THE UNIVERSITY OF WAIKATO
Department of Computer Science

COMP204-24B — Practical Networking & Cyber Security

A4 – Transport Layer Security

---

1 Introduction

This assignment will introduce you to programming network applications with Java using Transport Layer Security (TLS). You will do so by writing a secure file transfer system that allows a client to securely request a file from an authenticated server. Please have a look at Lecture 4C (Certificates, TLS) beforehand as it provides an overview of concepts and algorithms used by TLS.

2 Academic Integrity

The files you submit **must** be your own work. You may discuss the assignment with others *but the actual code you submit must be your own.* You must fully also understand your code and be capable of reproducing and modifying it. If there is anything in your code that you do not understand, seek help until you do.

You **must** submit your files to Moodle by the deadline **and** demonstrate your code in the lab to one of the teaching staff within one week of the deadline to have the marks counted.

This assignment is worth 5% of your final grade.

3 Overview

There are five steps. The aim of steps 1 and 2 is to create a trusted certificate for your server to present to the client during the TLS handshake. In step 1, you create a *certificate authority* (CA) that is trusted by your client. In step 2, you create a *signed certificate* and *keys* for your server.

The final result is two Java **KeyStores** (.jks), one for the client and the other for the server:

1. `ca-cert.jks`: a Java KeyStore containing the public key certificate of your Certificate Authority as a trusted root. You can then supply your Java client application with this KeyStore so that it trusts certificates issued by the CA.

2. `server.jks`: a Java KeyStore containing the keys and certificates that the server requires, namely:

    - the server's private key;
    - the server's public key certificate, signed by the trusted CA; and
    - the CA's public certificate (to complete the chain to the root certificate).

In steps 3, 4, and 5, you will write the Java TLS server and client applications.

Throughout this assignment, you will be setting passwords and passphrases that protect various files. Keeping track of these is difficult, and nothing important will be protected, so we ask that you to store them in a text (markdown) file, which we provide here. You need to submit this text file as part of your assignment submission.

4 Step 1: Prepare your Certificate Authority's Signing Key and Public-key Certificate

Validating a certificate requires a path to a trusted root certificate. In the usual case, this would mean using the root certificate store that ships with Java. To obtain a signature for your server certificate would require you to select a certificate authority (CA) and pay them to sign your cert, which we do not want to do for this assignment. Instead, you will create *your own* CA's private key as a signing key and public-key certificate and tell your Java application that anything signed by your CA has a valid certificate path. To create a signing key and a public-key certificate for the fake CA:

```
openssl req -new -x509 -keyout ca-private.pem -out ca-cert.pem -days 3650
```

This command will create your CA's private key (i.e. signing key, stored in `ca-private.pem`) that we will use to sign the public-key certificate of the server in Step 2 and your CA's public-key certificate (`ca-cert.pem`) that your client uses to validate your server's public-key certificate. It will be valid for 3650 days (about 10 years). `openssl` will ask you to enter a passphrase to protect the `ca-private.pem` file. You will need this passphrase to decode the private key when you later sign your server certificate. Use a toy passphrase (i.e. not a password that you actually use) and write it down in the markdown file provided in the Overview section (**row A**).

You will also be prompted for other fields to be incorporated into your CA's certificate. For Country Name, enter "NZ", for State Name enter "Waikato", for Locality "Hamilton", for Organization Name pick a name for your CA, for Unit Name enter nothing, for Common Name enter your name, for Email Address enter an email address for yourself. Here is one sequence of steps; of course, you will choose different names than below.

```
Country Name (2 letter code) [AU]:NZ
State or Province Name (full name) [Some-State]:Waikato
Locality Name (eg, city) []:Hamilton
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Marinho's Trusty CA
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:Marinho Barcellos
Email Address []:marinho.barcellos@waikato.ac.nz
```

At this point, you will have a private signing key (`ca-private.pem`), and your CA's public certificate (`ca-cert.pem`) that you can distribute to enable validation of your server public-key. However, to use the certificate with Java, we have to first put it into a Java KeyStore. Java provides the command, called `keytool`, for this.

```
keytool -import -trustcacerts -alias root -file ca-cert.pem -keystore ca-cert.jks
```

You have to provide a password for the KeyStore, even though it is simply storing public information. Record the password in the text file provided in the Overview section (**row B**).

You will be asked to trust the certificate. Type "yes".

If you need to check the information within a certificate, or Java KeyStore, use the commands below.

Check a certificate:

```
keytool -printcert -v -file ca-cert.pem
```

Record the CA's certificate fingerprint (SHA256) in the text file provided in the Overview section (**row C**).

Check which certificates are in a Java KeyStore

```
keytool -list -v -keystore ca-cert.jks
```

You should see one entry (i.e. trustedCertEntry) and the Certificate fingerprint (SHA-256) that should be the same as the one that you just recorded.

5 Step 2: Prepare the Server's key pair and Public-key certificate

In the next step, you will generate a server key-pair, public-key certificate and sign it with your CA's private key created in Step 1, so that clients will have a trusted certificate path. Your server will send the signed certificate during the TLS handshake. Your server's public-key certificate needs to correspond to the name of the machine on which you run your server. For example, if you are running your code on the machine lab-rg06-01.cms.waikato.ac.nz then

your server's certificate needs to have that name in it. We will use Java's `keytool` to generate the key pair. The key pair is stored in a Java KeyStore file containing both the public key and the private key.

```
keytool -genkeypair -alias lab-rg06-01.cms.waikato.ac.nz \
   -keyalg RSA -keystore server.jks
```

You will be prompted for a password to protect the KeyStore – in particular, to protect the private key in that file. Choose a different password than above, and record the password in the text file provided in the Overview section (**row Drow D**).

You will also be prompted for "your first and last name". This is actually prompting for the certificate's "common name" which is the host that we are signing for. In this case, we are signing for the name of the server, so enter the server's hostname (the *same* value as in the alias parameter above!). Here's one possibility:

```
What is your first and last name?
[Unknown]:lab-rg06-01.cms.waikato.ac.nz
What is the name of your organizational unit?
[Unknown]:
What is the name of your organization?
[Unknown]: University of Waikato
What is the name of your City or Locality?
[Unknown]: Hamilton
What is the name of your State or Province?
[Unknown]: Waikato
What is the two-letter country code for this unit?
[Unknown]: NZ
Is CN=lab-rg06-01.cms.waikato.ac.nz, OU=Unknown, O=University of
Waikato, L=Hamilton, ST=Waikato, C=NZ correct?
[no]: yes
```

At this point, you have created a public/private key pair for the server to use. Next, you have to create your server's public-key certificate that the client will trust. That is, a certificate signed by the CA's signing key (i.e. private key) we created in step one. The next step is to derive a certificate signing request (CSR) for our server certificate.

```
keytool -certreq -alias lab-rg06-01.cms.waikato.ac.nz -file server.csr \
    -keystore server.jks
```

Ordinarily, we would then contact an actual CA to sign the request. But, we're going to do roughly the same process that the CA does ourselves and using our CA created in Step 1.

```
openssl x509 -req -in server.csr -CA ca-cert.pem -CAkey ca-private.pem \
   -CAcreateserial -out server-cert.pem -days 90
```

You will be prompted for the passphrase for your `ca-private.key`. Enter whatever it was you wrote down in Step 1. You now have the signed server certificate saved to `server-cert.pem`. You then have to put the certificate in the server's KeyStore so that your server will supply the certificate to its clients. However, first, you have to store your CA's public-key certificate, `ca-cert.pem`, in the KeyStore so that it contains the complete chain.

```
keytool -import -trustcacerts -alias root -file ca-cert.pem \
   -keystore server.jks
keytool -import -alias lab-rg06-01.cms.waikato.ac.nz \
   -file server-cert.pem -keystore server.jks
```

You can take a look at the contents of your KeyStore with the following command:

```
keytool -list -keystore server.jks
```

You should see two entries: a `PrivateKeyEntry` for the server and `trustedCertEntry` for the CA's certificate. Record the CA's certificate fingerprint (SHA-256) in the text file provided in the Overview section (**row E**).

Compare this fingerprint value with the one you recorded in Step 1 and make sure they have the same value. At this point, you have a set of files you can use to establish a secure communications channel.

- `ca-private.pem`: the private key that our CA uses to sign certificates.
- `ca-cert.jks`: a KeyStore containing the public cert for our CA that we will supply to our client application, so it can establish a chain of trust.
- `server.jks`: a KeyStore containing the public certificate for our server signed by our CA, the server's corresponding private key, and the CA's public-key certificate to complete the chain.

You can take a look at the certificate you created for your server with the following command:

```
openssl x509 -pubkey -in server-cert.pem -noout -text
```

You should see the server's public-key (BEGIN PUBLIC KEY), the information of the CA who issued the server's public-key certificate, the period of time for which the certificate will be valid, and a string that describes whose certificate this is (the hostname), the algorithm and parameters used to generate the server's public-key, and finally the digital signature of our fake CA.

6 Step 3: TLS handshake in the server

Using TLS in an application can be confusing, partly because the APIs can be complex. The Java Secure Socket Extension (JSSE) API is no different (see

reference guide here). TLS Sockets (Java calls them SSLSocket) are created using SSLServerSocketFactory. The "factory" has the keys and certificates loaded into it so that when a SSLServerSocket is created, it is ready for use.

To start with, create or use a **MyTLSFileServer** file like described here. Check the Java Secure Socket Extension (JSSE) code provided and use it to create a custom SSLServerSocketFactory that (i) loads the server's signed certificate and associated private key stored in `server.jks`, (ii) binds to a port supplied as a command-line argument and then (iii) listens for incoming connections.

Your server class needs to accept a socket. Use the `readLine()` method on a BufferedReader that you construct on the socket's InputStream to force the TLS handshake in `MyTLSFileServer`. If your server listens on port 50202, you can check to see that the TLS handshake is working with the following command:

```
openssl s_client -connect lab-rg06-01.cms.waikato.ac.nz:50202
```

The output will show the certificate your server supplied, the TLS protocol negotiated, the cipher being used, as well as other information including the output of certificate verification. If it shows the following:

```
CONNECTED(00000004)
write:errno=0
---
no peer certificate available
---
No client certificate CA names sent
---
SSL handshake has read 0 bytes and written 293 bytes
Verification: OK
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
```

This is because you have not connected a BufferedReader to the accepted SSLSocket and used `readLine()` to force the handshake.

Otherwise, it will say that it is a self-signed certificate, rather than it being signed by a trusted CA. You can fix this by running:

```
openssl s_client -CAfile ca-cert.pem -connect lab-rg06-01.cms.waikato.ac.nz:50202
```

Now you should see the line `Verify return code: 0 (ok)` as you have told `openssl` to trust your CA's certificate.

If you encoded the passphrase in the source code of `MyTLSFileServer`, you must now use java.io.Console.readPassword to obtain the passphrase securely from the keyboard.

7 Step 4: TLS handshake in the client

The TLS handshake is somewhat simpler in the client. Create a `MyTLSFileClient` file like here and create an SSL socket that connects to the server and does the handshake. Your program should take three arguments:

- the hostname of the system to connect to,
- the port it is listening on,
- and the file to retrieve.

To start with, do *not* perform hostname verification (i.e. you may have comment out code and not use the setEndpointIdentificationAlgorithm method). Now, run your code as follows:

```
java MyTLSFileClient localhost 50202 cat.png
```

Your code *must have thrown an exception* because the certificate path verification will fail. What is the exception that is thrown? Record the exception in the text file provided in the Overview section (**row F**).

Next, tell Java that you want it to trust your CA certificate (Step 1) by running your code as follows:

```
java -Djavax.net.ssl.trustStore=ca-cert.jks -Djavax.net.ssl.trustStorePassword=password
MyTLSFileClient localhost 50202 cat.png
```

Note: you need to replace *password* with the actual password you entered for `ca-cert.jks` in Step 1.

The handshake should complete with no exceptions thrown. Next, use setEndpointIdentificationAlgorithm("https") to tell Java to perform hostname verification. Run your code again as above, with `localhost` as the name of the machine you are connecting to. It should throw an exception. What is the exception that is thrown? Record the exception in the text file provided in the Overview section (**row G**). Now, run your code specifying the name of the machine you are on. For example:

```
java -Djavax.net.ssl.trustStore=ca-cert.jks -Djavax.net.ssl.trustStorePassword=password
MyTLSFileClient lab-rg06-01.cms.waikato.ac.nz 50202 cat.png
```

The code should run *without* throwing an exception.

Next, get inspired by the client code fragment provided already (here) to obtain the X509Certificate sent by the server and have a look at the certificate.

8 Step 5: Transfer a file

In the client, send the name of the file requested to the server. In the server, take the name of the file and send the file contents back (i.e. read and write a binary file as you have already done in assignments 2 and 3). You can obtain the InputStream and OutputStream objects for the socket to help with this, and treat the SSLSocket as if it were a regular socket. If the file does not exist on the server, the server should just close the connection. Your client should write the data it receives to a file named underscore (_) followed by the original filename, this is to prevent overwriting the existing file. When the file transfer is complete, the server should close the socket and the client should exit. Compare the two files to ensure they are identical with the `shasum` command.

9 Grading

Your solution will be marked on the basis of how well it satisfies the specification, how well-formatted and easy to read your code is, and whether each block of code has at least some comment explaining what it does and why. The grading rubric is as follows:

- 2 marks: Certificates and keys created according to specification (steps 1, 2)
- 3 marks: File transfer according to specification (steps 3, 4, 5)

Your code should compile and run as a console program from the Linux command-line (i.e. no GUI). Students are encouraged to test their code on the Linux machines in R Block and demonstrate them to a teaching staff prior to submitting their solutions.

Make sure your name and student ID number are included in the header documentation of each source code file. You may include a `readme.txt` file with any notes you wish to the marker to read as they are marking your submission.

If you have the assignment verified in the lab prior to the due date, your feedback will be supplied verbally, and you will receive a grade after you have submitted your assignment.

You must submit the following files as part of your submission:

1. `ca-cert.pem`
2. `ca-private.pem`
3. `ca-cert.jks`
4. `server-cert.pem`

5. `server.jks`
6. `server.csr`
7. `MyTLSFileClient.java`
8. `MyTLSFileServer.java`
9. `notes.md`
10. `readme.txt` (optional)

Contents of notes.md:

```
## Step 1

A: Passphrase for \verb+ca-private.pem+:

B: Password for \verb+ca-cert.jks+:

C: The CA's certificate fingerprint (SHA256) in \verb+ca-cert.jks+:

## Step 2

D: Passphrase for \verb+server.jks+:

E: The CA's certificate fingerprint (SHA256) in \verb+server.jks+:

## Step 4

F: The exception from the first command:

G: The exception from the second command:
```