

# Exercise 11

*Setting up SSL security for some of our existing services*

## Prior Knowledge

Previous exercises

## Objectives

Understanding SSL certificates

Understanding TLS configuration

How to use Docker Compose to install certificates securely

## Software Requirements

(see separate document for installation of these)

- TBD

## Overview

*TBD*

## Steps

1. Check that openssl is installed on your Ubuntu server:  
`sudo apt-get install openssl`
2. The first step is that we are going to create a certificate authority (CA). This is because we don't want to have to go through the rigmarole of paying a real CA.

*Hint: if you are willing to host a real server on a real fully-qualified DNS name, then you can get a free certificate from the EFF via certbot. However, we aren't able to do that today.*

*Hint 2: If you really do want to create a 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. Make some directories:  
`mkdir -p ~/sec/ca/private`  
`mkdir -p ~/sec/server/keys/private`  
`mkdir -p ~/sec/client/keys/private`

4. 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.

5. 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 []:
```

Now enter the following **bold items**. Hit enter for the others.

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

6. That is our CA created. We can now “switch hats” and be the **Server administrator**.  
cd ~/sec/server/keys
7. The server needs a private key. This doesn't need to be as 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

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:
```

I propose you use “password” again.

8. 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 (ourselves). 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 []:
```

9. We now need to "send" that CSR to the CA:

```
cp server.csr.pem ~/sec/ca/
```

10. Now we need to switch back to being the CA:

```
cd ~/sec/ca
```

11. 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
(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:
```

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

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

13. The server also needs the CA's certificate:

```
cp ca.cert.pem ~/sec/server/keys
```

14. Switch back to being a Server Administrator:

```
cd ~/sec/server
```

15. We are now ready to use our keys and certificates in an application.

There is an updated version of our random app:

```
var https = require('https'),
    fs = require('fs'),
    express = require('express'),
    app = express();

app.get("/", function(req, res){

    obj = {random : Math.floor((Math.random() * 100) + 1)};
    res.json(obj);

});

var secureServer = https.createServer({
    key: fs.readFileSync('./keys/private/server.key.pem'),
    cert: fs.readFileSync('./keys/server.cert.pem'),
    ca: fs.readFileSync('./keys/ca.cert.pem'),
    requestCert: true,
    passphrase: "password",
    ciphers: "TLSv1.2",
    rejectUnauthorized: false
}, app).listen('8443', function() {
    console.log("Secure Express server listening on port 8443");
});
```

This should all be fairly obvious.

16. Download this code.

```
cd ~/sec/server
```

```
curl -L http://freo.me/sec-rand -o server.js
```

Your directory should look like:

```
> tree
```

```
.
├── ca
│   ├── ca.cert.pem
│   ├── ca.srl
│   ├── private
│   │   └── ca.key.pem
│   ├── server.cert.pem
│   ├── server.crt
│   └── server.csr.pem
├── client
│   └── keys
│       └── private
└── server
    ├── keys
    │   ├── ca.cert.pem
    │   ├── private
    │   │   └── server.key.pem
    │   ├── server.cert.pem
    │   └── server.csr.pem
    └── server.js
```

8 directories, 11 files

17. Make sure express is installed:

```
npm install express
```

18. Run it:

```
node server.js
```

19. Browse to <https://localhost:8443>

You should get a security error.

20. 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.

However, we can do that ourselves.

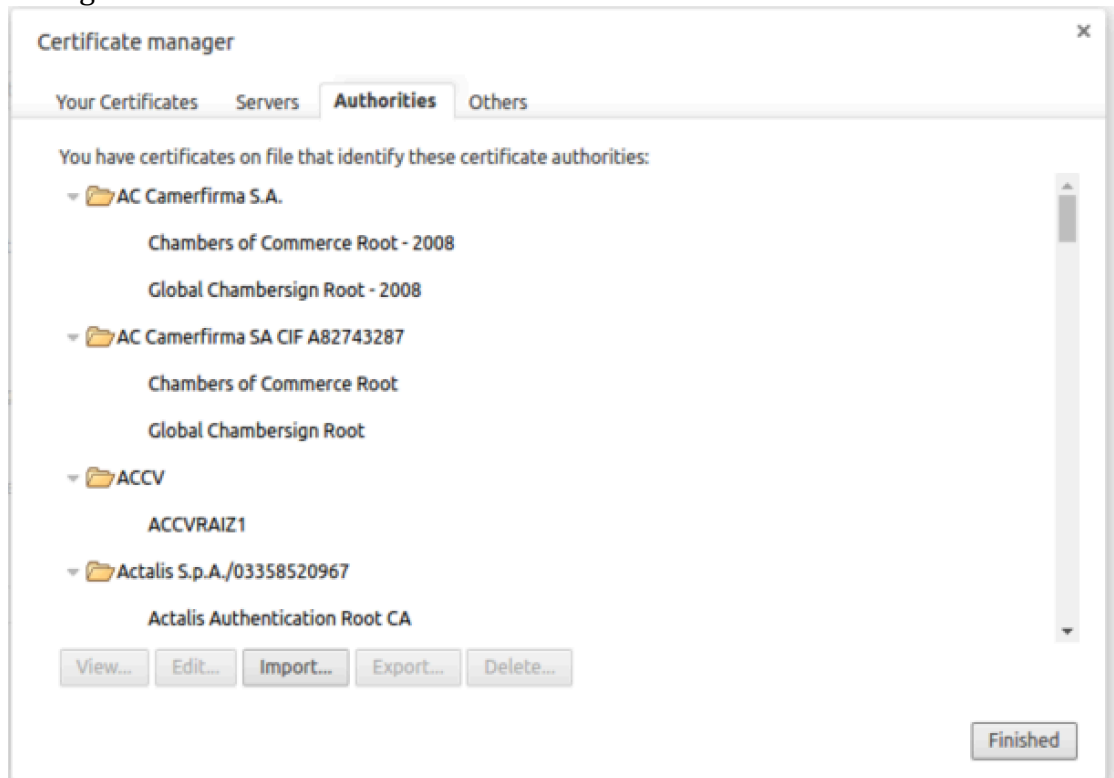
21. Do not accept it but instead add the certificate. In Chromium:

**Edit -> Preferences.**

**Show advanced settings**

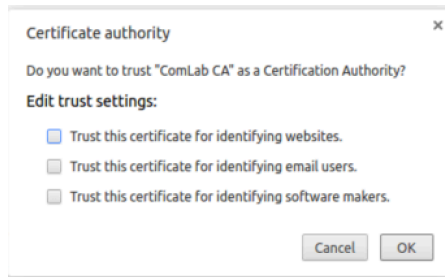
**HTTPS/SSL -> Manage Certificates**

## 22. Now go to Authorities:



Click **Import**  
Browse to **ca.cert.pem**

23. You should see:



24. Click Trust this certificate for identifying websites.

25. Click Finished and then close the settings.

26. Now try browsing again. You should have a lovely green padlock next to the server.

#### *Client*

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

28. First let's find our random JSON client:

```
cd ~/sec/client  
curl -L http://freo.me/rand-client -o random-client.py
```

29. Edit the client to use the URL: <https://localhost:8443>

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

31. We could work around this:

```
h = httplib2.Http(disable_ssl_certificate_validation=True)
```

However, this puts us in danger of Man in the Middle attacks.

32. Instead, modify the client to use the following:

```
h = httplib2.Http(ca_certs="./keys/ca.cert.pem");
```

33. Copy the ca.cert.pem into the client directory:

```
cp ~/sec/ca/ca.cert.pem ~/sec/client/keys
```

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



*Authentication using a client certificate*

35. We would like to authenticate the client as well as the server. One approach is to use a client certificate.

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.*

36. Now we need to create a CSR again.

```
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) []:paul@fremantle.org
Email Address []:

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

37. 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!

38. 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=paul@fremantle.org
Getting Private key
Enter pass phrase for private/client.key.pem:
```

39. Now lets get the CA to sign the same request:

```
cp client.csr.pem ~/sec/ca/
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=paul@fremantle.org
Getting CA Private Key
Enter pass phrase for private/ca.key.pem:
```

40. Copy the new certificate back to the client:

```
cp client.cert.pem ~/sec/client/keys/
```

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

```
cd ~/sec/client
```

42. We don't want to encode our password into the client. In fact its better to have no password on the client key and to rely on storing it securely.

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

43. Edit random-client.py and add the bold line.

```
import httpplib2
import json

url = "https://localhost:8443"

h = httpplib2.Http(ca_certs="./keys/ca.cert.pem")
h.add_certificate(key='./keys/private/nopass.key.pem',
cert='./keys/self-sign.cert.pem', domain='')
resp, content = h.request(url, "GET")

print "return code: " + resp['status']
result = json.loads(content)
print "random number: " + str(result['random'])
```

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

45. Let's find the serial number from the certificate.

```
openssl x509 -text -noout -in keys/self-sign.cert.pem
```

Certificate:

Data:

Version: 1 (0x0)

Serial Number:

8f:62:2e:7c:e3:a6:c8:9a

Signature Algorithm: sha1WithRSAEncryption

Issuer: C=UK, ST=Oxfordshire, L=Oxford, O=SEP, CN=paul@fremantle.org

Validity

Not Before: Jun 7 13:28:26 2016 GMT

Not After : Jun 7 13:28:26 2017 GMT

Subject: C=UK, ST=Oxfordshire, L=Oxford, O=SEP, CN=paul@fremantle.org

46. Let's edit our server code to check for this serial. Modify it to look like:

```
app.get("/",function(req,res){
  console.log(req.socket.getPeerCertificate());
  if (req.socket.getPeerCertificate().serialNumber=="8f622e7ce3a6c89a")
  {
    obj = {random : Math.floor((Math.random() * 100) + 1)};
    res.json(obj);
  }
  else {
    res.sendStatus(403); // forbidden
  }
});
```

*Notice I removed the ':'s from the serial. Of course you need to use your own serial number, not mine.*

47. 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.

48. Now let's enforce CA checking. Edit the server.js to read:

```
var secureServer = https.createServer({
  key: fs.readFileSync('./keys/private/server.key.pem'),
  cert: fs.readFileSync('./keys/server.cert.pem'),
  ca: fs.readFileSync('./keys/ca.cert.pem'),
  requestCert: true,
  passphrase: "password",
  ciphers: "TLSv1.2",
  rejectUnauthorized: true
}, app).listen('8443', function() {
  console.log("Secure Express server listening on port 8443");
});
```

49. Comment out the serial check in server.js at the same time.
50. Restart the server.
51. Try your client. It won't even connect as the server rejects it at the TLS layer.
52. Now edit the client to use **client.cert.pem** instead of self-sign.
53. Try again.
54. Now that we have a CA trust, we could trust entries in the client certificate, so we could validate the CN for example.
55. 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 authenticating via certificates. We have also explored server-only certificates and self-signed approaches.