Exercise 14

Setting up TLS security for some of our existing services

Prior Knowledge
Previous exercises

Objectives
Understanding TLS certificates
Understanding TLS configuration

Software Requirements
- Node.js, Typescript and Express
- OpenSSL
- Python

Overview
In this lab we are going to create TLS certificates for both the server and the client, and validate them to ensure encryption, integrity and authentication.

Steps
1. Check that openssl is installed on your Ubuntu server:
   ```
   sudo apt install openssl
   ```

   Hopefully it is already at the latest level.

2. The first step is that we are going to create a certificate authority (CA). This is because we want to try client certificates as well as server certs. In the past you needed to pay (often quite a bit) for server certificates. Nowadays, LetsEncrypt has made that free. You do need a fully qualified DNS name (e.g. www.freo.me) to do that

   Hint 2: If you really do want to create a production CA, do not follow these instructions. They are far too insecure. You should read widely, but this is a good starting place:
   https://jamielinux.com/docs/openssl-certificate-authority/

3. If you want to make this more interesting, you could form into pairs and take different roles between yourselves. Ideally one person would be the CA, while the other is the server and client. You will need to send files between each system e.g. via Slack.

4. This guide is written for one person, but I've annotated it for a multi-party exercise (CA, Server, Client)

   If you are doing it all yourself, do all the parts.

If you are doing this as a pair, please share screens as you do each part.

5. Make some directories:

   CA:
   ```
   mkdir -p ~/sec/ca/private
   ```

   Server
   ```
   mkdir -p ~/sec/server/keys/private
   ```

   Client
   ```
   mkdir -p ~/sec/client/keys/private
   ```

6. We are going to act as several different roles in this lab. The first role is going to be the **CA Administrator**.

   Let's make a private key for the CA:

```
cd ~/sec/ca
openssl genrsa -aes256 -out private/ca.key.pem 4096

Generating RSA private key, 4096 bit long modulus
.....................................................................
.........................++
.....................................................................
...++
e is 65537 (0x10001)
Enter pass phrase for private/ca.key.pem:
```

   Enter a password. Probably best to use something insecure like "**password**" since this is not for real.

```
Verifying - Enter pass phrase for private/ca.key.pem:
```

   Re-enter the password.

   We put the key into the private directory so we can keep track of which parts need security and which don't.

7. We now need a certificate for the CA.

```
openssl req -key private/ca.key.pem -new -x509 -days 8000
-sha256 -out ca.cert.pem
```

   (All on one line)

   First enter your password.

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:UK
State or Province Name (full name) [Some-State]:Oxfordshire
Locality Name (eg, city) []:Oxford
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Comlab CA
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
```

   Fill in like I did.

This *ca.cert.pem* file doesn't need securing. In fact we want to share this certificate as widely as possible.

8. That is our CA created. We can now "switch hats" and be the **Server administrator**.
   `cd ~/sec/server/keys`

9. The server needs a private key. This doesn't need to be a secure as the CA key (lasts a year instead of 20 years!) so we can use 2048 bits.

   `openssl genrsa -aes256 -out private/server.key.pem 2048`
   I propose you use "password" again.

```
Generating RSA private key, 2048 bit long modulus
..................+++
.....................................................................
.....................................................................
..............................+++
e is 65537 (0x10001)
Enter pass phrase for private/server.key.pem:
Verifying - Enter pass phrase for private/server.key.pem:
```

10. No-one will trust this key because it hasn't been signed. In order to create trust we need to get a CA to sign this key. Luckily we know a friendly CA. To ask the CA to sign the key, we create a Certificate Signing Request (csr).

```
openssl req -key private/server.key.pem -new -sha256 -out
server.csr.pem
```

Again all on one line
Now use the following **bold** entries. The only really important one is the FQDN (**localhost**) since this will be checked against the DNS name of the server.

```
Enter pass phrase for private/server.key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:UK
State or Province Name (full name) [Some-State]:Oxfordshire
Locality Name (eg, city) []:Oxford
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Localhost Website
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:localhost
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

11. We now need to "send" that CSR to the CA:
`cp server.csr.pem ~/sec/ca/`
(or send it to the CA and have them save it to that directory_

12. Now we need to switch back to being the CA:
`cd ~/sec/ca`

13. Recent versions of Chrome and other browsers require an extra field in the certificate called the Subject Alternative Name. This must match the Common Name. This is a bit of a pain, as OpenSSL makes this quite hard to set.

14. You need to create a file called extfile, with the following contents (available here: https://freo.me/extfile):

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
subjectAltName=DNS:localhost
```

15. We now need to sign the CSR (as the CA).

```
openssl x509 -req -days 365 -in server.csr.pem -CAkey private/ca.key.pem
-CA ca.cert.pem -out server.cert.pem -CAcreateserial -extfile extfile

(All on one line)

Signature ok
subject=/C=UK/ST=Oxfordshire/L=Oxford/O=Localhost Website/CN=localhost
Getting CA Private Key
Enter pass phrase for private/ca.key.pem:
```

16. Now "send" the certificate back to the Server Admin:

    ```
    cp server.cert.pem ~/sec/server/key
    ```

    (or send and save there)

17. The server also needs the CA's certificate:
    ```
    cp ca.cert.pem ~/sec/server/keys
    ```

18. Switch back to being a Server Administrator:
    ```
    cd ~/sec/server
    ```

19. In general the key file needs a password. However, since our code needs access to the file, it doesn't actually increase security to have the password. It is better to simply make sure the file is protected.

    As a result, let's create a non-password protected key file.

    ```
    openssl rsa -in keys/private/server.key.pem -out
    keys/private/server-nopass.key.pem

    Enter pass phrase for server.key.pem:
    writing RSA key
    ```

20. Now we need to configure our server code to use the newly minted certificates.

    I'm assuming you have by now got a clone of purchase-complete. If not,

    ```
    cd ~
    git clone https://github.com/pzfreo/purchase-complete.git
    cd purchase-complete
    yarn install
    ./start-postgres.sh
    ```

    If you have the OAuth2 server as your main codebase, you should disable the introspect validation (comment it out!)

    ```
    // app.use(introspect(introspect_url,client_id,client_secret));
    ```

21. We need to make a couple of modifications to the code:

    Firstly add the imports:

    ```
    import {createServer} from 'https';
    import {readFileSync} from 'fs';
    ```

22. Secondly, change the port:

```
const port = process.env.PORT || 8443;
```

23. Thirdly, comment out the lines:

```
63  // app.listen(port, () =>
64  //   console.log(`Purchase app listening at http://localhost:${port}`)
65  // );
```

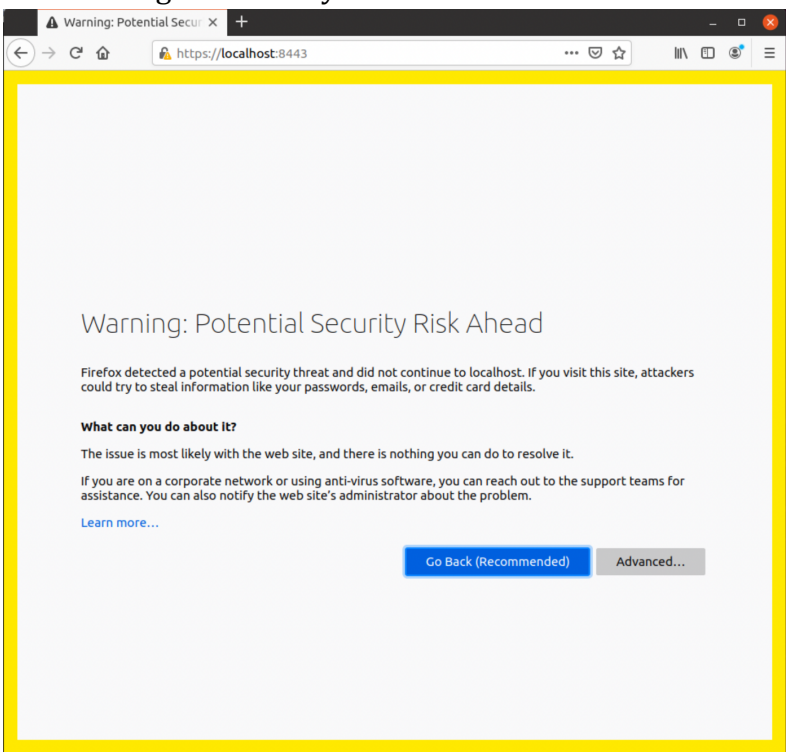
24. Finally, add new code to replace the http startup with https:

```
const base_dir = "/home/oxsoa/sec/server/keys";

const secureServer = createServer({
  key: readFileSync(base_dir+'/private/server-nopass.key.pem'),
  cert: readFileSync(base_dir+'/server.cert.pem'),
  ca: readFileSync(base_dir+'/ca.cert.pem'),
  requestCert: true,
  rejectUnauthorized: false
}, app);

secureServer.listen(port, function() {
  console.log(`Secure Purchase app listening at https://localhost:${port}`);
});
```

(snippet available here: http://freo.me/ts-tls)

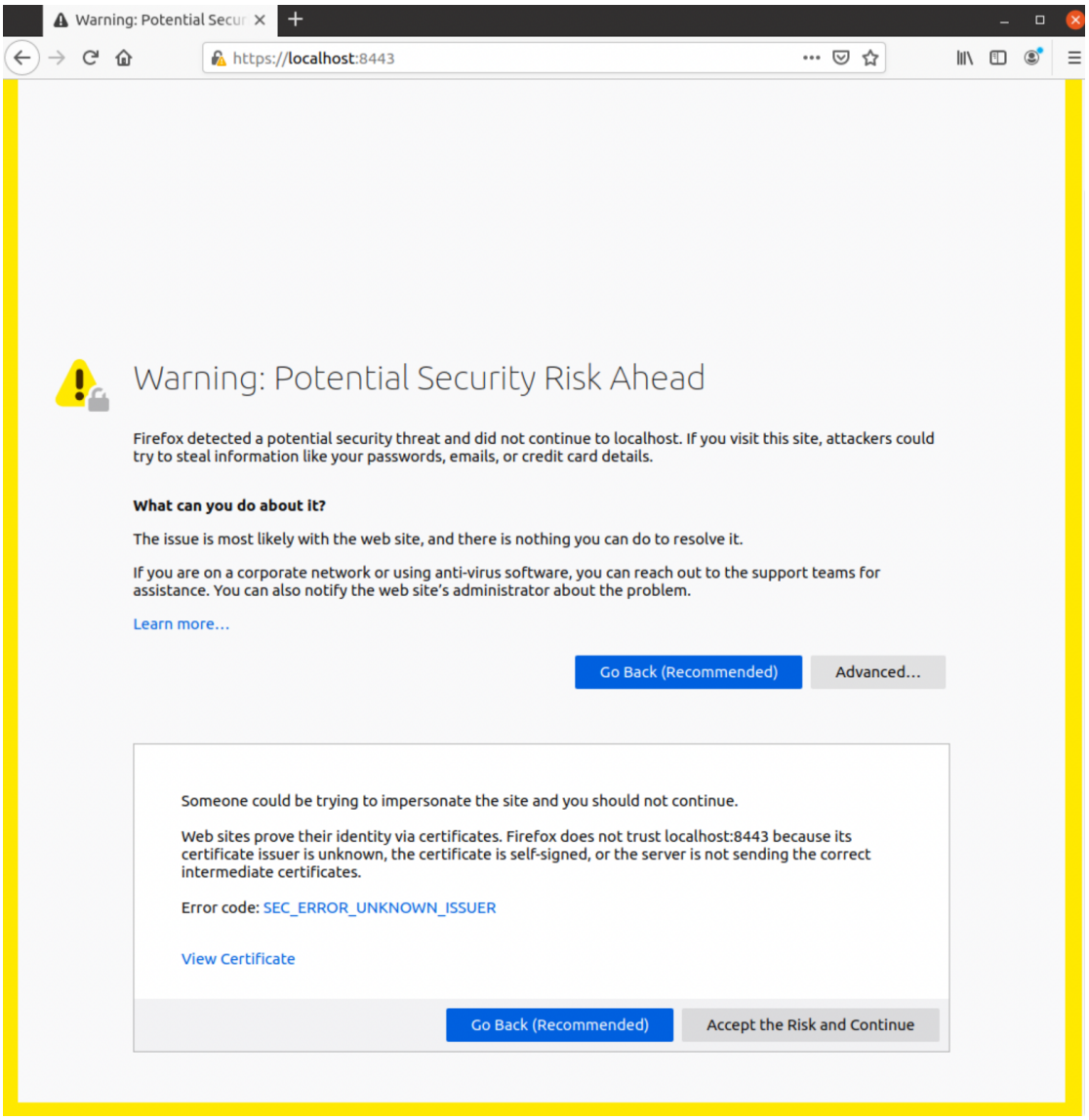
25. Use Firefox to browse to https://localhost:8443

You should get a security error:



26. Click **Advanced**

You should see:



27. Before reading on, try to understand what we have done so far and why Firefox is not convinced of the security.

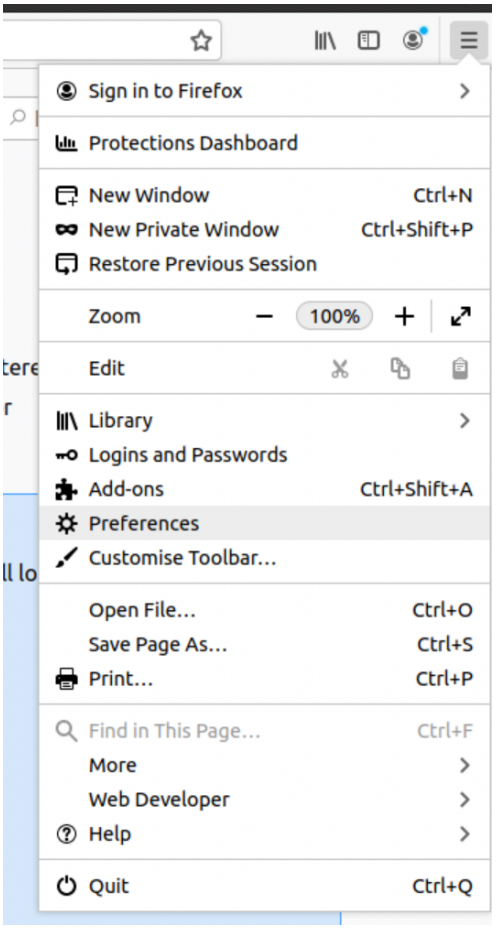
    To recap, we created a public+private keypair at the server-side. We then asked the CA to sign the public key to make a signed certificate. We then "installed" the signed certificate into the node server. What have we not yet done?

28. Don't accept the risk. Let's fix this "properly". Click **Go Back**

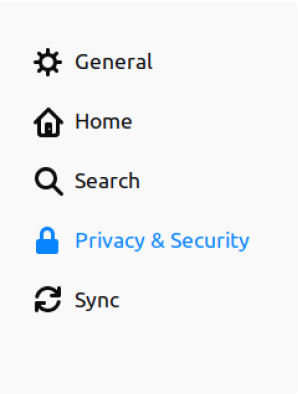
29. They say you get what you pay for, and in this case we didn't pay for our certificate. When you pay for a certificate, what you are really paying for is that the CA is trusted and hence browser manufacturers, SSL libraries, etc include the Root Certificate in their clients. The big innovation of LetsEncrypt was to get the root certificate for a free CA into all the browsers.

    However, we can add the CA's certificate to the client (in this case Firefox)

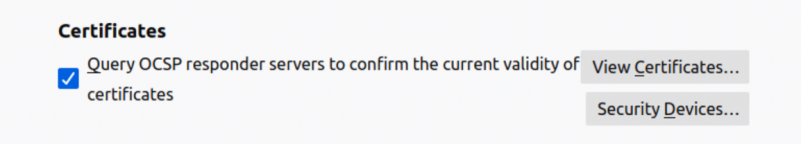
30. In Firefox go to Preferences:



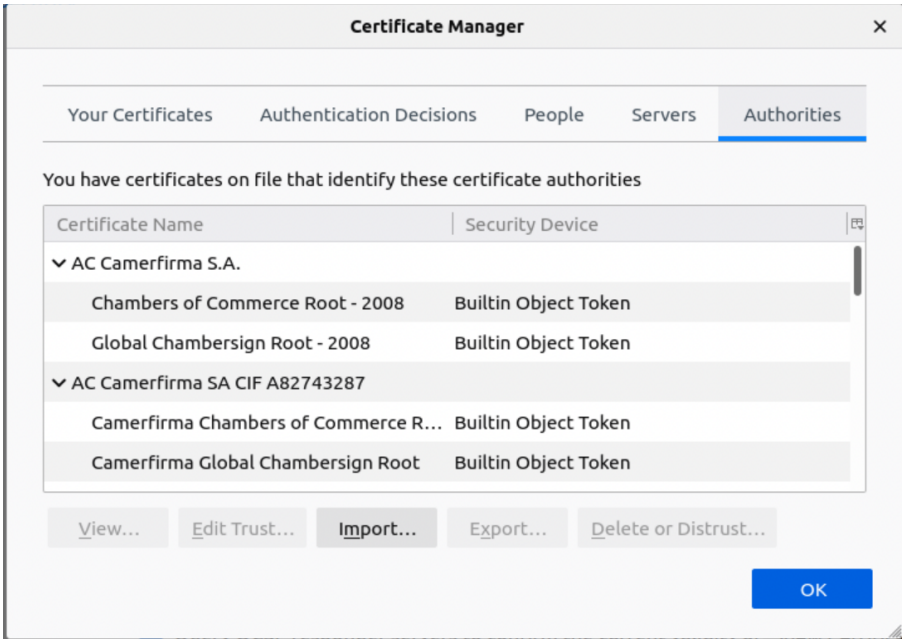
31. Now go to Privacy and Security:



32. Scroll to the end where you will find Certificates:

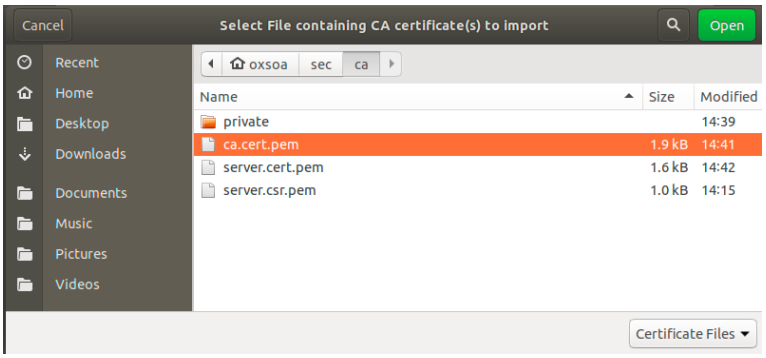


33. Click **View Certificates**

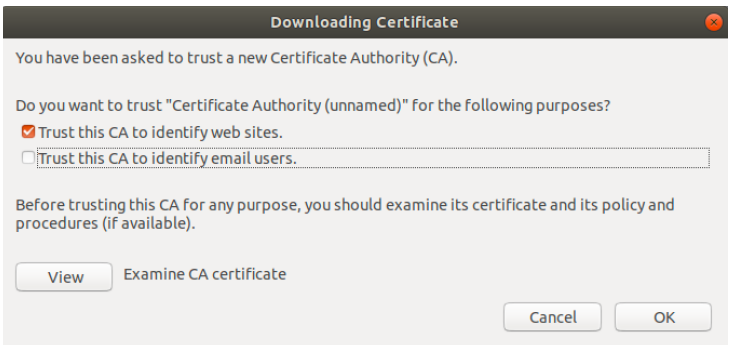


34. Click on the **Authorities** tab

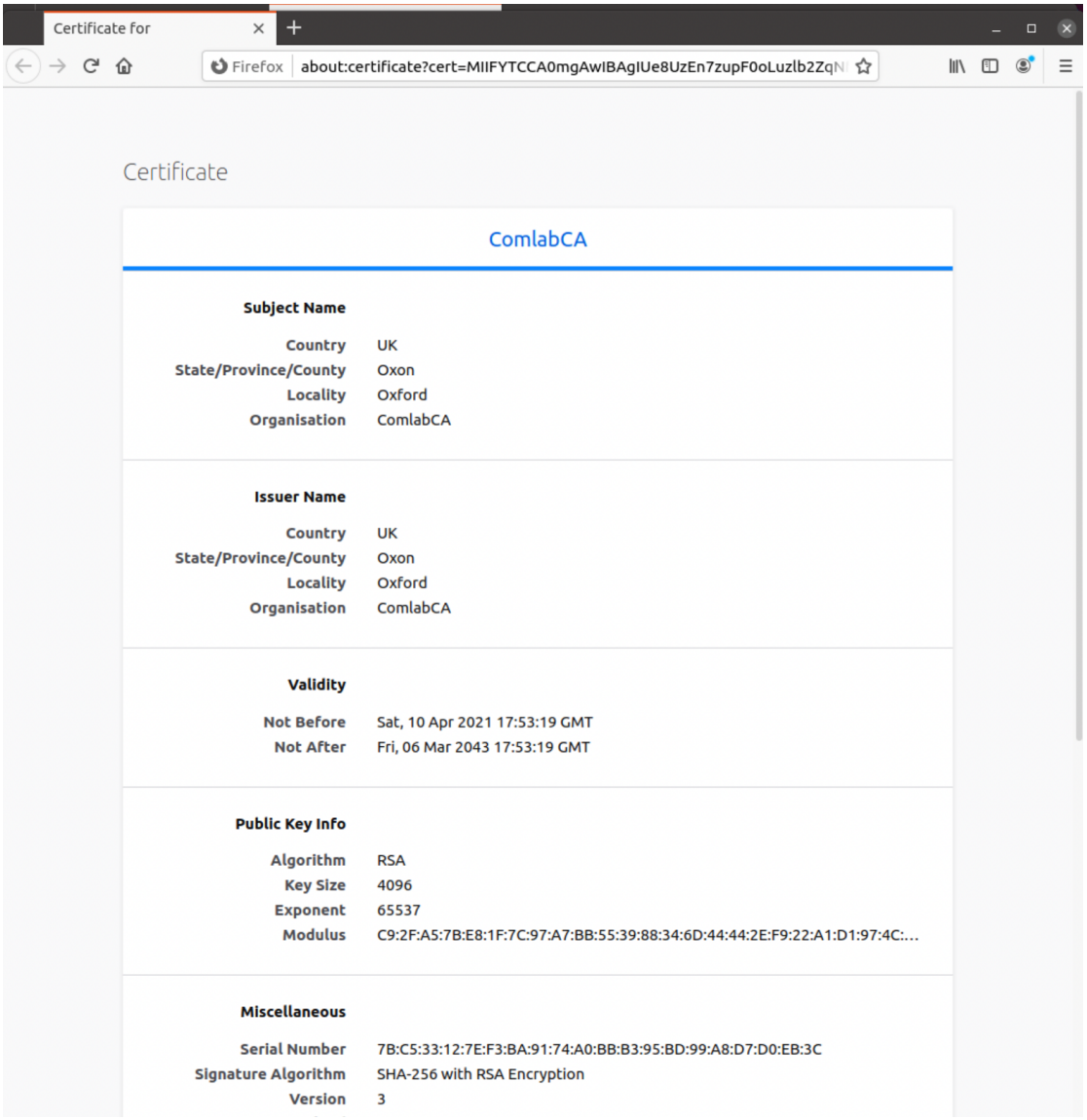
35. Click **Import**
    Browse to **ca.cert.pem** and select it



36. You should see:



37. Click View:



You should see data that the CA Administrator entered when creating the certificate.

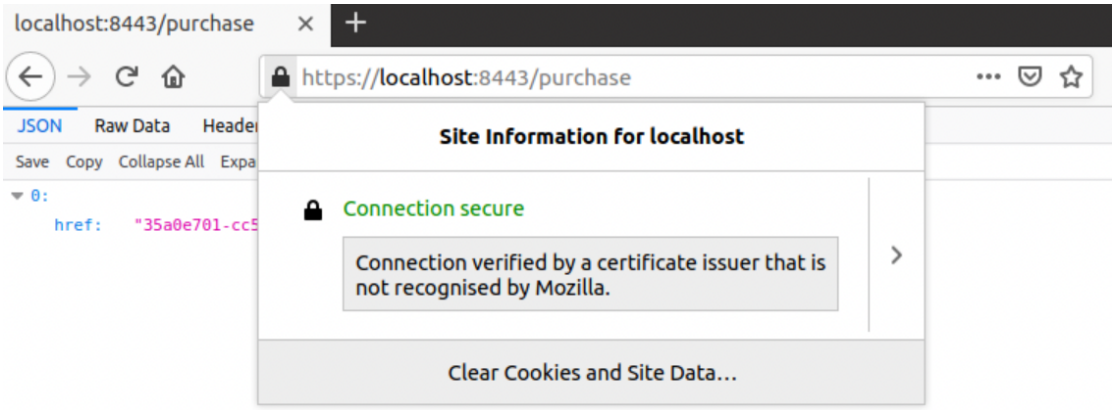
38. Click **Trust this certificate to identify websites**
Click **OK,** and then **OK** again

39. Close the settings.

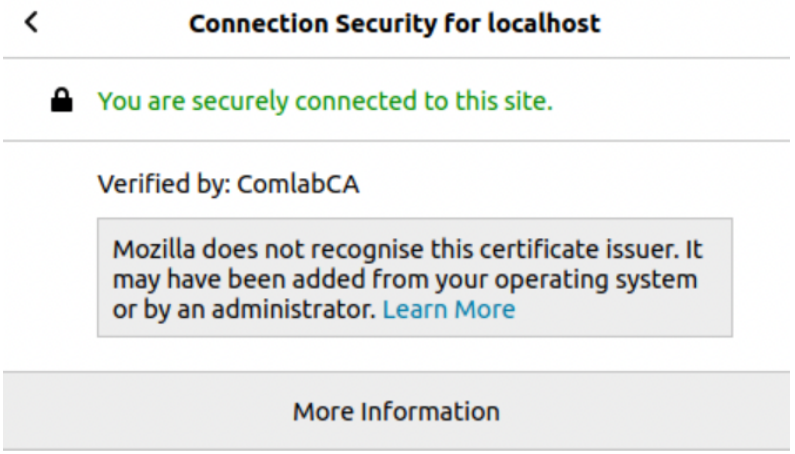
40. Now try browsing our “Secure Purchase” server:

    https://localhost:8443/purchase

    You should have a lovely secured padlock next to the server name in the URL bar. Click on it:



    Notice that Mozilla/Firefox are still warning that this in not one of their certificates. Click on the >



    This exactly describes the situation - we added the certificate issuer.

Client

We are now going to act as a third persona - the client administrator. Ideally this would be on a separate machine to the server, but given firewalls, etc, that is quite complex. So this lab assumes that the same machine is being used for the client and server, and that the server is still running in TLS mode.

41. Now we'd like to get our client working with this encryption.

42. If you have done the OAuth2 exercise you will have `~/python-purchase-client`

    If not, clone it:

    ```
    cd ~
    git clone
    https://github.com/pzfreo/python-purchase-client
    cd python-purchase-client
    ```

43. In the OAuth2 exercise, we had a client that read various secrets, etc. Let's ignore that for the minute, and start with a simpler client that does nothing except call the API.

    ```
    code purchase-create-bare.py
    ```

```
1   import os.path
2   from pathlib import Path
3
4   import requests
5
6   home = str(Path.home())
7
8
9   purchase_url = 'http://localhost:8000/purchase'
10
11  data = {
12      'paymentReference': "PR0001",
13      'poNumber': "PON0001",
14      'quantity': 3,
15      'customerNumber': "CUS0001",
16      'lineItem': "LI0001"
17  }
18
19  response = requests.post(purchase_url, json=data)
20
21  print(response.text)
```

44. Edit the client to use the URL: https://localhost:8443

45. Before you run it, think whether this will work?

46. Run it. As expected, it fails because the python runtime does not know about the CA certificate.

```
requests.exceptions.SSLError:
HTTPSConnectionPool(host='localhost', port=8443): Max retries
exceeded with url: /purchase (Caused by SSLError(SSLError("bad
handshake: Error([('SSL routines',
'tls_process_server_certificate', 'certificate verify
failed')])")))
```

47. We *can* work around this:
```
response = requests.post(purchase_url, json=data, verify=False)
```

    Try this out:

    It works, but you get a serious warning that this is not good!

```
/usr/lib/python3/dist-packages/urllib3/connectionpoo
l.py:999: InsecureRequestWarning: Unverified HTTPS
request is being made to host 'localhost'. Adding
certificate verification is strongly advised. See:
https://urllib3.readthedocs.io/en/latest/advanced-us
age.html#ssl-warnings
```

    Let's fix this.

48. First, send the CA certificate from the CA to the client developer:

```
cp ~/sec/ca/ca.cert.pem ~/python-purchase-client
```

49. Now modify the requests line:
```
response = requests.post(purchase_url, json=data, verify='./ca.cert.pem')
```

50. Try it again. Bingo! We now have encryption.

Authentication using a client certificate

51. We would like to authenticate the client as well as the server. One approach is to use a client certificate. Overall, this is less good than an API key, but there are some situations where this is a valid approach.

    First we need to create a client key etc.

```
cd ~/sec/client/keys
openssl genrsa -aes256 -out private/client.key.pem 2048
```

```
Generating RSA private key, 2048 bit long modulus
.....................................................................
.............................+++
...................+++
e is 65537 (0x10001)
Enter pass phrase for private/server.key.pem:
Verifying - Enter pass phrase for private/server.key.pem:
```

    Use password again.

52. Now we need to create a CSR again.

```
openssl req -key private/client.key.pem -new -sha256 -out
client.csr.pem
```

```
openssl req -key private/client.key.pem -new -sha256 -out client.csr.pem
Enter pass phrase for private/client.key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:UK
State or Province Name (full name) [Some-State]:Oxfordshire
Locality Name (eg, city) []:Oxford
Organization Name (eg, company) [Internet Widgits Pty Ltd]:SEP
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:your.name@cs.ox.ac.uk
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

53. There are two ways that we could validate the client certificate.
    The *proper way* is to get the CA (or another CA) to sign the certificate and then we can validate the certificate chain. The second more hacky way is to self-sign the certificate and hard code some attribute into our server logic to identify it.

    We are going to do both!

54. First, lets self-sign the certificate:
    ```
    openssl x509 -req -days 365 -in client.csr.pem -signkey
    private/client.key.pem -out self-sign.cert.pem
    ```
    ```
    Signature ok
    subject=C = UK, ST = Oxfordshire, L = Oxford, O = SEP, CN = your.name@cs.ox.ac.uk
    Getting Private key
    Enter pass phrase for private/client.key.pem:
    ```

55. Now let's get ask CA to sign the same request:
    Send the CSR to the CA
    ```
    cp client.csr.pem ~/sec/ca/
    ```

    Now acting as the CA, let's sign it
    ```
    cd ~/sec/ca/

    openssl x509 -req -days 365 -in client.csr.pem -CAkey
    private/ca.key.pem -CA ca.cert.pem -out client.cert.pem
    ```
    All on one line

    ```
    Signature ok
    subject=C = UK, ST = Oxfordshire, L = Oxford, O = SEP, CN = your.name@cs.ox.ac.uk
    Getting CA Private Key
    Enter pass phrase for private/ca.key.pem:
    ```

56. Send the new certificate back to the client:
    ```
    cp client.cert.pem ~/sec/client/keys/
    ```

57. Now we adjust the client code to use one or other of these certificates.

58. Before we do that, we don't want to encode our password into the client. In fact it's better to have no password on the client key and to rely on storing it securely.

    ```
    cd ~/sec/client

    openssl rsa -in keys/private/client.key.pem -out
    keys/private/client-nopass.key.pem
    ```
    (all on one line!)

59. Edit purchase-create-bare.py to include using the self signed certificate:

```
base_path='/home/oxsoa/sec/client/keys/'

response = requests.post(purchase_url, json=data,
    verify="./ca.cert.pem",
    cert=(base_path+'self-sign.cert.pem',
        base_path+'/private/client-nopass.key.pem'))
```

60. Now try the client. It will work, but there is no client authentication, because the client cert is not being checked.

61. Let's find the fingerprint of the certificate.

```
openssl x509 -fingerprint  -noout  -in
~/sec/client/keys/self-sign.cert.pem

SHA1 Fingerprint=63:29:60:26:0E:53:1A:F5:89:97:11:2D:9F:58:A5:B7:44:4B:11:78
```

(all on one line)

Note the fingerprint in bold.

62. Let's edit our server code to check for this serial. We can add a quick piece of new middleware.

First add this import to `src/app.ts`

```
import {TLSSocket} from 'tls';
```

Now add this code as another middleware before "RegisterRoutes". Obviously use your fingerprint not mine! (Snippet here: http://freo.me/tls-middleware)

```
const
FINGER="63:29:60:26:0E:53:1A:F5:89:97:11:2D:9F:58:A5:B7:44:4B:11:78";
app.use(function (req:express.Request, res:express.Response,
next:express.NextFunction)
{
  const tlsSocket = req.socket as TLSSocket;
  const incomingFinger = tlsSocket.getPeerCertificate().fingerprint;
  if (incomingFinger != FINGER) {
    res.status(401).send("Unauthorized");
  }
  next();
});
```

63. Test this out. (run both server and client)

64. Change the fingerprint in the code and retest to make sure that it really is checking for that exact fingerprint.

65. You could also check other aspects such as the DN, but these will be less secure since this is self-signed. You could also write a registration process that grabs the serial number during a certain phase and then looks for it later.

    However, this is not an easy or scalable approach. It is useful though when you need to enforce trust between just two parties. For example, you might have a secure proxy that is performing authentication and you want to ensure that all traffic goes via that proxy. You give the proxy a self-signed certificate and the server only accepts traffic from there.

The next approach solves the scalability concern by validating a proper CA-signed client key.

66. Let's enforce CA checking. Edit the src/app.ts to read:

```
const secureServer = createServer({
  key: readFileSync(base_dir+'/private/server-nopass.key.pem'),
  cert: readFileSync(base_dir+'/server.cert.pem'),
  ca: readFileSync(base_dir+'/ca.cert.pem'),
  requestCert: true,
  rejectUnauthorized: true
}, app);
```

The difference here is "rejectUnauthorised" is now true.

67. Comment out the fingerprint check middleware as well.

68. Restart the server.

69. Try your client. It won't even connect as the server rejects it at the TLS layer.

70. Now edit the `purchase-create-bare.py` client to use **client.cert.pem** instead of *self-sign.cert.pem*

71. Try again.

    Everything should now work, demonstrating that both the client and server trust each other because the CA has signed both certificates.

72. Now that we have a CA trust, we could trust entries in the client certificate, so we could validate the CN, DN, etc for example.

73. *Recap:*
    This has been a long lab, but security is a complex aspect.
    What we have done is to set up mutual SSL with both the client and server authentication via certificates. We have also explored server-only certificates and self-signed approaches.

Extension

1. Find the CN from the client certificate in the server code.

2. If you wanted to get the combination of Server-side TLS plus OAuth2 token that might be a good exercise to see if you've really understood this exercise plus the OAuth2 token exercise.