

## CG2111A Engineering Principles and Practices II

### Studio 15 Secure Alex to the Clouds!

#### Introduction

In this studio we will look at how to establish a connection to Alex from over the Internet, and how to secure that connection.

This studio consists of four compulsory activities, and one optional activity.

**Activity 1:** Becoming Your Own Certificate Authority.

**Activity 2:** Creating and Signing Server and Client Keys.

**Activity 3:** Writing TLS Programs.

**Activity 4:** Securing Alex.

**Optional Activity:** Introducing Network Debugging Tools

#### **Special Note:**

It is now easy to obtain free cryptographic certificates your website and applications from organizations like Let's Encrypt (<https://letsencrypt.org/>). However in this Studio we will learn how to generate our own certificates and how to sign them. This gives you some insight into how certifying authorities (CAs) work and how to produce your own certificates.

#### Team Formation

You will work in your regular Alex Teams of between 4 and 5 persons.

#### System Setup

Please ensure that your team has at least one of the following:

- a. A laptop running MacOS.
- b. A laptop running Ubuntu LINUX.
- c. A Windows laptop that has the LINUX Ubuntu Subsystem installed. See <https://techcommunity.microsoft.com/t5/windows-11/how-to-install-the-linux-windows-subsystem-in-windows-11/m-p/2701207> for instructions.

In addition. you will of course need Alex, properly set up for VNC or SSH access.

#### Submissions

No submissions necessary as this is an ungraded studio.

**Note:** Some parts of this studio are done on the Pi, some on your Ubuntu / MacOS / Windows LINUX Subsystem laptops. Please pay attention to the **WHERE** instructions.

---

## ACTIVITY 1 – BECOMING YOUR OWN CERTIFICATE AUTHORITY

---

For your TLS sessions to work, every host (client and server) needs a pair of public and private keys. In addition, you will need to generate pair of keys that is used for signing all keys from every server in the network. This special signing key is known as a Certificate Authority Key (CA key – i.e. Charlie’s key in the lecture). As it was foretold in ages past:

Three key pairs for the Elven-kings under the sky,  
Seven for the dwarf-lords in their halls of stone.  
Nine for the mortal men doomed to die.

One for the Key Signing Party on their dark thrones.

In the land of Computer Engineers where the Shadows lie. One key to rule them, one key to find them,

One key to bring them all and in the darkness bind them. In the Land of Computer Engineers where the Shadows lie.

So yes, the CA key that you use to sign all other keys is EXTREMELY IMPORTANT because they tell an entire organization that the keys they sign are legitimate and can be trusted. If you hold your company’s CA key and you lose it, I hope you have a fresh CV and spare cash for the lawsuits that follow.

Also this should be common-sense, BUT: **IF YOU ARE USING A REPO, DO NOT PUSH YOUR PRIVATE KEYS ONTO YOUR REPO! Add your private keys and CA keys to your .gitignore.**

(As an aside, the entire Internet – yes, all of it – is controlled by seven people who hold the keys that enable updates to the Domain Name Service. This is an extremely important job because DNS pollution can cause you to go to a phishing site when you try to go to <https://www.dbs.com> to do your Internet banking. You can read about these seven people here:

<https://www.theguardian.com/technology/2014/feb/28/seven-people-keys-worldwide-internet-security-web>)

Now that you understand the importance of keys and especially of your CA key, let’s see how to generate them. (Note: In this activity you will be generating “self-signed keys”. In the real-world keys are signed by external Certificate Authorities (CAs). Self-signed keys cannot be used for web servers as all modern web browsers – even those written by Microsoft – will reject keys that are not signed by a recognized group of CAs.)

### Step 1.0. SYSTEM SETUP

**WHERE: DO THIS PART OF YOUR STUDIO ON YOUR UBUNTU / MacOS OR WINDOWS LINUX SUBSYSTEM LAPTOP.**

Test that you have openssl by typing: openssl version

You should get a reply that looks something like this:

```
ctank@Colins-MacBook-Pro ~ % openssl version
OpenSSL 3.2.1 30 Jan 2024 (Library: OpenSSL 3.2.1 30 Jan 2024)
ctank@Colins-MacBook-Pro ~ %
```

If you instead get an error message saying that openssl is not found (or some similar message), install it using on Ubuntu:

```
sudo apt-get install openssl
```

On MacOS you will first need to install Homebrew. Please see <https://docs.brew.sh/Installation> for details. After installing homebrew you can proceed to install openssl.

```
brew install openssl
```

### Step 1.1. GENERATING THE CA KEY

First we generate the CA key and certificate. You will use this key to sign other keys. Open a new bash shell on your laptop (Windows LINUX Subsystem, Ubuntu or MacOS) and type:

```
openssl genrsa -verbose -des3 -out signing.key 4096
```

This is what it means:

openssl: The software used for generating and signing keys.

genrsa: Generate a Rivest-Shamir-Adleman (RSA) key. RSA is the best known public key cryptosystem.

-des3: The Data Encryption Standard (DES) is a popular private-key encryption standard created by the US Government. Triple-DES (3DES) uses a pair of private keys K1 and K2 in this way:

Encrypt message M using DES and K1 to get message M'. Decrypt M' using DES and K2 to get M''. Encrypt M'' using DES and K1 to get M'''.

Why all this trouble? As it turns out, simply encrypting a message twice, first with K1 then with K2 will result in simply encrypting the message with a concatenated key K1+K2, reducing the key-depth and making it far less secure. This odd 3DES algorithm removes that weakness.

The `-des3` option means that we should encrypt our keys using 3DES.

-out: Specifies the filename for the output key

4096: Specifies that we should generate a 4096-bit key pair. 1024-bit keys are no longer supported.

You will be asked for a password. Select your password carefully and remember it, or you won't be able to sign any certificates.

You will see openssl working its magic:

[illegible]

You will now have a CA key that looks like this (To fit everything into one screen, this shows a 1024-bit key. Your 4096-bit key will be much longer).

```

pi@ctank:~/CG1112/Week 11 Studio 1$ cat signing.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,9052F1B84ED9B9D0

zQwHgev6Xg8Tvxddk2EG7E6UtVsXApAJ8LfKcm/Iu+18RuHZr5/AB/hS90ApCNm
nf654z1e0hW0Di5TOvHZ82IlVDs8FoxrdA72VCNs65dyyFqSnAOSpKDTmQTGUDqV
nBmCmZLPNpohTPDI1lHjfs0gpRbEGK7ZUAUnMtHV7pEpRlcNnZvzu5lS4jmjJ4b8
SVRl92aZ0kGX0kZoGwVh2LLJ4gnmnHhHOS0Cw7TRm//FRV8FCwE/K+/OwwzWXCiLG
ITMFiZcFhWw5Vfov3LmXbziRb5aVGI8ht1e0pGNslht9qPECNCw3Srwvs5AkiGUt
k/JrLeDiTW10EA/oW7rKx82FvON65PUrPsvWJVkAMEd+ha/yVO2CYYZBNdCfPEHU
6CsvClFpp57iQKulaJt/voTlKDIGomfgXRH/VofLh+uumxxt4VhmeTgd4z92Oj4v
MuqPVSrrxTGm+1DsW4gOi3h7HvTfBZ2Bk6Aftc1NQt2hKO1YTzbyqXaFyxgBE7h8
38I7ip6ciRsh+RZOCHcM+k1coG7b1uCXmjQtsaaEK+u/fiTK6YJVgSxPTKsyoomC
IOLzFHJ/X61RHCucB6H41lNMwbeHc5g7/99TaJ4hruYcnKlLD8+sXDgBw2ToiuX5
rRO5v/sl3fbr8NjYJgCg1VuAFLb01Vo8/NVZFDyo/YLzwXJ+eB5UyLUU+Wtcln/p
vi7eoUksgTCWD99HwxPddj/kSy0fPex90PLGI5KyPf1x2wdM7TYMUHjteL1Whs4C
8dkMUV8wg/Yw5s4znNkEwvCp5XNpA99zEgupLlwKPWTky/0YtxCl9w==
-----END RSA PRIVATE KEY-----
pi@ctank:~/CG1112/Week 11 Studio 1$ █

```

**KEEP THIS CA KEY SAFE BECAUSE IT WILL BE USED TO SIGN ALL OTHER KEYS USED BY YOUR ORGANIZATION / EPP TEAM. IF ANYONE ELSE GETS THIS CA KEY, THEY CAN SIGN SPURIOUS PRIVATE KEYS AND GENERATE CERTIFICATES TO COMPROMISE YOUR PROJECT.**

#### Step 1.2. SIGNING THE CA KEY TO GENERATE A CA CERTIFICATE

The next step is to sign your CA key to create a CA certificate. This CA certificate is the public key that clients use to verify your signature on other certificates, and **MUST** be copied over to all clients who wish to connect to any of your servers. To generate the CA certificate:

```
openssl req -x509 -new -nodes -key signing.key -sha256 -days 1024 -out signing.pem
```

This means:

**req -x509:** Generate an X509 certificate for this key. X509 is a standard that is agreed upon by the entire world on what a certificate should contain and what it looks like.

**-new:** This is a new request.

**-nodes:** Don't encrypt the output key

**-key:** Specifies the key you are signing.

**-sha256:** Use the 256-bit Secure Hash Algorithm (SHA256) to generate a "digest" or a "hash" (they mean essentially the same thing) of the key. A "digest" is a non-trivial irreversible mathematical transformation. Non-trivial means that it is extremely hard to produce an identical digest unless you know the original contents, and irreversible means that it is impossible to recover the original contents from the digest. This digest is used to verifying the signature.

-days 1024: This certificate is valid for 1024 days.

-out: Write the certificate to this file

You will be prompted for your private key password, then asked to fill in some data. The screen on the next page shows example data. Fill in your own data (or make some up.)

**IMPORTANT:** Remember what you used for your Common Name. Here we are using signer.com. You **MUST NOT** use this same name for your other certificates later!

```
ctank@Colins-MacBook-Pro keys % openssl req -x509 -new -nodes -key signing.key -sha256 -days 1024 -out signing.pem
Enter pass phrase for signing.key:
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]:SG
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) []:signer.com
Email Address []:
ctank@Colins-MacBook-Pro keys %
```

Once done you will have a certificate:

```
-----BEGIN CERTIFICATE-----
MIIF1TCCA32gAwIBAgIUU76iIvxcz1pjZ8HbwZo0yn6F57gDQYJKoZIhvcNAQEL
BQAwWjELMAkGA1UEBhMCU0cxZzARBgNVBAGMClNvbWU3RhdGUxITAfBgNVBAoM
G1udGvYybmV0IFdpZGdpdHMgUHR6IEUxOzZETMBEGA1UEAwKc2lnbmVYLmNvbTAe
Fw0yNDAzMjUxNTAzMzFaFw0yNzAzMDMxNTAzMzFaFmFoxCzA3BGNVBAITA1NHMRMw
EQYDVQQIDApTb211LVN0YXR1MSEwHwYDVQQKDBhJbnRlcm5ldCBXaWRnaXRzIFB0
eSBMdGQxZzARBgNVBAMMCnNpZ251ci5jb20wggiMA0GCSqGSIb3DQEBQUAA4IC
DwAwggIKAoICAQCDqwxTe0+XatG2SeeByBejkOmb5e38JNF4SJoecx/tXU6Yx5
uXcA1KYGNc0E610hxPHjuoIZmkRaU4Szbku5F9m/kdnd+2Foe+dz17Fn4Aeo4hK6
hmC+v6Vhk15rH11oGR2urc1AJqtXo+pLQVtmJ6BwjBdX7gaLQZAA2hHcNe8u01Rs
VktP0Z0kH/jhEuCiEqbt4MgMpkVynAR816Kfhaka61KoHqx0BC6m+KURmbb31j
k8LNUrNE+I3PAB+0BY6G159FVF/6tAKgMSKMyS9ynozs8zoRKd516jAmkFnNwH2
9B+dqyGTKKCWUcGa054TC7KjYcggVVEzsdjKKDkkfDofaxe08pxG4FT1t7RoEX1K
NGefqGW9ZUzYKbJoDa6MIgcYnMe0c1CaW3jq1KtaJdcQGLS7IS8EWCTOnU0/pC
1giqqz09Kk57KETieX7IqTfdQzrbBkb/ov2kmyrSdHKnhCQ63p1bY/Rxt1Ed9y8
/ShRPIn5sYE/wV7Z4tpp6bt1goEHx0XehB+vxhkp6y57ITZ6xVWA/eCUgOkzqND0
D7Z2XmXp1G7kM01HsiKg4EctPI/gQ6Z3HJq/rH2B1XusQ5ipue0Z77zpbXn+U04B
j3T8peq3Rzgf+BCw+eZk4wZ/db1HzkTvtE3M8hr7WpT+tb/fbp4WndUwIDAQAB
o1MwUTAdBgNVHQ4EFgQUUNam//Q36xZUnhFdEU1mHyCGIswHwYDVR0tBBgwFoAU
UNam//Q36xZUnhFdEU1mHyCGIswDwYDVR0TAQH/BAUwAwEB/zANBgkqhkiG9w0B
AQsFAAOCAQEAJyCj7u2EZGzBU2gug4yxj8FHN1T8rwdDa53qK07f0Y6YfxJ0nULZ
iamScviKTWv+2dnanVwn7eYMSmdjRbVaxR0hxrmo000DCsafi+K7qWDz21IH2BNM6
zK8oLS98U4ScxV300xJfnj8sJLTKmndotcx4i9DATIBHN4NCdPgZYS1SpCw1/W
WDtpKyYf8WtAouEvVoAFQKJD0FCqAH3j+ZD1SutdneEted+6sidVucw2BB9V4pKq
7LPL+hGCC4NtwPuFacwJVgUYV8mbqx5GKkLLNgA30BiAcfxZvSMZHGxNHT8vNSF
ENFXK00mWGXjkk2X3CSmR+Jp6wN4n1UqeAIEc1DrqEZB0S2T80sYred2jo+n6v0t
uWw7WdEuNRVW6j/u2G/19MwzqK7Ki0n1kw37HiW2NmZPR5zAyw6OCGH3PUuWRabN
05JcCLsipyu6TeQ7LHDWUI8VWvuH/+bSZVYYNQHG6nbU1zpgI8IvX/I7Py7fyLGs
bAWj+zyRdFpo+HWjPJ404v9RsWJ9U1e0f8w5Sx5Y5Kgx28/tX4T4p0+F0rBis175
/XzMzkGq3f5gYD8YfyERuUfqI1SBgtUrFp48hN4m4A7NmnvPms4YJLM6jYOHhzi
6A7uF1vpXbS2UWnnkuuTEBK9dV5KXBJcnkxxQ1wJ+2PU2wq68wT2juY=
-----END CERTIFICATE-----
```

THIS CERTIFICATE signing.pem MUST BE COPIED OVER TO ALL CLIENTS WISHING TO CONNECT TO YOUR SERVER. If you don't, your clients cannot verify server keys and the TLS session will fail. If your server is also authenticating clients, then this certificate must also be copied onto your server.

That's all there is to it! You are now ready to sign keys for your entire organization! So let's begin generating keys for both Alex and the laptop you are going to control him with. ☺

---

## ACTIVITY 2 – GENERATING KEYS FOR ALEX AND YOUR LAPTOP

---

We will now generate public and private keys for Alex and your laptop. Recall from the lecture that the public keys will be distributed to everyone as certificates signed by your signing key you generated in Activity 1, together with your CA certificate, so that anyone wishing to use your public key can verify that it has been properly signed.

Let's begin!

### Step 2.1. GENERATING ALEX'S KEY

**WHERE: DO THIS PART OF THE STUDIO ON ALEX'S PI.**

#### **Step 2.1.0: Preparation.**

SSH or VNC over to Alex's Pi, and follow Step 1.0 to verify that openssl is installed. If not, do "sudo apt-get install openssl" to install openssl onto your Pi. You will need an Internet connection.

#### **Step 2.1.1: Generate Alex's Private Key.**

We will now generate Alex's private key. To do this:

```
openssl genrsa -out alex.key 4096
```

This will generate a 4096 bit key for Alex. Notice that unlike our CA key in Step 1.1, here we don't specify `-des3` as we don't want to encrypt the key, otherwise we need to enter the password every time we want to connect.

#### **Step 2.1.2: Create a "Certificate Signing Request".**

We would now like to use our CA key to sign Alex's public key. To do this, we create a "certificate signing request" or CSR:

```
openssl req -new -key alex.key -out alex.csr
```

Once again it will ask you for your country, organization name, etc. **The most important question is the "Common Name" field (circled in red below). Put in any domain name here EXCEPT what you used for the signer, BUT YOU MUST REMEMBER WHAT IT IS OR ACTIVITY 3 WILL FAIL. Here in this example we used mine.com.**

Leave the challenge password and optional company name blank.

```

ctank@Colins-MacBook-Pro keys % openssl req -new -key alex.key -out alex.csr
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]:SG
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) []:mine.com
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
ctank@Colins-MacBook-Pro keys %

```

Question 1:

Write down the Common Name that you used for Alex.

### Step 2.1.3: Signing Alex's Certificate. WHERE: On Alex's Pi

Since your CA key is on your laptop, you now need to transfer Alex's Certificate Signing Request over to your laptop. Using Cyberduck or other means, transfer alex.csr to your laptop, ensuring that it is in the same directory as the CA key and CA certificate you generated in Activity 1. **DO NOT TRANSFER alex.key!**

#### WHERE: On your laptop

Now on your laptop, ensure that alex.csr is in the same directory as signing.key and signing.pem that you generated in Activity 1. If so, we sign Alex's public key by doing (note: all in one line):

```
openssl x509 -req -in alex.csr -CA signing.pem -CAkey signing.key -CAcreateserial -out alex.crt -days 500 -sha256
```

Here's what's happening:

- x509: Create an X509 certificate.
- in: Specify input certificate signing request (CSR) file.
- CA: Specify CA's certificate.
- CAkey: Specify CA's private key.
- CAcreateserial: Creates a .srl file (in our case, signing.srl) containing the serial numbers of certificates you sign.
- out: Specifies the file to write the certificate to.
- days: Specifies how many days this certificate will last. It should not last longer than the key used to sign this certificate.
- sha256: Use SHA256 to generate the hash for the signature on the certificate.



When you hit enter, openssl will print out the certificate to be signed, and ask for your signing key password.

```
ctank@Colins-MacBook-Pro keys % openssl x509 -req -in alex.csr -CA signing.pem -CAkey signing.key -CAcreateserial -out alex.crt -days 500 -sha256

Certificate request self-signature ok
subject=C=SG, ST=Some-State, O=Internet Widgits Pty Ltd, CN=mine.com
Enter pass phrase for signing key:
ctank@Colins-MacBook-Pro keys %
```

In practice you would verify this information with the applicant through some means (e.g. asking for identification documents), but since this key was generated by you, just feel free to key in the password to your signing key. When you hit enter, it will produce the certificate in alex.crt.

You can see the contents of the certificate:

```
-----BEGIN CERTIFICATE-----
MIIFgGCCA2qgAwIBAgIUZaaUyQm0M+dnsfMAppzEKNsV+gwDQYJKoZIhvcNAQEL
BQAwWjELMAkGA1UEBhMCU0cxExARBgNVBAgMClNvbWUtU3RhdGUxITAFBgNVBAoM
GE1udG9yYmV0IFdpZGdpdHgUHR5IEx0ZDEtMBEGA1UEAwK2LnbnVybWVudG9y
Fw0yMDA2MjUxNTUwNDRAfW8yNTA3Mjg0NTEwNDRAfG9xChAJBgNVBAYTA1NHRMRW
EQYDVQIDApTb211LVN0YXR1MSEwHwYDVQQKDBhJbnR1cm5ldCBXaWRnaXRzIFB0
eSBMdG9xETAPBgNVBAMMCG1pbmUyZ9tMIIC1jANBgkqhkiG9w0BAQEFAAOCAg8A
MIICCgKCAgEAswwZw1YVa1IRLWB/kJXnYbYFp3gWMP7mGKCubGtCpQFRjesk+j0P
FAFIK5xYba4ASSIPvD8OFdQNC9Xp24dnWjCzvmQ4BcCC1gFEDsfqPmXkd2Nt9cU
MUUdUu1q3tZL10KtEzjuVGvDuxon11sP018z1echqcYbhxXN15Spz6u2Td1/h+S
SZCM1oXKq7IjCn8B30e1BMR1As/uf5x+w/La7HfML7QTKfzR8pVfdd01ZOPWK
E1U010500P1Dnc2HmCc0zj0aagUhh2+4zLjv0VRkRyRV41CY00SvBBe0/gB2et1
07Y0Snu/GER01T8NH0jvW2bWp2103SGKL0Xf8c0qgtndK3fz0y1HykVzh1XxUS
I2d8A1aMEQ19NHhWkcgP46z1j91a4FM+M0KDS+mdtweSqYB3fzJ0563JYrbw
IQVNA02+UvK15EqdFxbes8ufikgcZL64y0PgvVn+3Lmv7A2nvdtdKkdt1hhAAsTJ
KLVPehedXkCF0C9VerwL2ua3aE4fh9o1D1AC1hdwJ3JoaY1fHCDK56beP1/XIkH
OpxxWagb+/DUMPr3YukXwY3tswMakqB1f1kyt/mhQJzbU0b61ZntInOLKmSg0Q
Ek0dz1v6nJE+Zez2MizvFivtok1F09TqrDb0GkZVzFmhzMcoo2da8UCAwEAAaNC
MEAwHQYDVIR80BBYEFDSnaPFRZ3NpB2VOYCBEWm8/nGKzMB8GA1UdIwQYMBAAFFDW
pv/0N+sWV34RFRFNZ8gh1MLMA0GCSqGSb3DQEBQ0UAA4ICAQBWRPLB02QFRIB1
Cxs2fukP13AdPhC55NedwyzKavo5EaB5iyyvkBQ07ozeIu13Rbu0GtCXjC693T
zPHMe+4z3Ja+D2gwzT0m14DvKaip/oErdtVBMUyVwNkKZcyUirJJEChzEU+LbL
t5NktmfzKPsPZmC3T3e3X1V9qsdSB1ePMcLm1Ed0Q5SuneV0WUPBKMGK2sa341YU3
WNmsjo3E1j1yy46Wkyq03KobvzrUFS1+6UaqoN0ZQ56RrbXGURovRgW13n0T8A39
gfcZDgcKXKjHu0hUwSBtTqhb11XedgD/51MKYE8hVof684U1s8vRQdmo/mQ1BRe
x0JiutT/NlUoBLxvG9j33b0NntEgtX8cdvfyXqHAvpXjAcWc9YjmnNtob1q6ljX5
x1be2S3X4exFt15nC4n1NgMvOmZvDqOKkzajMIERUVg1QW0+ZtIXR05N1zUISe
2/chjijAdccNy/n5i3f8gorzmTjRXLRu4PK6K7HMcEG76901tFRNe8Ez2b6W3EtL
TGYPiOnbtbOnSn+btAuB+YSB0TfTwtV7ykkhFp4Hhdx+6I9cY1os2D0QDwT5Aa
PVZV2gi/cqj1bGfC8gTtJ176TrGUmFpW6jFq21ZoKndDTJ/hyysXrFASyUdBoMR
YgxFJrY9S80H3tD91cLhItAbt0JNw==
-----END CERTIFICATE-----
```

It looks very much like signing.pem, so don't confuse them. ;)

#### Step 2.1.4: Transferring Alex's certificate, and the CA certificate to the Pi

##### WHERE: On your laptop

Using Cyberduck or some other means, transfer BOTH alex.crt and signing.pem to the Pi. Ensure that it is in the same directory as alex.key. As per the lecture, TLS will send alex.crt over to your laptop during the handshake process. This allows you to certify that you are indeed connecting to Alex and not to someone pretending to be him.

##### Some Notes:

- Notice that you never need to generate a public key? This is actually done when generating the CSR. OpenSSL will use your private key to generate the public key and put it into the CSR file.
- When requesting for a key to be signed, you NEVER transfer the private key to the CA. You only transfer the CSR file. This is also true if you are using a proper CA like Verizon or Digicert.
- In fact, your private key should never leave your server (except when you are making a backup of the key : In which case ensure that your backup is secure.)
- Once the CSR file is signed to create the certificate (.crt) file, you need to transfer BOTH Alex's certificate, and the signing certificate back to Alex. Otherwise Alex is unable to verify the signature.

- Most web servers require your private key and certificate to be combined into a single file. To do this (on Alex's Pi):

```
cat alex.key alex.crt > alex.pem
```

As we are not running a web server in this studio, this step is not needed. As alex.pem contains your private key, it should never leave Alex's Pi.

## Step 2.2. GENERATING YOUR LAPTOP KEY

**WHERE: On your laptop. You must be running MACOS, Ubuntu, or the Windows LINUX subsystem.**

### **Step 2.2.1. Generating the Private Key for your Laptop**

As before, generate the private key:

```
openssl genrsa -out laptop.key 4096
```

### **Step 2.2.2. Generate a Certificate Signing Request**

Generate the CSR for your laptop for signing.

```
openssl req -new -key laptop.key -out laptop.csr
```

As before you must take note of the common name you use for your laptop. Here we are using "yours.com". You must not use the same common name as your signing certificate.

As before leave the challenge password and optional company name blank.

```
ctank@Colins-MacBook-Pro keys % openssl req -new -key laptop.key -out laptop.csr
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]:SG
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) []:yours.com
Email Address []:

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

Question 2:

Write down the Common Name for your laptop.

Copy laptop.csr to the same directory as your signing.key and signing.pem files generated in Activity 1.

### **Step 2.2.3. Sign the Key**

Sign the CSR to generate the certificate for your laptop (Note: All in one line)

```
openssl x509 -req -in laptop.csr -CA signing.pem -CAkey signing.key -CAcreateserial -out laptop.crt -days 500 -sha256
```

As before openssl will print out the details of the certificate, and you have to enter your signing key password to generate laptop.crt.

### **Step 2.2.4 Copy over the Certificate**

Copy the laptop.crt file to the same directory as laptop.key. When connecting to Alex, your laptop will send laptop.crt over to Alex, for Alex to verify your identity and ensure that you are allowed to connect to him. This prevents someone from kidnapping Alex.

You are now done setting up your keys for Alex and your laptop! Now let's start programming!

---

## **ACTIVITY 3: TLS PROGRAMMING**

---

Both TCP/IP and TLS programming are long, complicated and mechanical processes, so to save you some grief (and to give time to more interesting experiments), we have created a simple-to-use library that lets you create TLS clients and servers with just a few lines of code. This library is in the tls\_server\_lib.zip and tls\_client\_lib.zip files enclosed with this studio.

**Please see the optional “Under the Hood: TCP/IP and TLS Programming” document for a full(ish) tutorial on how to write TCP/IP and TLS code.**

### Step 3.0. SYSTEM SETUP

Begin by transferring the `tls_server_lib.zip` file to your Pi. You also need a copy of this zip file on your laptop.

#### **ON YOUR PI:**

##### **Step 3.0.1 Setting up the TLS Server Creation Library**

Unzip `tls_server_lib.zip` into the same directory as your `alex.key`, `alex.crt` and `signing.pem` files.

##### **Step 3.0.2 Installing TLS Programming Libraries.**

Install the standard TLS programming libraries in Ubuntu using:

```
sudo apt-get install libssl-dev
```

#### **ON YOUR LAPTOP:**

##### **Step 3.0.3 Installing the Client Creation Libraries**

Unzip `tls_client_lib.zip` into the same directory as your `laptop.key`, `laptop.crt` and `signing.pem` files.

##### **Step 3.0.4 Installing the TLS Programming Libraries**

You now need to install the standard TLS programming libraries. For Ubuntu / Windows LINUX

Subsystem:

```
sudo apt-get install libssl-dev
```

For MACOS:

You need to install `openssl-devel`:

```
brew install openssl-devel
```

You will also need to install the ommand line developer tools. To do so type:

```
Xcode-select --install
```

Note: There are two dashes (`--`) before the “install” keyword.

You will get a pop-up dialog box. Click “Install” (NOT Install XCode!). Now when you type:

```
g++
```

It should tell you:

```
ctank@Colins-MacBook-Pro client % g++
clang: error: no input files
ctank@Colins-MacBook-Pro client %
```

If you had installed Homebrew and openssl as per Step 1.0 above, you already have libssl. Unlike in Ubuntu, however, when you compile you need to explicitly specify the include directory using the `-I` (capital i, not small L) option, and the library directory using the `-L` option.

```
g++ -I /opt/homebrew/include/openssl -L /opt/homebrew/opt/openssl/lib <rest of
command> <rest of linker options>
```

You will see this later on.

We will now begin writing our TLS server called `test_tls_server` (Step 3.1) on the Pi, and our TLS client called `test_tls_client` (Step 3.2) on the laptop.

**Note:** Steps 3.1 and 3.2 and their sub-parts can be done together. You however cannot launch `test_tls_client` without first starting `test_tls_server`.

### Step 3.1. WRITING THE TLS SERVER

#### **WHERE: ON THE RASPBERRY PI**

The TLS Server Creation Library is found in `tls_server_lib.cpp`, `tls_common_lib.cpp`, `tls_pthread.cpp`, with the main server creation code in `make_tls_server.cpp`. It is linked to the openssl library that (`libssl-dev` on Ubuntu and `openssl` on MACOS) that you installed earlier.

The main function call is `createServer`:

```
void createServer(const char *keyFilename, const char *certFilename, int portNum,
void *(*workerThread)(void *), const char *caCertFilename, const char *peerName, int
verifyPeer);
```

This looks really complicated, and the following table explains what each parameter does:

Parameter Name	Description
<code>keyFilename</code>	Filename for Alex's private key (alex.key)
<code>certFilename</code>	Filename for Alex's certificate (alex.crt)
<code>portNum</code>	Port number to create the server on. The client will connect to this port.
<code>workerThread</code>	Pointer to the worker function, which handles all connections. The worker function must be declared as <code>void *funcname(void *conn)</code> , <code>funcname</code> can be any valid C function name, and <code>conn</code> can be any valid C variable name. We conventionally call this parameter <code>conn</code> because the library will pass the SSL connection through this parameter.

caCertFilename	Filename for the CA's certificate (signing.pem)
peerName	Name of the client that is connecting, as recorded in Question 2 above (Common Name for the laptop). Must be set if verifyPeer is 1.
verifyPeer	Set to 1 to verify the peer that is connecting, 0 otherwise.

createServer creates a loop to listen to portNum for a new connection, and when it gets one, it uses the worker function pointed to by workerThread to spin off a new thread that runs concurrently with the loop listening to portNum. This arrangement allows you to handle multiple connections at a time, and it is called a “multithreaded server”. Again see the Under the Hood document for details.

The worker function is declared as follows:

```
void *worker(void *conn) {
// conn = SSL connection established by createServer

... Code to handle the connection ...

EXIT_THREAD(conn);
}
```

The call to EXIT\_THREAD at the end of the worker ensures proper clean-up when the worker thread exits, and is necessary.

To write to the SSL connection:

```
int sslWrite(void *conn, const char *buffer, int len);
```

This writes len bytes from buffer to the SSL connection. The parameters and return values are:

Parameter Name	Description
conn	The SSL connection. This is passed in to your worker function by create_server.
buffer	The data to write to the SSL connection.
len	The number of bytes to write.
<b>RETURNS:</b>	Number of bytes actually written. Returns 0 if the connection is closed, or <0 if there has been an error. Use perror to print out the error.

To read from the SSL connection:

```
int sslRead(void *conn, char *buffer, int len);
```

This reads a maximum of len bytes of data from the SSL connection conn to buffer. The parameters and return values are:

Parameter Name	Description
conn	The SSL connection. This is passed in to your worker function by create_server.
buffer	Data from the SSL connection are stored here.
len	The maximum number of bytes to read.

<b>RETURNS:</b>	Number of bytes actually read. Returns 0 if the connection is closed, or <0 if there has been an error. Use perror to print out the error.
-----------------	--

Now that all these formalities are out of the way, let's actually start writing code!

### Step 3.1.1 Writing the TLS Server Code

SSH or VNC to your Pi, and switch to the directory where you unzipped `tls_server_lib.zip` to. Ensure that your `alex.crt`, `alex.key` and `signing.pem` files are there in the same directory as well.

We will now create an echo server. An echo server is one that listens to a port, then just echoes back everything it hears. There is a file called "`test_tls_server.cpp`" provided in the **`tls_server_lib.zip`** file. Use vim or nano to open it, and understand the code. In particular, see how to create a new server, and how to create the worker function. The listing is shown below.

**NOTE:** Replace the `#define CLIENT_NAME` with the Common Name of your Laptop as per Question 2 above. Ensure that `alex.key`, `alex.crt` and `signing.pem` are in the same directory.

```

#include "make_tls_server.h"
#include "tls_common_lib.h"

#include <stdio.h>

#define PORTNUM 5001
#define KEY_FNAME    "alex.key"
#define CERT_FNAME   "alex.crt"
#define CA_CERT_FNAME "signing.pem"
#define CLIENT_NAME   "laptop.epp.com"

// We are making an echo server. So we
// just echo back whatever we read.

void *worker(void *conn) {
    int exit = 0;

    while(!exit) {

        int count;
        char buffer[128];

        count = sslRead(conn, buffer, sizeof(buffer));

        if(count > 0) {
            printf("Read %s. Echoing.\n", buffer);
            count = sslWrite(conn, buffer, sizeof(buffer));

            if(count < 0) {
                perror("Error writing to network: " );
            }
        }
        else if(count < 0) {
            perror("Error reading from network: ");
        }

        // Exit if we have an error or the connection has closed.
        exit = (count <= 0);
    }

    printf("\nConnection closed. Exiting.\n\n");
    EXIT_THREAD(conn);
}

int main() {
    createServer(KEY_FNAME, CERT_FNAME, PORTNUM, &worker, CA_CERT_FNAME, CLIENT_NAME, 1);
    while(server_is_running());
}

```



This code creates a worker function “worker” that loops around listening to the TLS connection, and when it receives something it writes it back to the TLS connection to echo it. The loop exits if it reads or writes 0 or less bytes to the connection, meaning that either the connection has closed, or there was an error writing to it.

The main part is simple: It calls `createServer`, specifying the private key filename, certificate filename, port number, a pointer to the worker thread (notice that it passes in `&worker`, which means “pointer to worker”), the CA certificate filename, the Common Name for your laptop, and a “1” to enable client verification (i.e. to check the certificate for your laptop).

Notice that it creates the server at port 5001.

The main loop calls the `server_is_running()` function, which returns TRUE as long as the server loop is still running. This prevents `test_tls_server` from spawning the server threads and exiting, which will create zombie or orphaned threads.

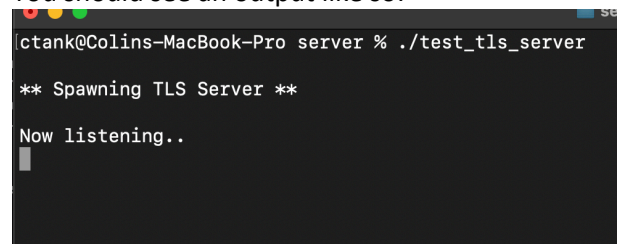
To compile, type, in a single line:

```
g++ test_tls_server.cpp tls_server_lib.cpp tls_pthread.cpp make_tls_server.cpp
tls_common_lib.cpp -pthread -lssl -lcrypto -o test_tls_server
```

To start the server, type:

```
./test_tls_server
```

You should see an output like so:

A terminal window with a dark background. The prompt is 'ictank@Colins-MacBook-Pro server %'. The command './test\_tls\_server' has been entered. The output shows two lines: '\*\* Spawning TLS Server \*\*' and 'Now listening..'. A cursor is visible on the line 'Now listening..'.

```
ictank@Colins-MacBook-Pro server % ./test_tls_server
** Spawning TLS Server **
Now listening..
```

### Step 3.1.2 Connecting to Your Server with a Browser

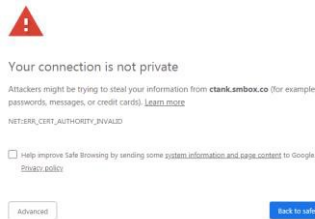
- Ensure that your laptop can connect to your Pi, via SSH or VNC (although we don’t need this at this step).
- Launch Chrome or some other decent browser that isn’t Edge nor Internet Explorer.

- In the address bar type <https://<ip address of pi>:5001> where <ip address of Pi> is the IP address of your Pi. So if your Pi's IP address is 192.168.0.103, you would type the following into the address bar:

<https://192.168.0.103:5001>

### Question 3

Do a screen capture of what you see in the browser, and on your Pi in the test\_tls\_server programme. Paste these into your report.



What do you think has happened? Why is there an error on your browser?

### Step 3.2. WRITING THE TLS CLIENT

#### WHERE: ON YOUR UBUNTU/MACOS/WINDOWS LINUX SUBSYSTEM LAPTOP

The TLS Client Creation Library is found in tls\_client\_lib.cpp, tls\_common\_lib.cpp, tls\_pthread.cpp, with the main server creation code in make\_tls\_client.cpp. It is linked to the openssl library that (libssl-dev on Ubuntu and openssl on MACOS) that you installed earlier.

The main function call is createClient:

```
void createClient(const char *serverName, const int serverPort, int verifyServer,
const char *caCertFname, const char *serverNameOnCert, int sendCert, const char
*clientCertFname, const char *clientPrivateKey, void *(*readerThread)(void *), void
*(*writerThread)(void *));
```

This looks really complicated, and the following table explains what each parameter does:

Parameter Name	Description
serverName	The host name or IP address of your server. In this case it will be the IP address of Alex's Raspberry Pi.
serverPort	The port of the server we are connecting to on the Pi. In our example it will be port 5001, since that the port number we called createServer with in Step 3.1.1.
verifyServer	Set this to 1 to verify the server's identity, or 0 to not identify. If set to 1, caCertFname must hold the filename of the CA certificate file.

caCertFname	The filename of the CA certificate, in our case it would be signing.pem. Must be set if verifyServer is 1.
serverNameOnCert	The Common Name of the server, as stated in Question 1.
sendCert	Set to 1 to send the client's certificate over to the server, if the server requests.
clientCertFname	Filename of the client's certificate. Must be set if sendCert is 1.
clientPrivateKey	Filename of the client's private key. Must be set if sendCert is 1.
readerThread	Pointer to the reader function, which receives data from the SSL connection.
writerThread	Pointer to the writer function, which sends data to the SSL connection.

Notice that unlike createServer which creates just a single worker function to deal with reading and writing to the SSL connection, createClient actually requires two functions: a reader function to read the SSL connection, and a writer function to write to the SSL connection.

The reader and writer functions are declared as follows:

```
void *reader(void *conn) {
    // conn = SSL connection established by createClient

    ... Code to handle the connection ... stopClient();
    EXIT_THREAD(conn);
}

void *writer(void *conn) {
    // conn = SSL connection established by createClient

    ... Code to handle the connection ... stopClient();
    EXIT_THREAD(conn);
}
```

Both the reader and writer functions terminate with EXIT\_THREAD, which is necessary to ensure proper cleanup before quitting. They also call stopClient, which stops the main client loop in createClient.

To write to the SSL connection:

```
int sslWrite(void *conn, const char *buffer, int len);
```

This writes len bytes from buffer to the SSL connection. The parameters and return values are:

Parameter Name	Description
conn	The SSL connection. This is passed in to your worker function by create_server.
buffer	The data to write to the SSL connection.
len	The number of bytes to write.
<b>RETURNS:</b>	Number of bytes actually written. Returns 0 if the connection is closed, or <0 if there has been an error. Use perror to print out the error.

To read from the SSL connection:

```
int sslRead(void *conn, char *buffer, int len);
```

This reads a maximum of len bytes of data from the SSL connection conn to buffer. The parameters

and return values are:

Parameter Name	Description
conn	The SSL connection. This is passed in to your worker function by <code>create_server</code> .
buffer	Data from the SSL connection are stored here.
len	The maximum number of bytes to read.
<b>RETURNS:</b>	Number of bytes actually read. Returns 0 if the connection is closed, or <0 if there has been an error. Use <code>perror</code> to print out the error.

Now that all these formalities are out of the way, let's actually start writing code!

### Step 3.2.1 Writing the TLS Client Code

We will now create a client for our echo server. The writer function reads from the keyboard and writes to the SSL connection, while the reader function reads from the SSL connection and prints to the screen.

(Aside: This is actually why we have separate reader and writer functions: Both reading the keyboard and reading the SSL connection are "blocking" calls, i.e. they will not exit until data actually comes in. If we used a single worker function, we can't read the SSL connection until we enter something on the keyboard, and then we can't read the keyboard until the client receives something from the SSL connection. This is usually an unworkable system.)

There is a file called "test\_tls\_client.cpp" provided in **tls-client-lib.zip**, which contains our client code. Use `vim` or `nano` to open it, and look through the code to understand it. In particular, look at how we create a new client, and how we create the reader and writer functions. The listing is shown below.

**NOTE:** Replace `#define SERVER_NAME` with the IP address of your Pi, and `#define SERVER_NAME_ON_CERT` with the Common Name of the server, as per Question 1, above.

```

#include "make_tls_client.h"

// Our reader thread just reads the socket connection and prints it
void *readerThread(void *conn) {
    int exitReader = 0;

    while(!exitReader) {
        char buffer[128];
        int len = sslRead(conn, buffer, sizeof(buffer));

        if(len < 0) {
            perror("Error reading socket: ");
        }

        if(len > 0) {
            printf("\nReceived: %s\n", buffer);
        }

        exitReader = (len <= 0);
    }

    printf("Server closed connection. Exiting.\n");
    stopClient();
    EXIT_THREAD(conn);
}

void *writerThread(void *conn) {
    int exitWriter = 0;

    while(!exitWriter) {
        char buffer[128];
        printf("Input: ");
        fgets(buffer, 128, stdin);
        printf("\n");

        int len = sslWrite(conn, buffer, sizeof(buffer));

        if(len < 0) {
            perror("Error writing to server: ");
        }

        exitWriter = (len <= 0);
    }

    printf("writer: Server closed connection. Exiting.\n");
    stopClient();
    EXIT_THREAD(conn);
}

#define SERVER_NAME "localhost"

```

```

#define PORT_NUM 5001
#define CA_CERT_FNAME "signing.pem"
#define CLIENT_CERT_FNAME "laptop.crt"
#define CLIENT_KEY_FNAME "laptop.key"
#define SERVER_NAME_ON_CERT "alex.epp.com"

int main() {

    createClient(SERVER_NAME, PORT_NUM, 1, CA_CERT_FNAME, SERVER_NAME_ON_CERT, 1,
CLIENT_CERT_FNAME, CLIENT_KEY_FNAME, readerThread, writerThread);

    while(client_is_running());
}

```

Notice that the main() calls client\_is\_running() inside a while loop. This function returns true as long as the main client loop is running, and prevents test\_tls\_client.cpp from exiting after spawning the client threads, to prevent creation of zombie and/or orphaned threads.

To compile test\_tls\_client, type the following in a single line:

```

g++ test_tls_client.cpp make_tls_client.cpp tls_client_lib.cpp tls_pthread.cpp
tls_common_lib.cpp -pthread -lssl -lcrypto -o test_tls_client

```

If you are using MACOS (again in a single line):

```

g++ -I/opt/homebrew/include -L /opt/homebrew/opt/openssl/lib test_tls_client.cpp
make_tls_client.cpp tls_client_lib.cpp tls_pthread.cpp tls_common_lib.cpp -pthread -
lssl -lcrypto -o test_tls_client

```

To run, ensure that test\_tls\_server is running on the Pi, ensure that laptop.key, laptop.crt and signing.pem are in the same directory as test\_tls\_client, then on your laptop type:

./test\_tls\_client

If everything goes well, you will see:

```

ctank@Colins-MacBook-Pro client % ./test_tls_client
Host 127.0.0.1 IP address is 127.0.0.1
CERTIFICATE DATA:
C=SG, ST=Some-State, O=Internet Widgits Pty Ltd, CN=mine.com

SSL SERVER CERTIFICATE IS VALID
Input: █

```

Type in "Hello World" and you will see:

```

ctank@Colins-MacBook-Pro client % ./test_tls_client
Host 127.0.0.1 IP address is 127.0.0.1
CERTIFICATE DATA:
C=SG, ST=Some-State, O=Internet Widgits Pty Ltd, CN=mine.com

SSL SERVER CERTIFICATE IS VALID
Input: Hello World

Input:
Received: Hello World

```

Notice it shows Input:, then a line break, then Received: Hello World. This is because both the reader and writer functions are running concurrently and overwriting each other's output on the screen. You can continue to type, and when you hit Enter you will see what you typed being echoed back to you.

Now let's try some experiments!

### STEP 3.3. MESSING AROUND WITH TLS

We will now do two experiments to show the value of having signed certificates.

#### **Step 3.3.1 Fake Server**

##### **WHERE: ON THE RASPBERRY PI**

We will now simulate what happens if someone tries to impersonate Alex. The situation is that Eve has created her own Alex (named Fake Alex) and wants to con you into controlling Fake Alex so that she can commit nefarious acts with it.

Of course she can't possibly access Alex's private key (you didn't push it onto a repo, did you?), and without it, having Alex's certificate is useless because she can't decrypt anything encrypted with the public key in Alex's certificate. So she has to generate a new set of keys.

And of course you (the CA) is not going to sign her fake keys, so she also has to create her own CA keys and certificates.

Since you only have one Pi, we will simulate this fakery on that one Pi.

##### **IMPORTANT:**

**THIS STEP SHOULD NOT BE DONE BY THE SAME PERSON WHO CREATED THE CA KEY AND CERTIFICATE! DO NOT OVERWRITE YOUR CURRENT SIGNING.KEY AND SIGNING.PEM!**

- i) Appoint one team member to be Eve. **To ensure that you do not corrupt your actual CA key and certificate, this must not be the same person who created the original signing.key and signing.pem,**
- ii) Rename alex.key to alex.key.old, and alex.crt to alex.crt.old.
- iii) Rename signing.pem to signing.pem.old.
- iv) Get Eve to produce a fake CA key and certificate:

```
openssl genrsa -des3 -out signing.key 4096
openssl req -x509 -new -nodes -key signing.key -sha256 -days 1024 -out signing.pem
```

- v) Now Eve should produce a fake key pair for Fake Alex, and use her fake CA key and certificate to sign it:

```
openssl genrsa -out alex.key 4096
openssl req -new -key alex.key -out alex.csr
```

(Note: Use the exact same Common Name as you did in Question 1)

```
openssl x509 -req -in alex.csr -CA signing.pem -CAkey signing.key -CAcreateserial -out alex.crt -days 500 -sha256
```

- vi) Make sure that the new alex.key, alex.crt and signing.pem are in the same directory as test\_tls\_server.
- vii) Now restart ./test\_tls\_server on the Pi first, then run ./test\_tls\_client on the laptop.

#### Question 4.

Describe what happens when test\_tls\_client is run. What error code is produced? Google and describe what this error code means.

The following two steps are very important: if you do not do them the rest of your studio will fail.

- viii) DELETE alex.key, alex.crt and signing.pem.
- ix) RENAME alex.key.old to alex.key, alex.crt.old to alex.crt, and signing.pem.old to signing.pem.

### Step 3.3.2 Fake Client

#### WHERE: ON YOUR LAPTOP

Eve wasn't able to get you to control her Fake Alex, so now she has come up with a scheme to hijack Alex! She was able to trick you (haha) to sign a fake key certificate. Unfortunately, you refuse to sign the certificate if she uses the same Common Name as your laptop (Question 2), so she has created a new key pair, but under the Common Name of "evil.eve.com". To try to fool you she will connect to the server using the Common Name you created in Question 2.

To simulate this fakery:

- i) Appoint the same team member to be Eve.
- ii) Go to the same directory where you compiled test\_tls\_client. **Rename laptop.key and laptop.crt on your laptop (important!)**
- iii) Get Eve to generate new laptop.key and laptop.csr files:

```
openssl genrsa -out laptop.key 4096
openssl req -new -key laptop.key -out laptop.csr
```

**(Note: Use the "evil.eve.com" as the Common Name)**

- iv) Eve sends the CSR to the CA to sign, who (unwittingly) does so. **Note: This must be the actual CA who created the CA certs at the start of the studio, not Eve!**

```
openssl x509 -req -in laptop.csr -CA signing.pem -CAkey signing.key -CAcreateserial -out laptop.crt -
days 500 -sha256
```

Hiak hiak.. now she has a fake key and certificate to access Alex! Or so she thinks!

- v) Ensure that the new (fake) laptop.crt and laptop.key files are in the same directory as test\_tls\_client.
- vi) Start ./test\_tls\_server on the Pi, and ./test\_tls\_client on the laptop.



Question 5.

Describe what happens when `test_tls_client` is run. What error code is produced? Google and describe what this error code means.

- vii) DELETE `laptop.key` and `laptop.crt`.
- viii) Rename `laptop.key.old` to `laptop.key` and `laptop.crt.old` to `laptop.crt`

---

#### ACTIVITY 4 – ALEX TO THE CLOUDS!

---

In this Activity we will bring Alex to the clouds! Steps 4.1 and 4.2 are fairly similar, and use what you have learnt in Activity 3 to implement a client-server system that lets you control Alex from your laptop!

The server is found in `tls-alex-server.cpp` from the `tls-server-lib.zip` file, to be run on the Pi. This is a networked version of the `alex-pi.cpp` programme from Week 8 Studio 2, with commands like move forward 50 cm at 80% being received over a TLS connection instead of directly from the terminal on the Pi.

The client is found in `tls-alex-client.cpp` from the `tls-client-lib.zip` file. This reads commands from the keyboard *on your laptop* and sends it over to the server on the Pi, allowing you to control Alex from your laptop.

**NOTE: Steps 4.1 and 4.2 can be done at the same time, although you must launch the server in Step before the client in Step 4.2.**

**Also ensure that the completed Alex firmware is running on the Arduino.**

##### Step 4.1. BUILDING THE ALEX TLS SERVER

**WHERE: On the Raspberry Pi.**

For example, you may see:

```
/* TODO: Create an integer variable x and set it to 1 */  
/* END TODO */
```

Your solution will look like this:

```
/* TODO: Create an integer variable x and set it to 1 */ int x = 1;  
  
/* END TODO */
```

See Activity 3 on how to do each part, e.g. how to create a new server, how to read/write a TLS connection, etc.

Question 6.

Cut and paste all the TODO sections into your report, and explain the changes you have made.

- i. Compile using (note, all in a single line):

```
g++ tls-alex-server.cpp tls_server_lib.cpp tls_pthread.cpp  
make_tls_server.cpp tls_common_lib.cpp serial.cpp serialize.cpp -  
pthread -lssl -lcrypto -o tls-alex-server
```

- ii. **Ensure that your ORIGINAL alex.key, alex.crt and signing.pem (not the fake certificates generated by Eve) are present in the same directory as tls-alex-server, and launch tls-alex-server. See test\_tls\_server.cpp on how to create a server and load the certificates, keys, etc.**

Step 4.2. BUILDING THE ALEX TLS CLIENT

**WHERE: On your LINUX / MACOS / Windows Linux Subsystem Laptop**

- i. Switch to the directory where you unzipped tls-client-lib.zip to. Ensure that the ORIGINAL laptop.key and laptop.crt (not the fake ones made by Eve) are there, together with signing.pem.
- ii. Use vim to open the tls-alex-client.cpp file.
- iii. Locate all the TODO markers. Follow the instructions on the TODO line, between the TODO and END TODO markers.

For example, you may see:

```
/* TODO: Create an integer variable x and set it to 1 */  
/* END TODO */
```

Your solution will look like this:

```
/* TODO: Create an integer variable x and set it to 1 */ int x = 1;  
  
/* END TODO */
```

See Activity 3 on how to do each part, e.g. how to create a new server, how to read/write a TLS connection, etc.

Question 7.

Cut and paste all the TODO sections into your report, and explain the changes you have made.

- iv. Compile using (note, all in a single line):

```
g++ tls-alex-client.cpp make_tls_client.cpp tls_client_lib.cpp tls_pthread.cpp  
tls_common_lib.cpp -pthread -lssl -lcrypto -o tls-alex-client
```

On MACOS (Again, all in a single line):

```
g++ -I/opt/homebrew/include -I/opt/homebrew/opt/openssl/lib tls-alex-client.cpp  
make_tls_client.cpp tls_client_lib.cpp tls_pthread.cpp tls_common_lib.cpp -pthread -
```

```
lssl -lcrypto -o tls-alex-client
```

- v. **Ensure that your ORIGINAL laptop.key, laptop.crt (not the fake certificates generated by Eve) and signing.pem files are present in the same directory as tls-alex-server, and launch tls-alex-client. See test\_tls\_client.cpp on how to load up the keys, server name, etc.** You should see messages from both tls-alex-server (on the Pi) and tls-alex-client (on the laptop) showing that they are connecting.
- vi. You can now send commands from your laptop to control Alex! :D

---

### OPTIONAL ACTIVITY – NETWORK TOOLS

---

**WHERE: On your Ubuntu / MACOS / Windows Linux Subsystem laptop, or on the Pi if it has internet access**

The aim of this activity is just to get you familiar with three rather useful command-line tools. There are many more useful tools (e.g. ifconfig, which you used to determine Vincent's IP address, route which lets you configure your computer as a router, tcpdump that lets you capture TCP/IP data, etc), but these are outside the scope of EPP.

We will focus on just three of these tools:

ping: Lets you check if a connection is working.

traceroute: Lets you trace how a packet is being delivered from your machine to a destination. netcat: The Swiss Army Knife of the networking world. Lets you do all sorts of cool stuff.

#### Step 1.

When we first set up a network connection we often want to know if the connection actually works. The LINUX shell provides us with a very useful tool called "ping" that helps us to test that connection:

- a. Your local machine has a special host name called "localhost" and a special IP address 127.0.0.1
- b. To begin, you can ping yourself: ping localhost

OR:

ping 127.0.0.1

If your TCP/IP is set up right on your computer, you will see an output like this:

```

PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.038 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.070 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.064 ms
64 bytes from localhost (127.0.0.1): icmp_seq=4 ttl=64 time=0.074 ms
64 bytes from localhost (127.0.0.1): icmp_seq=5 ttl=64 time=0.063 ms
64 bytes from localhost (127.0.0.1): icmp_seq=6 ttl=64 time=0.071 ms
64 bytes from localhost (127.0.0.1): icmp_seq=7 ttl=64 time=0.069 ms
64 bytes from localhost (127.0.0.1): icmp_seq=8 ttl=64 time=0.068 ms
64 bytes from localhost (127.0.0.1): icmp_seq=9 ttl=64 time=0.077 ms
64 bytes from localhost (127.0.0.1): icmp_seq=10 ttl=64 time=0.064 ms
64 bytes from localhost (127.0.0.1): icmp_seq=11 ttl=64 time=0.064 ms
64 bytes from localhost (127.0.0.1): icmp_seq=12 ttl=64 time=0.058 ms
64 bytes from localhost (127.0.0.1): icmp_seq=13 ttl=64 time=0.073 ms
64 bytes from localhost (127.0.0.1): icmp_seq=14 ttl=64 time=0.050 ms
64 bytes from localhost (127.0.0.1): icmp_seq=15 ttl=64 time=0.064 ms
64 bytes from localhost (127.0.0.1): icmp_seq=16 ttl=64 time=0.066 ms

```

Hit CTRL-C to exit from ping.

The first column tells us the size of the ping packet (64 bytes), the second tells us the IP address of the system responding to our pings, the third is an ICMP (Internet Control Message Protocol) sequence number – which is basically just what the name suggests it is. The fourth column shows the packet's Time To Live (TTL). The TTL is set at 64 “hops”. A “hop” is a single point-to-point transmission of a packet (e.g. from one router to the next along a route).

The TTL is decremented by at least 1 at each router a packet passes through, and it prevents a packet from circulating indefinitely inside the network.

The last column shows the “round trip time” it takes for a packet to go to the destination (127.0.0.1 in this case) and back to the sender.

When you hit CTRL-C, ping tells you how many packets were sent, how many were received, the percentage of packets that were lost, the total time taken, and the minimum, average, maximum and standard-deviation round-trip times.

```

--- localhost ping statistics ---
17 packets transmitted, 17 received, 0% packet loss, time 16000ms
rtt min/avg/max/mdev = 0.038/0.064/0.077/0.011 ms

```

Now let's try something more exciting: ping [www.google.com](http://www.google.com)

You will see an output like this:

```

PING www.google.com (74.125.200.106) 56(84) bytes of data.
64 bytes from sa-in-f106.1e100.net (74.125.200.106): icmp_seq=1 ttl=48 time=1.70 ms
64 bytes from sa-in-f106.1e100.net (74.125.200.106): icmp_seq=2 ttl=48 time=1.27 ms
64 bytes from sa-in-f106.1e100.net (74.125.200.106): icmp_seq=3 ttl=48 time=1.30 ms
64 bytes from sa-in-f106.1e100.net (74.125.200.106): icmp_seq=4 ttl=48 time=1.26 ms
64 bytes from sa-in-f106.1e100.net (74.125.200.106): icmp_seq=5 ttl=48 time=1.29 ms

```

Notice that the round-trip-time (last column) is now significantly larger. This is expected since the packets are now traveling a much greater distance through many more routers.

Indeed the averages show us this:

```

--- www.google.com ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14019ms
rtt min/avg/max/mdev = 1.186/1.240/1.277/0.038 ms

```

The average round-trip-time (rtt) within localhost was 0.064ms, but to Google it is now 1.24 ms. The standard deviation is also larger at 0.038ms against 0.011 ms for localhost, reflecting more uncertainty in round-trip-times, likely caused by packets taking different routes each time through the internet.

Ping is really useful for debugging a bad connection. For example if you can ping hosts that are on the same network as you but cannot ping hosts outside of your network, there's a very high chance that your network gateway – the special computer that connects you to the outside world – is down, or that your computer is misconfigured and is using the wrong network gateway.

If your connection is bad you will see error messages like:

```

PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
From 192.168.0.1 icmp_seq=1 Destination Net Unreachable
From 192.168.0.1 icmp_seq=2 Destination Net Unreachable
From 192.168.0.1 icmp_seq=3 Destination Net Unreachable
From 192.168.0.1 icmp_seq=4 Destination Net Unreachable
From 192.168.0.1 icmp_seq=5 Destination Net Unreachable
From 192.168.0.1 icmp_seq=6 Destination Net Unreachable
From 192.168.0.1 icmp_seq=7 Destination Net Unreachable
^C
--- 8.8.8.8 ping statistics ---
7 packets transmitted, 0 received, +7 errors, 100% packet loss, time 6256ms

```

When you see a message like “Destination Net Unreachable” it strongly suggests that your network is not configured properly or that your connection is down. Notice that we have 100% packet loss.

On the other hand if your network is fine but the host is down, you will see something like this:

```

From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=1 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=2 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=3 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=4 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=5 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=6 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=7 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=8 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=9 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=10 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=11 Destination Host Unreachable
From bb219-75-76-98.singnet.com.sg (219.75.76.98) icmp_seq=12 Destination Host Unreachable

```

You may also see ping timeout if a host is down. If your network programs run very slowly, you can use ping to determine whether you have a slow network connection by using the rtt column. There have been cases of rtt times exceeding 1 second (!!!).

You can of course use ping Vincent to check that your connection works.

**NOTE: For security reasons many networks are configured to drop ICMP packets, so a failure like the one shown above may not necessarily indicate that the host is down.**

### Step 2.

Ping tells you if a connection is alive. If you have a slow or dead connection, ping cannot tell you where the problem is. For that you can use a second tool called traceroute. In the LINUX or MACOS shell, type (Note: type “sudo apt-get install traceroute” if LINUX tells you that you do not have traceroute installed):

traceroute [www.facebook.com](http://www.facebook.com)

When you do this you will get an output like so:

```

traceroute to www.facebook.com (157.240.13.35), 30 hops max, 60 byte packets
 1 188.166.176.253 (188.166.176.253) 1.032 ms 1.019 ms 1.012 ms
 2 138.197.250.210 (138.197.250.210) 1.122 ms 138.197.250.208 (138.197.250.208) 1.389 ms 138.197.250.206 (138.197.250.206) 0.617 ms
 3 32934.sgw.equinix.com (27.111.228.65) 15.098 ms 138.197.245.14 (138.197.245.14) 1.182 ms 32934.sgw.equinix.com (27.111.228.94) 2.205 ms
 4 * po161.asw02.sin4.tfbnw.net (157.240.41.32) 1.813 ms 1.807 ms
 5 po222.psw04.sin6.tfbnw.net (157.240.41.187) 1.639 ms po121.asw01.sin1.tfbnw.net (31.13.30.80) 2.199 ms po181.asw01.sin4.tfbnw.net (74.119.79.182) 1.904 ms
 6 po241.psw04.sin6.tfbnw.net (157.240.32.239) 1.769 ms 157.240.36.99 (157.240.36.99) 1.300 ms 157.240.36.95 (157.240.36.95) 1.071 ms
 7 157.240.36.139 (157.240.36.139) 1.348 ms edge-star-mini-shv-02-sin6.facebook.com (157.240.13.35) 1.085 ms 1.181 ms

```

(Sorry that’s a little small)

Traceroute shows us a list of routers and gateways that a packet passes through to get from your computer to Facebook, and the rtt for that router to your computer.

The first column is the “hop” number, meaning how many routers/gateways the packet has passed through. The next few columns of each hop show the routers that the packet passed through, and the rtt to that router.

But hang on; do you notice that you often have more than one router being shown on a single hop? This is because traceroute actually sends out THREE (3) packets, not one, and each packet may traverse over a different router.

```

7 173.252.67.15 (173.252.67.15) 1.332 ms 173.252.67.163 (173.252.67.163) 1.644 ms edge-star-mini-shv-01-sin6.facebook.com (157.240.7.35) 1.411 ms

```

You may need to zoom in, but the line above shows what we mean; in hop 7 the first packet went through 173.252.67.15 with a rtt of 1.332 ms, the second packet went through 173.252.67.163 with a rtt of 1.644, and the third packet went through 157.240.7.35 with a rtt of 1.411 ms.

You can run:

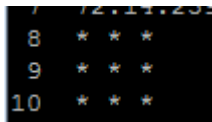
tracert [www.facebook.com](http://www.facebook.com)

Several more times and you will see that each time the packets pass through different gateways and routers. This adds robustness to the internet. If any (or a big bunch) of routers go down, packets will still find their way to the destination somehow.

In fact there have been times in the past where cables were accidentally cut, but the Internet continued working, albeit somewhat slower since the packets were forced to take less efficient routes:

<http://www.computerweekly.com/news/450431205/Australia-Singapore-submarine-cable-connection-cut-again>

Sometimes you will see rows of stars:



This means that packets for these hops took an unusually long time to traverse between hops.

The traceroute command is very useful for diagnosing slow connections, or to find where a connection may have broken.

### Step 3.

Now let's look at the most interesting of the 3 commands: netcat. Netcat (nc) is known as the "Swiss Army Knife" of the networking world because it can do a very wide range of things; it can work as a server, as a client, as a file transfer program, as a port scanner (to detect open ports in a host), and even as a crude web server! Let's explore some of these features of netcat.

#### a. netcat Client/Server

Open up two shells. On one shell we will run netcat as a server on port 8877: `nc -l 8877`

This runs netcat as a server at port 8877. You will see nothing happening. This is fine since netcat is waiting for a connection.

On the second shell we will run netcat as a client:

nc localhost 8877

This runs netcat as a client connecting to 8877 on localhost. Again you will see nothing happening. This really isn't very exciting. But now in the second shell (where you are running netcat as a client), type in "HELLO WORLD!"

You will now see whatever you type in the client side appearing in the server side! Magic!

#### b. netcat File Copy

Well ok that was kind of silly. Let's try something better. On two laptops running Ubuntu or MACOS type:

ifconfig

Note the IP addresses of both laptops.

Now let's do the following. We will call one laptop A and the other B:

Step Number	Laptop A	Laptop B
1	Type:  echo "hello cg1112 is so darn cool!" > hello.txt	Type:  rm -f hello.txt to ensure that you do not have hello.txt in your directory.
2		nc -l 8877 > hello.txt
3	nc <laptop B IP address> 8877 < hello.txt  E.g.  nc 172.88.151.251 8877 < hello.txt  (This is an EXAMPLE. The actual IP address would be different.	
4		cat hello.txt

Once laptop A is done, the hello.txt file should be transferred over to laptop B. Now isn't that pretty awesome?

#### c. Netcat as a Port Scanner

**NOTE: Do not scan the ports of any systems you do not own. You are likely to trigger a security alert and get banned from those systems. You may also get the whole of NUS banned!**

You can use netcat to see what ports are available on your system. To do so, type:



```
nc -z -v localhost 1-1000
```

This means to do a port scan (-z) with verbose output (-v) – meaning that netcat will return us more information than norm – on localhost, from ports 1-1000.

You will see that for the most part all the ports are closed (zzz). Let's use netcat to open a server on port 8877 and try again. Open up two shells. On the first shell type:

```
nc -l 8877
```

This will start a server on port 8877. On the second shell type:

```
nc -z -v localhost 8800-8899 | less
```

Now you will see:

```
nc: connect to localhost port 8876 (tcp) failed: Connection refused
nc: connect to localhost port 8877 (tcp) failed: Connection refused
Connection to localhost 8877 port [tcp/*] succeeded!
nc: connect to localhost port 8878 (tcp) failed: Connection refused
nc: connect to localhost port 8879 (tcp) failed: Connection refused
```

This shows us that port 8877 has a service running on it.