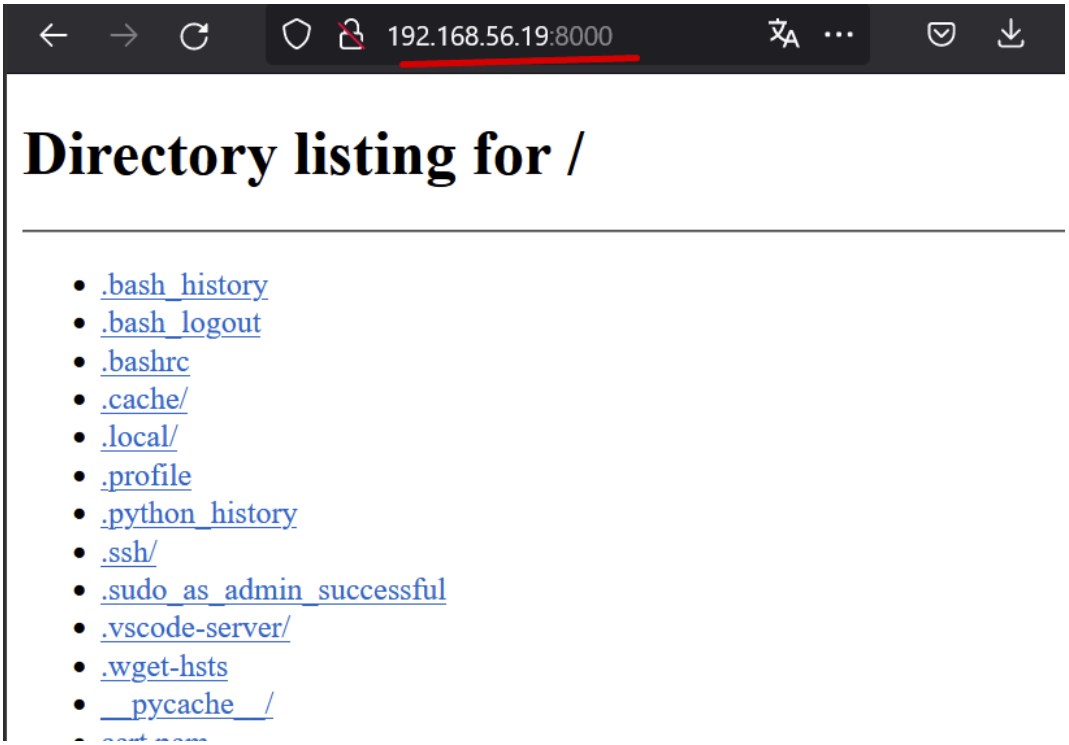
Un dels usos més habituals de TLS és per xifrar connexions HTTP.

Amb Python és molt fàcil arrencar un servidor http al port 8000 que escolta en totes les interficies disponibles de la màquina:

Pots veure que el servidor et permet navegar per tot el sistema de fitxers des d'on s'ha executat l'ordre:



Com que el servidor està escoltant desde la interfície “0.0.0.0” (que vol dir totes), també pots accedir desde el navegador de la màquina anfitriona:



Però com podem xifrar la connexió?

Doncs hem de fer un “wrap” del socket que fa servir `http.server` :

http-server.py

```python
from http.server import HTTPServer, SimpleHTTPRequestHandler
from ssl import SSLContext, PROTOCOL_TLS_SERVER

context = SSLContext(PROTOCOL_TLS_SERVER)
context.load_cert_chain("cert.pem","key.pem")

server = HTTPServer(('0.0.0.0',8443), SimpleHTTPRequestHandler)

with context.wrap_socket(server.socket, server_side= True) as tls_socket:
    server.socket = tls_socket
    server.serve_forever()

```

Podem arrencar el nostre servidor:

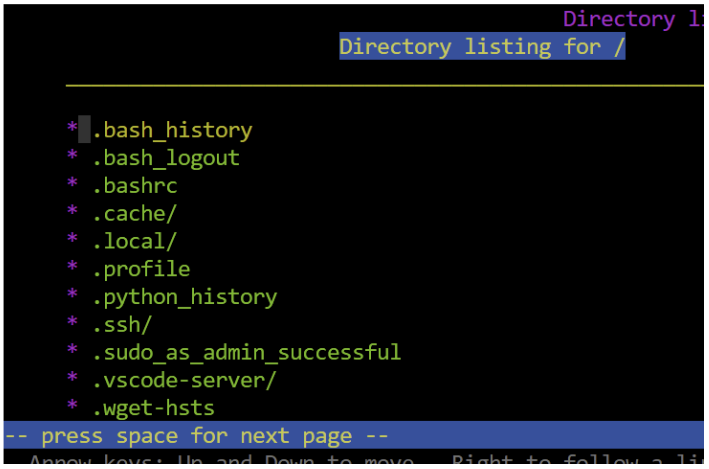
```
box@tls:~$ python3 http-server.py
```

```
box@tls:~$ lynx https://127.0.0.1:8443
```

Podem veure que funciona (si acceptem el certificat autofirmat):

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 3

```
box@tls:~$ python3 http-server.py
127.0.0.1 - - [07/Nov/2023 09:08:25] "GE
T / HTTP/1.0" 200 -
```



També podem veure que podem interactuar amb el servidor amb el client `openssl` :

```
box@tls:~$ openssl s_client -quiet -brief -connect 127.0.0.1:8443
Can't use SSL_get_servername
depth=0 C = ES, ST = Barcelona, O = XTEC, CN = xtec.dev
verify error:num=18:self-signed certificate
CONNECTION ESTABLISHED
Protocol version: TLSv1.3
Ciphersuite: TLS_AES_256_GCM_SHA384
Peer certificate: C = ES, ST = Barcelona, O = XTEC, CN = xtec.dev
Hash used: SHA256
Signature type: RSA-PSS
Verification error: self-signed certificate
Server Temp Key: X25519, 253 bits
GET / HTTP/1.1

HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.10.12
Date: Tue, 07 Nov 2023 09:37:28 GMT
Content-type: text/html; charset=utf-8
```

PostgreSQL

A DAW-2 - Bases de dades varem apendre a treballar amb la base de dades PostgreSQL.

```
box@tls:~$ mkdir data
box@tls:~$ docker run --rm --name db -p 5432:5432 \
> -v $PWD/data:/var/lib/postgresql/data \
> -e POSTGRES_PASSWORD=password -d postgres:15.4
9850035838e7d6587364bdb8a0b8957ec7ebd3000671a64110e7ed01adec76c7
box@tls:~$
box@tls:~$ docker exec -it db bash
root@9850035838e7:/#
root@9850035838e7:/# psql -U postgres
psql (15.4 (Debian 15.4-2.pgdg120+1))
Type "help" for help.

postgres=# \l
                                                 List of databases
   Name    |  Owner   | Encoding |  Collate   |   Ctype    | ICU Locale | Locale Provider
-----------+----------+----------+------------+------------+------------+-----------------
 postgres  | postgres | UTF8     | en_US.utf8 | en_US.utf8 |            | libc
 template0 | postgres | UTF8     | en_US.utf8 | en_US.utf8 |            | libc
           |          |          |            |            |            |
 template1 | postgres | UTF8     | en_US.utf8 | en_US.utf8 |            | libc
           |          |          |            |            |            |
(3 rows)

postgres=# quit
root@9850035838e7:/# exit
exit
box@tls:~$
```

Per connectar-te a la base de dades faras servir l'adaptador postgres `psycopg2` per a Python:

```
$ sudo apt install -y libpq-dev
$ pip install psycopg2
```

db.py 1

db.py > ...

```python
import psycopg2

connection = psycopg2.connect(host="127.0.0.1",
                              port=5432,
                              user="postgres",
                              password="password")
connection.autocommit = True

cursor = connection.cursor()
cursor.execute('SELECT %s as connected;', ('Connection to postgres successful!',))
print(cursor.fetchone())
```

Pots veure que et pots connectar a la base de dades:

```
box@tls:~$ python3 db.py
('Connection to postgres successful!',)
box@tls:~$
```

TLS es pot aplicar a qualsevol protocol client-servidor, també a la conexió entre el nostre client i la base de dades PostgreSQL.

db.py 1

db.py > ...

```python
import psycopg2

connection = psycopg2.connect(host="127.0.0.1",
                              port=5432,
                              user="postgres",
                              password="password",
                              sslmode='require')
connection.autocommit = True

cursor = connection.cursor()
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS 3

```
Traceback (most recent call last):
  File "/home/box/db.py", line 3, in <module>
    connection = psycopg2.connect(host="127.0.0.1",
  File "/home/box/.local/lib/python3.10/site-packages/psycopg2/__init__.py", lin
e 122, in connect
    conn = _connect(dsn, connection_factory=connection_factory, **kwasync)
psycopg2.OperationalError: connection to server at "127.0.0.1", port 5432 failed
: server does not support SSL, but SSL was required

box@tls:~$
```

Hem de crear uns certificats pel servidor postgres:

```
box@tls:~$ docker exec db openssl req -new -x509 -days 365 -nodes -text -out
 /var/lib/postgresql/data/server.crt -keyout /var/lib/postgresql/data/server
.key -subj="/CN=db.xtec.dev"
........+......+.......+.........+++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++*.....+....+......+...+...........+......+.+..........
....++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*.....+
.+..+......+.+...........+..........+........+.+...+......+..+...+..........
+......+...........+.......+++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++
...+..+..........++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++*.++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*..
.....+...+...+..+..........+..+......+..........+......+......+.....+...+.+.
..+........+...+.......+.....................+.+........+...+............+
............+.+..++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++
-----
box@tls:~$
```

Ja podem parar el contenidor `db` i tornar a executar un altre contenidor amb l'opció ssl=on:

```
box@tls:~$ docker stop db
db
box@tls:~$ docker run --rm --name db -p 5432:5432 -v $PWD/data:/var/lib/postgresql
/data -e POSTGRES_PASSWORD=password -d postgres:15.4 -c ssl=on
5ca7e6fc97b88a65a475b57ca07c79b99f3e88acbe3a757ecf3781d5ffbc1fee
box@tls:~$
```

L'applicació db.py ja es pot connectar mitjançant una connexió xifrada tls:

```
box@tls:~$ python3 db.py
('Connection to postgres successful!',)
box@tls:~$
```

Podem utilitzar el client openssl per veure la connexió TLS:

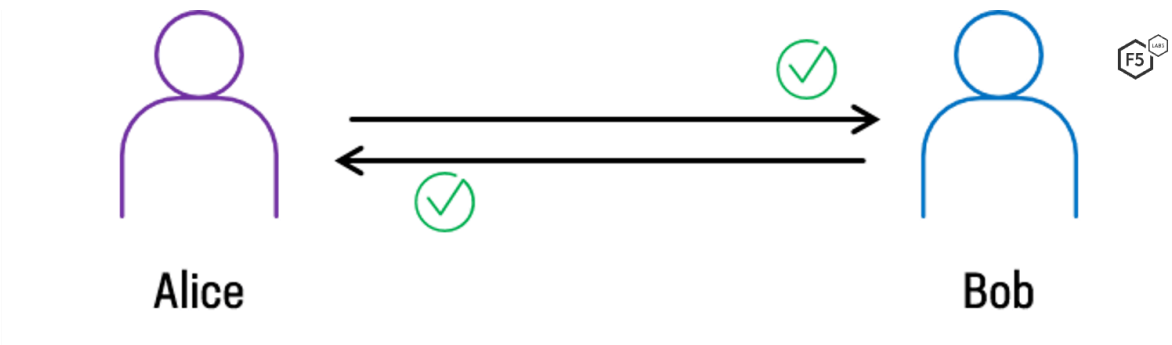
```
box@tls:~$ openssl s_client -quiet -brief -starttls postgres -connect 127.0.0.1:54
32
Can't use SSL_get_servername
depth=0 CN = db.xtec.dev
verify error:num=18:self-signed certificate
CONNECTION ESTABLISHED
Protocol version: TLSv1.3
Ciphersuite: TLS_AES_256_GCM_SHA384
Peer certificate: CN = db.xtec.dev
Hash used: SHA256
Signature type: RSA-PSS
Verification error: self-signed certificate
Server Temp Key: ECDH, prime256v1, 256 bits
```

mTLS

Important!. S'ha de canviar a Python

Mutual Transport Layer Security (mTLS) és un procés que estableix una connexió TLS xifrada en la qual ambdues parts utilitzen certificats digitals X.509 per autenticar-se mútuament.

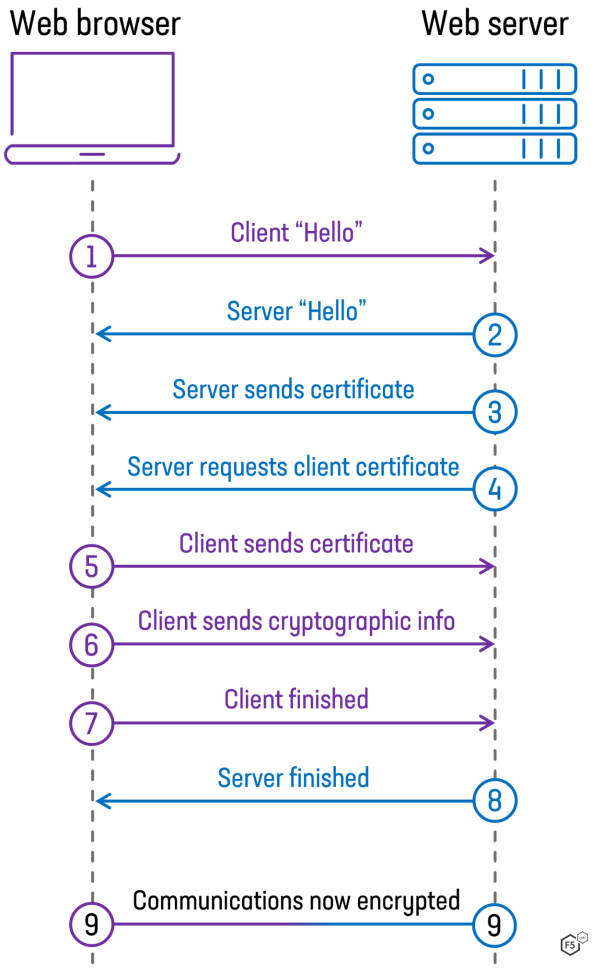
Diguem l'Alícia i el Bob*fer* necessita una autenticació més forta. Potser estan tractant amb transferències financeres de gran valor o enviant informació sensible. Potser hi ha una bona probabilitat que un actor maliciós pugui robar la contrasenya d'Alice i ocupar el seu lloc. Bob pot voler o*necessitat* per comprovar criptogràficament que l'Alícia és realment qui diu que és. En aquest cas, potser voldrien*mútuament* autenticar-se mútuament. És a dir, l'Alice no només verifica la identitat d'en Bob, sinó que Bob també autentica la identitat d'Alice. A través del web, això es pot realitzar mitjançant certificats digitals i TLS mutus.



Autenticació bidireccional, o mútua, com es veu quan s'utilitza mTLS, en què tant el client (Alice) com el servidor (Bob) s'autentiquen mútuament.

mTLS no és un protocol nou. L'autenticació mútua forma part de l'estàndard TLS i ha format part de l'especificació des que es va anomenar Secure Sockets Layer (SSL). Qualsevol servidor web que utilitzi TLS per assegurar el seu trànsit hauria de ser capaç d'autenticar-se mútuament. Per tal d'implementar l'autenticació mútua, el servidor ha de demanar específicament al client el seu certificat, però, la majoria dels servidors web no estan configurats per fer-ho de manera predeterminada.

La següent figura mostra una versió simplificada d'una connexió TLS 1.2, coneguda com a "encaixada de mans". La majoria de llocs web d'Internet salten els passos 4 i 5. Per als llocs i serveis que necessiten realitzar una autenticació mútua, el servidor, al pas 4, enviarà un missatge al client demanant-li que proporcioni un certificat (a més de dir-li de quines CA accepta certificats).



Passos durant una connexió de mans TLS 1.2 entre el client i el servidor quan es configura l'autenticació TLS mútua.

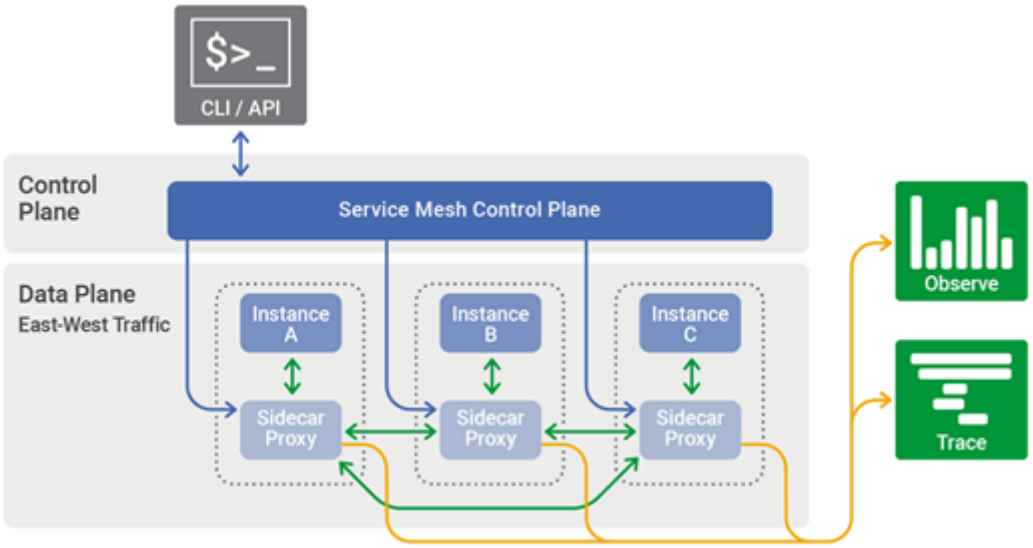
Entorns de malla de servei

Moltes aplicacions modernes utilitzen a arquitectura de microserveis que separa els components de l'aplicació en serveis discrets que s'executen en diversos servidors, sovint dins de contenidors. Per tal d'intercanviar dades i càlculs, aquests serveis s'han de comunicar a través de la xarxa. Compareu això amb les aplicacions monolítices tradicionals que realitzen tota la comunicació en memòria. Els entorns al núvol, amb una escala aparentment il·limitada i una automatització flexible, proporcionen un lloc ideal per desplegar aplicacions basades en microserveis i, tot i que el

TLS estàndard pot xifrar la comunicació entre aquests microserveis, encara deixa oberta la possibilitat que algú manipuli qualsevol dels serveis.

L'ús d'una "malla de servei" permet als propietaris d'aplicacions governar el trànsit de servei a servei. Una malla de servei pot centralitzar la gestió dels microserveis i us permet controlar els dos extrems de la connexió, utilitzant mTLS per xifrar i autenticar les dades al cable.

La figura següent mostra les interaccions d'alt nivell de les sessions TLS autenticades mútuament en un entorn de malla de servei.



Entorns de malla de servei que utilitzen mTLS per xifrar i autenticar serveis.

Exemple

Anem a veure un exemple de mTLS d'ús del navegador web cURL (el client) per connectar-se a un servidor web Node.js (el servidor) que serveix al nom DNS*localhost*. Al fer-ho:

- El client validarà que el servidor és de confiança per oferir contingut per al nom DNS*localhost*
- El servidor validarà que el client és conegut, és a dir, l'autenticarà

El primer pas és crear una autoritat de certificació (CA) en la qual confien tant el client com el servidor. La CA és només una clau pública i privada amb la clau pública embolicada en un certificat X.509 autofirmat. L'ordre que fem servir per fer-ho és:

```
openssl req  -new -x509  -nodes -days 365 -subj '/CN=xtec'  -keyout ca.key
-out ca.crt
```

Això genera dos fitxers,*ca.key* i*ca.crt*,en el Format PEM (codificació base64 de la clau privada i del certificat X.509 respectivament).

Mirant el*openssl req* documentació, veiem que el-*nou* i-*x509* les opcions permeten la creació d'un certificat arrel CA X.509 autofirmat. El*nodes* L'opció (Sense DES) desactiva la seguretat de la clau privada amb una contrasenya; aquesta opció és opcional. El*subjecte* l'opció proporciona la identitat de l'AC; en aquest cas el Nom Comú (NC) de*l meu-ca*. La resta d'opcions s'explica per si mateix.

Podem donar la volta i inspeccionar el certificat mitjançant l'ordre següent:

```
openssl x509  --in ca.crt -text --noout
```

Les opcions aquí s'expliquen per si mateixes tal com es documenta a la pàgina*openssl x509* documentació.

Mirant la sortida, podem confirmar una sèrie de coses:

```
Certificate:
      Data:
      Version: 3 (0x2)
      Serial Number:
            09:1f:11:80:92:69:8d:12:7f:1c:fa:77:00:24:67:4e:15:1a:82:c6
      Signature Algorithm: sha256WithRSAEncryption
      Issuer: CN = xtec
      Validity
            Not Before: Nov  9 21:53:39 2022 GMT
            Not After : Nov  9 21:53:39 2023 GMT
      Subject: CN = xtec
      Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                  Public-Key: (2048 bit)
                  Modulus:
                  Exponent: 65537 (0x10001)
      X509v3 extensions:
            X509v3 Subject Key Identifier:
                  F8:77:0C:F0:F4:60:D9:33:A8:67:B4:F5:43:A6:3D:13:F0:34:5A:AE
            X509v3 Authority Key Identifier:
                  F8:77:0C:F0:F4:60:D9:33:A8:67:B4:F5:43:A6:3D:13:F0:34:5A:AE
            X509v3 Basic Constraints: critical
                  CA:TRUE
      Signature Algorithm: sha256WithRSAEncryption
      Signature Value:
```

- Tant el*Assignatura* i*Emissor* tenir el valor*CN = xtec;* això indica que aquest certificat està autofirmat
- El*Validesa* indica que el certificat té una validesa d'un any
- El*X509v3 Restriccions bàsiques* valor*CA: CERT* indiqueu que aquest certificat es pot utilitzar com a CA, és a dir, es pot utilitzar per signar certificats

A continuació creem la clau i el certificat del servidor; començant per la clau:

```
openssl genrsa  -out server.key 2048
```

Les opcions aquí s'expliquen per si mateixes tal com es documenta a la pàgina*openssl genrsa* documentació.

Recordeu que el nostre objectiu aquí és crear un certificat de servidor per al nom DNS*localhost* signat per la CA. Ara creem una sol·licitud de signatura de certificat (CSR) amb el nom comú (CN)*localhost*:

```
openssl req  -new -key server.key -subj '/CN=localhost'  -out server.csr
```

Utilitzant la CSR, la CA (realment utilitzant la clau i el certificat de la CA) crea el certificat signat:

```
openssl x509  -req -in server.csr -CA ca.crt  -CAkey ca.key  -CAcreateserial
-days 365  -out server.crt
```

La sortida és el certificat del servidor signat,*servidor.crt*, en format PEM.

La majoria de les opcions són familiars (des de dalt) o s'expliquen per si mateixes, tal com es documenta a la pàgina*openssl x509* documentació. L'única excepció és la*CAcreateserial*opció que gestiona un fitxer creat recentment,*ca.srl*, que permet que cada certificat creat per aquesta CA tingui un número de sèrie únic.

Com abans, inspeccionem el certificat mitjançant l'ordre següent:

```
openssl x509  --in server.crt -text  --noout
```

Mirant la sortida, podem confirmar una sèrie de coses:

- El*Emissor* té el valor*CN = xtec;* això indica que aquest certificat està signat per la*xtec* autoritat de certificació
- El*Validesa* indica que el certificat té una validesa d'un any
- El*Assignatura* té el valor*CN = host local*; això indica que aquest certificat es pot enviar a un client per validar que el servidor és de confiança per oferir contingut per al nom DNS*localhost*

Bàsicament repetim el procés per crear la clau i el certificat del client; començant per crear la clau del client:

```
openssl genrsa  -out client.key 2048
```

Creació del CSR amb el nom comú arbitrari de*client:*

```
openssl req  -new -key client.key -subj '/CN=client' -out client.csr
```

I finalment la creació del certificat del client:

```
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial
-days 365 -out client.crt
```

> **Nota**.Si inspeccioneu aquest certificat, observareu que el*Número de sèrie* és realment diferent del certificat del servidor.

Una observació és que tant els certificats de servidor com de client són certificats X.509 v1 més simples; el certificat CA, però, és un certificat X.509 v3. Això es deu al fet que OpenSSL crea automàticament certificats autofirmats X.508 v3 (certificat CA) i no hem subministrat cap extensió v3 en signar els certificats de servidor i client (utilitzant el*fitxer ext* i*extensions* opcions).

Nota.L'ús de diverses extensions X.509 v3 està fora de l'abast d'aquest document.

Amb totes les nostres claus i certificats (ca, servidor i client) creats podem configurar el nostre servidor i client.

Instal·leu npm i creeu un projecte nou:

```
sudo apt install -y npm
npm init
```

El servidor és el bàsic Hola món exemple, proporcionat per Node.js, millorat per admetre mTLS.

```
nano index.js
```

```
const https = require('https');
const fs = require('fs');

const hostname = 'localhost';
const port = 3000;

const options = {
    ca: fs.readFileSync('ca.crt'),
    cert: fs.readFileSync('server.crt'),
    key: fs.readFileSync('server.key'),
    rejectUnauthorized: true,
    requestCert: true,
};

const server = https.createServer(options, (req, res) => {
```

```
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Aquí el*requestCert*,*rebutjar*No autoritzat, i*això* Les opcions s'utilitzen per requerir que el navegador (client) proporcioni un certificat signat pel certificat CA per interactuar amb el servidor.

El*clau* i*cert* Les opcions permeten que el servidor proporcioni el certificat de servidor signat de CA.

Executeu el servidor:

```
$ node index.js
Server running at http://localhost:3000/
```

El client és simplement el navegador web cURL amb opcions:

```
$ curl  --cacert ca.crt --key client.key --cert client.crt https://localhost:3000
Hello World
```

Aquí el*cacert* s'utilitza perquè el client (cURL) pugui validar el certificat subministrat pel servidor. El*clau* i*cert* s'utilitzen perquè el client enviï el certificat de client signat de la CA amb la sol·licitud.

De fet, observem que aquesta sol·licitud torna amb èxit*Hola món*. Si, però, deixem fora el*cacert* opció, obtenim l'error:

```
$ curl   --key client.key --cert client.crt https://localhost:3000
curl: (60) SSL certificate problem: self-signed certificate in certificate
chain
More details here: https://curl.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore could not
establish a secure connection to it. To learn more about this situation and
how to fix it, please visit the web page mentioned above.
```

D'altra banda, si deixem fora les opcions de clau i certificat, obtenim un error diferent:

```
$ curl  --cacert ca.crt https://localhost:3000
curl: (56) OpenSSL SSL_read: error:0A00045C:SSL routines::tlsv13 alert
certificate required, errno 0
```