Tutorial 8 (TLS - Transport Layer Security)

In this tutorial, we will program a primitive TLS-based application. First, we will run a TLS server using OpenSSL and then write a Python client to interact with it.

1) Self-signed Digital Certificates

To run a TLS server, we will need a certificate. Recall that digital certificates are usually signed by trusted third parties such as Certificate Authorities (CAs). This allows us to verify whether the public key contained in the certificate belongs to the correct party. However, for experimentation, we cannot use these certificates since they must be purchased from a CA. Therefore, in development and lab settings, we use self-signed certificates.

A self-signed certificate is a digital certificate issued by the person or entity creating the certificate rather than by a trusted third-party certificate authority. This means the certificate is not backed by a well-known and trusted third party, so it may not be considered as secure as a certificate issued by a trusted authority.

Simply put, a self-signed certificate is neither privately nor publicly certified by a CA. Instead of seeking the certificate from a public or private CA, it is signed using its own private key; i.e., the signature in the certificate can be validated by the public key contained within the certificate itself.

We have already created a certificate for you—*certificate.crt*—which is available in the tutorial folder.

Use the `openssl` command you learned last week to inspect the certificate's contents.

- What is the public key algorithm used in the certificate?
- What is the email address associated with the certificate?
- What is the Common Name (CN)? We will need it later.

2) A Quick and Dirty TLS Server for Debugging Purposes

The `openssl s_server` command allows you to run a TLS server on a specified port. This server will perform TLS handshakes as required and display anything the client sends. However, you need to pass a few arguments to make it work.

- Open a new SSH connection to the VM.
- Run the following OpenSSL command to start the server:
  - **sudo openssl s_server -accept 4443 -cert certificate.crt -key private.key**
- You will be prompted to enter the access code for the private key. Enter `usyd`.

- If everything is running correctly, you will see the message `"Using default temp DH parameters ACCEPT"` in the command line.
- If you get an error like `"Address already in use"`, try using a different port number.
- **Do not close this terminal window** while the server is running.

3) Complete the TLS Client Code

Let's begin writing our TLS client. You will find starter code in the cell below. Read it carefully, then refer to the API documentation as needed (https://docs.python.org/3/library/ssl.html) to complete the code.

- How do you create a socket and initiate a connection?
- What is an SSL Context, and what parameters does the context require?
- How do we use the context with the connection?

In [ ]:
```
import ssl
import socket
import time

hostname =   # TODO - Set hostname to localhost
port =             # TODO - Set the port number you are listening on in O

# This function handles communication once the TLS connection is establis
# TODO: Read and understand what it does. Nothing to change here.
# Understand what the server needs to do to close the connection.

def handle(conn):
    KEEP_ALIVE = True
    while KEEP_ALIVE:
        print("PING...")
        conn.write(b'PING...')
        time.sleep(1)
        res = conn.recv().decode().strip()
        if res == "HELLO" and KEEP_ALIVE:
            print("Received HELLO, sending CLIENT ALIVE...")
            conn.write(b'CLIENT ALIVE...')
        elif res != "HELLO" and KEEP_ALIVE:
            KEEP_ALIVE = False
            print("No HELLO received, closing the connection")
            conn.write(b'CLIENT CLOSING...')
            conn.close()

# TODO - Refer to https://docs.python.org/3/library/ssl.html
# TODO - Complete the code below

def main():
    sock =   # TODO add the connection parameters
    context =   # TODO create SSL context
    context.load_verify_locations()  # TODO specify the path to the certi
    context.verify_flags  # TODO - Disable strict X.509 checks (self-sign
    conn =   # Wrap the socket with SSL
    try:
        handle(conn)
    except Exception as e:
```

```
        print(e)
    finally:
        conn.close()

main()
```

Run the code. In the server side you can type replies to the client in the command line.

4) Extension - Improving Security

Back to the client. Modify it such that:

- It verifies that the Organization and Organizational Unit in the server certificate have the correct values:
  - O (Organization) = The University of Sydney
  - OU (Organizational Unit) = The School of Computer Science

Why is it necessary for `localhost` to appear in the Common Name (CN) field?

In [ ]:
```
import ssl
import socket
import time

hostname = 'localhost' ### TODO - Set hostname to localhost
port = 4444 ### TODO - Set the port number you are listening in openssl

### This function handles the communication the TLS connection is establi
### TODO Read and understand what it does. Nothing to change here. Unders

def handle(conn):
    KEEP_ALIVE=1
    while KEEP_ALIVE==1:
        print("PING...")
        conn.write(b'PING...')
        time.sleep(1)
        res = conn.recv().decode().strip()
        if res == "HELLO" and KEEP_ALIVE==1:
            print("Received HELLO, sending PING")
            conn.write(b'CLIENT ALIVE...')
        elif res != "HELLO" and KEEP_ALIVE==1:
            KEEP_ALIVE=0
            print("No HELLO received, Closing the connection")
            conn.write(b'CLIENT CLOSING...')
            conn.close()

### TODO - Have a look here https://docs.python.org/3/library/ssl.html
### TODO - And complete the codes

def main():
    sock =   # TODO add the connection parameters
    context =   # TODO create SSL context
    context.load_verify_locations()  # TODO specify the path to the certi
    context.verify_flags  # TODO - Disable strict X.509 checks (self-sign
    conn =   # Wrap the socket with SSL
    cert =  ### TODO - Use the getpeercert() method to access the certifi
```

```
    subject = dict(x[0] for x in cert['subject'])
    organizationName =  ### TODO - Extract organizationName from subject
    organizationalUnitName =  ### TODO - Extract organizationalUnitName f

    print(organizationName)
    print(organizationalUnitName)

    if ### TODO Add the if condition to check O and OU
        try:
            handle(conn)
        except Exception as e:
            print(e)
        finally:
            conn.close()

main()
```

5) Capturing Packets

Next, we are going to use the packet capture tool `tshark` to capture the packets exchanged between your client and the TLS server running in the VM.

- Go to the `/tmp` directory and run the command: `sudo tshark -i lo -w ./capture.pcap`
- Here, `lo` refers to the loopback interface on which our server is listening.
- While `tshark` is running, execute your client code again, then press `Ctrl+C` to stop capturing.
- If everything is working correctly, you will see an output like: `Capturing on 'Loopback: lo'`
- Download the capture file to your local machine.
- Download and install **Wireshark** on your computer from www.wireshark.org.
- Open the `capture.pcap` file using Wireshark.
- Find the communication between your client and the OpenSSL server.
- **Hint:** A TLS communication starts with a `Client Hello` message. Do a regular expression search.
- Once you find the packet, right-click and use the **Follow TCP Stream** option to get the full message flow.
- Inspect the **payload** tab for the packet content.

Question

- What TLS version does the communication use?
- What is the selected Cipher Suite?

4) Homework – Try the Above with Your Own Self-Signed Certificate

Use the link below to learn how to create your own self-signed certificate in Ubuntu:

- https://learn.microsoft.com/en-us/azure/application-gateway/self-signed-certificates
- Verify that when you specify a different hostname in the certificate, the code above no longer works.