COMP3310/6331 – Tute #10

Outline of Tute/Lab:

This lab will introduce you to the *Secure Sockets Layer (SSL)* protocol. As you should have seen by now from the last lecture, essentially SSL introduces a level of encryption for packets to mitigate against bad things being done to network traffic. You will also see reference in places to *TLS (Transport Layer Security)* which is effectively a newer, and more complex, version of SSL (post SSLv3).

There are a few useful concepts to go through.

- If you're not familiar with cryptography, and in particular public key cryptography, think of cryptography as encoding your messages (with a special key that you and your recipient both have) to avoid snoopers seeing what you're sending. In public-key cryptography, there are **two** keys (one private, one public, the latter you can share with everybody, but only the private key can decrypt things encrypted with the public key, and vice-versa).
- A 'certificate' is effectively a document/file that somebody else formally and legally validates that a particular public key belongs to a named individual. We say that ***"a certificate binds an identity to a public key"***. It usually requires some 3rd party company who sells you a certificate, that they digitally sign. They act as a 'certificate authority' for your certificate. In turn, they themselves need to have their certificate signed by a higher authority, and so on, and we end up with a chain of certificates (up to a small number of globally trusted Certificate Authorities or CAs)

This lab has four stages:

1. Download and build the openssl software
2. Build a trust store
3. Observe SSL Handshake with the openssl client (s_client) and wireshark
4. Program with SSL - Java and C

You will experiment with the openssl package to connect to a HTTPS (http over ssl) site and observe the output. You will also download C and Java SSL programs to compile and run them. For some additional background it might be helpful to watch some 'SSL Handshaking' – e.g. the video at https://www.youtube.com/watch?v=sEkw8ZcxtFk or https://sites.google.com/site/ddmwsst/create-your-own-certificate-and-ca/ssl-socket- communication#TOC-SSL-with-Client-Authentication

Preparation:

This lab relies on a unix(like) environment and command-line access. Students with access to a unix/linux machine or Mac should have no problem. For those using Windows 10, if you haven't already you can install the *"Windows Subsystem for Linux"* using the instructions at https://docs.microsoft.com/en-us/windows/wsl/install-win10 and then select a Linux distribution - I recommend Ubuntu. You can then fire up an Ubuntu instance by clicking on the icon/menu item and get your traditional unix bash shell.

Please note: The directories assume a standard lab environment, so you may need to adjust some paths, in particular /students/user is changed to wherever your home directory is in your environment. Using '~' in place may suffice.

Stage 1: Download and build the openssl software

Create an openssl directory for installation

```
user@host$ mkdir ~/openssl
user@host$ cd ~/openssl
```

Download and unpack the latest stable version

```
user@host$ wget http://www.openssl.org/source/openssl-1.0.2r.tar.gz
user@host$ tar -xvzf openssl-1.0.2r.tar.gz
```

Build openssl

NOTE: Beware of the line wraps here, the '- -' is directly before the arguments.

```
user@host$ pwd
/students/user/openssl
user@host$ cd openssl-1.0.2r/
user@host$ ./config --prefix=/students/user/openssl \
--openssldir=/students/user/openssl enable-ec_nistp_64_gcc_128

user@host$ make depend --jobs=8
user@host$ make --jobs=8
user@host$ make install
```

Note you have installed the openssl package in your local (not system) directory, /students/user/openssl.

Check the openssl version

```
user@host$ ~/openssl/bin/openssl version
OpenSSL 1.0.2r 14 May 2019
```

Try "man openssl" and "openssl version -a" on a terminal session.

Stage 2: Build a trust store

A "trust store" contains a list of certificates for many well-known trusted Certificate Authorities (CAs), and is essential to the process of SSL handshaking. Your browser for example has many certificates it comes with (trusted companies) and uses them for secure HTTP connections. (For more information on how a root certificate is involved in SSL, you can watch the segment ~1:45-3:40 at https://www.youtube.com/watch?v=lLw0dICMA_Y)

Unfortunately this is a little complex, but it's useful to be familiar with it. To build a trust store, first you will download the Mozilla-maintained certificate and a Perl script, and convert the certificate into a PEM format used by openssl.

Download the certificates

```
user@host$ wget https://hg.mozilla.org/mozilla-central/raw-
file/tip/security/nss/lib/ckfw/builtins/certdata.txt \
--output-document certdata.txt
```

Download the Perl script, then generate the .crt file.

```
user@host$ wget
https://raw.githubusercontent.com/curl/curl/master/lib/mk-ca-
bundle.pl

user@host$ chmod +x mk-ca-bundle.pl
user@host$ ./mk-ca-bundle.pl
```

The script has now generated a 'ca-bundle.crt' file. Copy it to your /students/user/openssl/certs directory `user@host$ cp ca-bundle.crt ../openssl/certs`

You can view the ca-bundle.crt file:
`user@host$ view ca-bundle.crt`

NOTE: This is your own trust store, not a system-wide one. On Ubuntu 16.04, the default location of a trust store is /etc/ssl/certs and this folder is a part of Ubuntu 16.04 installation – you may want to have a look.

Stage 3: SSL Handshake protocol

Now you will use the ca-bundle.crt file when you connect to a HTTP over SSL (https://) site with the openssh "s_client". First, change your PATH environment variable:

```
user@host$ export PATH=/students/user/openssl/bin:$PATH
```

On Ubuntu 16.04, the openssl "s_client" does not use the default trust store (/etc/ssl/certs), but an ssl connection needs a certificate, so it makes one up – that has not been 'signed' by any trusted CA. We end up with an error that says there is a "self-signed" certificate in the certificate chain when we connect to a webserver somewhere (you are asserting yourself that you are trustworthy…!). Note here that 443 is the standard port number for https, like 80 for http, and the server is sending you its certificate (you have to trust it, like it trusts you!):

```
user@host$ openssl s_client help
user@host$ openssl s_client -connect www.hp.com:443
CONNECTED […]
verify error:num=19:self signed certificate in certificate chain
[… and then lots of interesting info about the server and handshake …]
Verify return code: 19 (self signed certificate in certificate chain)
```

Clearly not happy. Now we'll use the openssl s_client with the -CAfile option to point to our trust store, where we keep properly signed certificates. In this case it shouldn't complain about the self signed certificate in certificate chain now, it verifies each of the certificates from the chain.

```
user@host$ openssl s_client -connect www.hp.com:443 -CAfile
/students/user/openssl/certs/ca-bundle.crt
CONNECTED […]
depth=2 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert Global Root CA verify
return:1
depth=1 C = US, O = DigiCert Inc, CN = DigiCert SHA2 Secure Server CA verify return:1
depth=0 C = US, ST = California, L = Palo Alto, O = HP Inc, OU = AIS, CN = hp.com verify return:1
[… server and handshake stuff…]
Verify return code: 0 (ok)
```

Much better. Now try the command below, with a different target and different trust store (the built-in one), and enter GET / HTTP/1.0\n\n and observe the output:

```
user@host$ openssl s_client -connect www.anu.edu.au:443 \
--CAfile /etc/ssl/certs/ca-certificates.crt
```

Run wireshark and observe the SSL handshake. Unpack the packets and compare the structure with a normal telnet http GET request.

Stage 4: SSL programming

We'll start with Java:

Download an example Java SSLSocketClient file

```
user@host$ mkdir ~/java-ssl; cd ~/java-ssl
user@host$ wget
https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse
/samples/sockets/client/ SSLSocketClient.java
```

Change 'www.verisign.com' to 'www.google.com', compile and run it. Check the output:

```
user@host$ javac SSLSocketClient.java
user@host$ java SSLSocketClient
```
HTTP/1.0 200 OK
Date: Sun, 13 May 2018 06:08:29 GMT
[…more http headers…]

Compare the line 79 in the source file with the corresponding lines in the C version below. Note that the Java client sends the HTTP Request (line 86) after SSL Handshaking.

```
socket.startHandshake();
out.println("GET / HTTP/1.0");
```

Now we'll add some debugging and check the output. Notice that Java has its own trust store at */usr/lib/jvm/java-8-openjdk-amd64/jre/lib/security/cacerts*. You should see the exchange of ClientHello, ServerHello, ServerHelloDone, and so on.

```
user@host$ java -Djavax.net.debug=all SSLSocketClient
```

Now the SSL Socket Client in C.

- `user@host$ mkdir ~/c-ssl; cd ~/c-ssl`
- In a browser go to http://fm4dd.com/openssl/sslconnect.htm and download the sslconnect.c file to the above folder.
- Change www.hp.com to www.google.com.
- If you get a compile error below of "openssl/bio.h: No such file or directory…" you'll need to install the ssl libraries:
  `user@host$ sudo apt-get install libssl-dev`

- If you get a compile error below of "warning: sslconnect.c: implicit declaration of function 'close' then add '#include <unistd.h>' at the top
- Compile:
  `user@host$ gcc -o sslconnect sslconnect.c -lssl -lcrypto`

Run it and observe the output, hopefully it looks somewhat like the below
`user@host$ ./sslconnect`
Successfully made the TCP connection to: https://www.google.com.
Successfully enabled SSL/TLS session to: https://www.google.com.
Retrieved the server's certificate from: https://www.google.com.
Displaying the certificate subject data:
C=US, ST=California, L=Mountain View, O=Google LLC, CN=www.google.com
Finished SSL/TLS connection with server: https://www.google.com.

For the final exercise, go to the site below and modify/implement the 'Simple SSL Server' and the 'Simple SSL Client' – perhaps work in pairs. Test the Server and the Client against each other.

https://sites.google.com/site/ddmwsst/create-your-own-certificate-and-ca/ssl-socket-communication#TOC-SSL-with-Client-Authentication

Finished!

Questions from lectures:
This tute is probably long enough already, but some questions for review:

1. In the 7 layer model, what are some security 'opportunities' at each layer?

   Use your imagination - Physical = tapping, Link = break in and listen/interfere with wifi or ethernet say, Network = break in and listen/interfere on routers/gateways/modems/firewalls, Transport allows you to see/modify application communications, Applications = where we ultimately want to break into for access to content or to interfere. All layers support some kind of passive or active attacks

2. The idea behind encryption is mainly seen as providing confidentiality. Why do we have both SSL/TLS and IPSec, if they do pretty much the same thing?

   They run at different layers (TLS = Transport, IPSec = Network) and have different features. TLS supports application confidentiality, but still reveals src/dest/port information. IPSec hides everything about the original packet, and provides tunnels that have endpoints different to the actual src/dest of the application. IPSec also covers every application, while TLS is application-by-application (they each need to decide to do it).

3. What does it mean to have 'confidentiality', 'authenticity' and 'integrity' of packets traversing the network? What about 'freshness'?

   Standard points: confidentiality = nobody can listen, authenticity (or non-repudiation) = trust the other endpoint is who they claim to be, integrity = packets have not been tampered with, freshness = packets are new and not a replay.

4. Why does TOR (onion routing) use 3 routers for passing traffic through?

   To ensure that none of the three get to see both the source and destination addresses, and so no device can identify the endpoints of a conversation.

5. Why would a worker on the road use a 'mixed-mode' VPN connection to their office? What's the benefit, compared to say a 'transport-mode' VPN?

   A mixed-mode VPN is basically a host at one end and a router (or equivalent) at the other (office) end. This means the remote worker will appear like any other device on the office network. A transport-mode VPN directly ties to a host at both ends (remote and office), which is fine if those are the only devices needing to communicate. But the packets from the remote host won't be forwarded to anything on the office network.

6. How does ingress filtering prevent (most) spoofing?

   A router providing a connection to the wider internet, say, can check that incoming packets on a particular interface only come from addresses that it would formally send to (forward) on that interface. If it sees packets with other source addresses coming in from the 'wrong' side then it can drop them as being spoofed (fake) source addresses.