COMPX204-18B - Secure File Transfer with TLS

September 10, 2018

1 Introduction

This practical will introduce you to programming network applications with Java using Transport Layer Security (TLS). You will do so by writing a small secure file transfer system that allows a client to securely request a file from an authenticated server. Please take some time to look through Lectures 10 and 11 available on Moodle, for an overview of TLS. Lecture 10 and 11 explains the theory behind some of the algorithms used by TLS, and Lecture 12 contains code that you will find useful.

2 Cutting and Pasting

This assignment contains many commands that you might be tempted to cut and paste. If you do, be careful where the commands are printed on more than one line in the PDF file. These commands have the second line indented, to signify that the line is a continuation of the previous line.

3 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 don't understand, seek help until you do.

You **must** submit your files to Moodle in order to receive any marks recorded on your verification page.

This assignment is due **Monday September 24th by 11am** and is worth 6% of your final grade.

Step 1: Prepare a CA's Signing Key

Validating a certificate path 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 certificates 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 practical. Instead, you will create your own signing certificate used as a CA's certificate and tell your Java application that anything signed by it has a valid certificate path. To create a signing certificate:

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

This command will create a certificate (i.e. CA's certificate) we will use to sign certificates. It will be valid for about 10 years (3650 days). openssl will ask you to enter a pass phrase to protect the ca-private.pem file. You will need this pass phrase to decode the private key when you later sign your server certificate. You should enter a pass phrase here, and it should not be the same as any other password you use. To help you remember the pass phrase (and because the pass phrase is a toy passphrase that you will only use for this assignment) write it down here:

```
ca-private.pem pass phrase:
```

You will also be prompted for other fields to be incorporated into the certificate. For Country Name, enter "NZ", for State Name enter "Waikato", for Locality "Hamilton", for Organization Name enter your name, for Unit Name enter nothing, for Common Name enter your name again, for Email Address enter an email address for yourself. Here is what I did:

```
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]:Shaoqun Wu
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:Shaoqun Wu
Email Address []:shaoqun@waikato.ac.nz
```

At this point you will have a signing private key, and a certificate you can distribute to enable validation of your server key. However, to use the certificate with Java, we have to put it into a file format (JKS, i.e. Java keystore) that Java will use. Java provides a keytool utility 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 it here:

```
ca-cert.jks password:
```

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

Step 2: Prepare Server Key Pair

In the next step, you will generate a server key-pair and sign it with your signing certificate, so that clients will have a trusted certificate path. Your server will send the signed certificate during the TLS handshake. Your server key needs to correspond to the name of the machine on which you run your server. For example, if you are sitting at the machine cms-r1-17.cms.waikato.ac.nz then your certificate needs to have that name in it. We will use Java's keytool to generate the keypair. The keypair is stored in a file containing both the public key and the private key.

```
keytool -genkeypair -alias cms-r1-17.cms.waikato.ac.nz -keyalg RSA
        -keystore server.jks
```

You will be prompted for a password to protect the keystore – in particular the private key in that file. Choose a different password than above.

```
server.jks password:
```

You will also be prompted for "your first and last name". This is actually prompting for the certificate's "common name" – or whatever it is that we're 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 what I entered:

```
What is your first and last name?
  [Unknown]:  cms-r1-17.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=cms-r1-17.cms.waikato.ac.nz, OU=Unknown, O=University of Waikato,
              L=Hamilton, ST=Waikato, C=NZ correct?
 [no]: yes
Enter key password for <cms-r1-17.cms.waikato.ac.nz>
        (RETURN if same as keystore password):
```

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

```
keytool -certreq -alias cms-r1-17.cms.waikato.ac.nz -file server.csr
```

```
        -keystore server.jks
```

Ordinarily, we would then contact an actual CA, but we're going to do roughly the same process as the CA does ourselves and fake it.

```
openssl x509 -req -in server.csr -CA ca-cert.pem -CAkey ca-private.pem

        -CAcreateserial -out server-cert.pem
```

You will be prompted for the passphrase for your ca-private.key. Enter whatever it was you wrote down for step one. You will then have the signed server certificate in server-cert.pem. You then have to put the certificate in the server's keystore so that your server will supply the certificate to clients, and you also have to store the ca-cert used to sign the key, to complete the chain.

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

        -keystore server.jks

keytool -import -alias cms-r1-17.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 don't have to enter a password, because it can summarise the keystore without divulging the private key. You should see the root certificate you created, and a private key entry for the server. At this point you have a set of files you can use to establish a secure communications channel.

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

In practice, you normally would not have to create and use `ca-private.pem` or `ca-cert.jks`, but we need it so the client can establish the certificate path in the absence of a certificate signed by an actual CA.

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

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

You should see the public key, a digital signature, a string that describes who signed the certificate, the period of time for which your certificate will be valid, and a string that describes whose certificate this is (the hostname).

Step 3: TLS handshake in the server

Using TLS in an application is complicated, partly because the APIs can be complex. The Java Secure Socket Extension (JSSE) API is no different. TLS Sockets (Java calls them SSLSocket) are created using a SSLServerSocketFactory. An SSLServerSocket- Factory has the keys and certificates loaded into it so that when an SSLServerSocket is created it is ready for use.

To start with, create a MyTLSFileServer class and use the code from Lecture 12 to create a custom SSLServerSocketFactory that loads the signed certificate and associated private key (server.jks), binds to a port supplied as a command line argument, and then listens for incoming connections. Your server class needs to accept a socket, and then use the readLine method on a BufferedReader that you construct on the socket's InputStream. The readLine is necessary to force the TLS handshake in MyTLSFileServer.

If your server listens on port 40202, you can check to see that the TLS handshake is working with the following command:

```
openssl s_client -connect localhost:40202
```

the output will show the Certificate presented, the TLS protocol negotiated, the Cipher used, as well as other information including the output of path verification. It will say there is a self-signed certificate, rather than something signed with a trusted CA. If you use:

```
openssl s_client -CAfile ca-cert.pem -connect localhost:40202
```

the output will say server verification was ok. Note that the s client is not performing hostname verification; if it was, it would complain because the hostname used (localhost) is not the same as the hostname you set in the certificate signing request in step two.

If you encoded the passphrase in the source code of MyTLSFileServer, please consider reading about java.io.Console.readPassword and using that method to obtain the passphrase securely from the keyboard. This is not required for your assignment but is required in real life.

Step 4: TLS handshake in the client

The TLS handshake is somewhat simpler in the client. Create a MyTLSFileClient class and use the code from Lecture 12 to 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. Next, run the code as follows:

```
java MyTLSFileClient localhost 40202 cat.png
```

Your code should throw an exception because certificate path verification will fail. What is the exception that is thrown?

```
Record the exception here:
```

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

```
java -Djavax.net.ssl.trustStore=ca-cert.jks

     MyTLSFileClient localhost 40202 cat.png
```

The handshake should complete with no exceptions thrown. Next, use the setEndpointIdentificationAlgorithm code from Lecture 12 to tell Java to perform hostname verification. Run your code as above, with localhost as the name of the machine you are connecting to. It should throw an exception again. What is the exception that is thrown?

```
Record the exception here:
```

Now, run your code specifying the name of the machine you are on. I ran my code as follows:

```
java -Djavax.net.ssl.trustStore=ca-cert.jks

     MyTLSFileClient cms-r1-17.cms.waikato.ac.nz 40202 cat.png
```

The code should run without throwing an exception. Next, leave the setEndpoint IdentificationAlgorithm code in place, but also use the code from Lecture 12 that obtains the X509Certificate sent by the server, extracts the CommonName, and print it out to the terminal. It should match the hostname you specified when you ran the command above. Do a string comparison to convince yourself you could do hostname verification yourself if you had to.

Transfer a file

In the client, send the name of the file specified to the server. In the server, take the name of the file and send the file back. 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. When the client records the file from the server, it must prepend the output filename with an underscore character (_) to prevent the source file being over-written.

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 **`sha512sum`** command.

VERIFICATION PAGE

Name: _____

Id: _____

Date: _____

Note: this practical is due **Monday 24 of September**, and the code submitted to the Moodle assignment page after you have had the assignment verified. It will be marked out of 6 and is worth up to 6% of your final grade.

1. Show the demo the output of `openssl x509 -in server-cert.pem -noout -tex`
2. Show the demo the output of `openssl s_client -CAfile ca-cert.pem -connect localhost:40202`
3. What exception is thrown when MyTLSFileClient is not supplied with your CA certificate?

4. What exception is thrown when you tell MyTLSFileClient to validate the hostname, but you use the wrong hostname?

5. Demonstrate that your MyTLSFileClient can retrieve a binary file correctly by using the sha512sum command to show there were no errors introduced in copying the file.
6. Explain to the tutor or demo the structure of your MyTLSFileServer and MyTLSFileClient programs.