# Transport Layer Security (TLS) Lab

## 1 Overview

Nowadays more and more data transmissions are done through the Internet. However, when data are transmitted over such a public network unprotected, they can be read or even modified by others. Applications worrying about the security of their communication need to encrypt their data and detect tampering. Cryptographic solutions can be used to achieve this goal. There are many cryptographic algorithms, and even for the same algorithm, there are many parameters that can be used. To achieve interoperability, i.e., allowing different applications to communicate with one another, these applications need to follow a common standard. TLS, Transport Layer Security, is such a standard. Most web servers these days are using HTTPS, which is built on top of TLS.

The objective of this lab is to help students understand how the TLS works and how to use TLS in programming. The lab guides students to implement a pair of TLS client and server programs, based on which students will conduct a series of experiments, so they can understand the security principles underlying the TLS protocol. Students will also implement a simple HTTPS proxy program to understand the security impact if some trusted CAs are compromised. The lab covers the following topics:

- Public-Key Infrastructure (PKI)
- Transport Layer Security (TLS)
- TLS programming
- HTTPS proxy
- X.509 certificates with the Subject Alternative Name (SAN) extensions
- Man-In-The-Middle attacks

**Prerequisite.** This lab depends on the PKI lab. Students should do the PKI lab before working on this lab.

**Readings.** Detailed coverage of PKI and TLS can be found in the following:

- Chapters 24 and 25 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.

**Lab Environment.** This lab has been tested on our pre-built Ubuntu 16.04 and Ubuntu 20.04 VMs (the 20.04 VM will be officially released in Summer 2021).

## 2 Task 1: TLS Client

In this task, we will incrementally build a simple TLS client program. Through the process, students will understand the essential elements and security considerations in TLS programming.

### 2.1 Task 1.a: TLS handshake

Before a client and a server can communicate securely, several things need to be set up first, including what encryption algorithm and key will be used, what MAC algorithm will be used, what algorithm should be used for the key exchange, etc. These cryptographic parameters need to be agreed upon by the client and the server. That is the primary purpose of the TLS Handshake Protocol. In this task, we focus on the TLS handshake protocol. The following sample code initiates a TLS handshake with a TLS server (the name of the server needs to be specified as the first command line argument).

Listing 1: `handshake.py`

```
#!/usr/bin/python3

import socket, ssl, sys, pprint

hostname = sys.argv[1]
port = 443
cadir = '/etc/ssl/certs'

# Set up the TLS context
# context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT) # For Ubuntu 20.04 VM
context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)      # For Ubuntu 16.04 VM
context.load_verify_locations(capath=cadir)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True

# Create TCP connection
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((hostname, port))
input("After making TCP connection. Press any key to continue ...")

# Add the TLS
ssock = context.wrap_socket(sock, server_hostname=hostname,
                            do_handshake_on_connect=False)
ssock.do_handshake()   # Start the handshake
pprint.pprint(ssock.getpeercert())
input("After handshake. Press any key to continue ...")

# Close the TLS Connection
ssock.shutdown(socket.SHUT_RDWR)
ssock.close()
```

**Tasks.** Use the code above to communicate with a real HTTPS-based web server. Additional code may need to be added to complete the tasks. Students can find the manual for Python's SSL module online. Please report the following:

- What is the cipher used between the client and the server?

- Please print out the server certificate in the program.
- Explain the purpose of `/etc/ssl/certs`.
- Use Wireshark to capture the network traffics during the execution of the program, and explain your observation. In particular, explain which step triggers the TCP handshake, and which step triggers the TLS handshake. Explain the relationship between the TLS handshake and the TCP handshake.

## 2.2 Task 1.b: CA's Certificate

In the previous task, we use the certificates in the `/etc/ssl/certs` folder to verify server's certificates. In this task, we will create our own certificate folder, and place the corresponding certificates in the folder to do the verification.

Please create a folder called `certs`, and change the `cadir` line in the client program to the following. Run the client program and report your observation.

```
cadir = './certs'
```

To solve this problem, you need to place the corresponding CA's certificate into your `certs` folder. Please use your client program to find out what CA certificate is needed to verify the `www.example.com` server's certificate, and then copy the certificate from the `/etc/ssl/certs` to your own folder. Run your client program again. If you have done everything correctly, your client program should be able to talk to the server.

It should be noted that copying CA's certificate to the `"./cert"` folder is not enough. When TLS tries to verify a server certificate, it will generate a hash value from the issuer's identify information, use this hash value as part of the file name, and then use this name to find the issuer's certificate in the `"./cert"` folder. Therefore, we need to rename each CA's certificate using the hash value generated from its subject field, or we can make a symbolic link out of the hash value. In the following command, we use `openssl`to generate a hash value, which is then used to create a symbolic link.

```
$ openssl x509 -in someCA.crt -noout -subject_hash
4a6481c9

$ ln -s someCA.crt 4a6481c9.0
$ ls -l
total 4
lrwxrwxrwx 1 ... 4a6481c9.0 -> someCA.crt
-rw-r--r-- 1 ... someCA.crt
```

**Additional requirement:** Please conduct this task for two different web servers that use different CA certificates.

## 2.3 Task 1.c: Experiment with the hostname check

The objective of this task is to help students understand the importance of hostname checks at the client side. Please conduct the following steps using the client program.

- Step 1: Get the IP address of `www.example.com` using the `dig` command, such as the following:

```
$ dig www.example.com
...
;; ANSWER SECTION:
www.example.com.  403   IN A  93.184.216.34
```

- Step 2: Modify the `/etc/hosts` file, add the following entry at the end of the file (the IP address is what you get from the `dig` command).

```
93.184.216.34   www.example2020.com
```

- Step 3: Switch the following line in the client program between `True` and `False`, and then connect your client program to `www.example2020.com`. Describe and explain your observation.

```
context.check_hostname = False  # try both True and False
```

**The importance of hostname check:** Based on this experiment, please explain the importance of hostname check. If the client program does not perform the hostname check, what is the security consequence? Please explain.

## 2.4 Task 1.d: Sending and getting Data

In this task, we will send data to the server and get its response. Since we choose to use HTTPS servers, we need to send HTTP requests to the server; otherwise, the server will not understand our request. The following code example shows how to send HTTP requests and how to read the response.

```
# Send HTTP Request to Server
request = b"GET / HTTP/1.0\r\nHost: " + \
          hostname.encode('utf-8') + b"\r\n\r\n"
ssock.sendall(request)

# Read HTTP Response from Server
response = ssock.recv(2048)
while response:
  pprint.pprint(response.split(b"\r\n"))
  response = ssock.recv(2048)
```

**Tasks.** (1) Please add the data sending/receiving code to your client program, and report your observation. (2) Please modify the HTTP request, so you can fetch an image file of your choice from an HTTPS server (there is no need to display the image).

## 3 Task 2: TLS Server

Before working on this task, students need to create a certificate authority (CA), and use this CA's private key to create a server certificate for this task. How to do these is already covered in another SEED lab (the PKI lab), which is the prerequisite for this lab. In this task, we assume all the required certificates have already been created, including CA's public-key certificate and private key (`ca.crt` and `ca.key`), and the server's public-key certificate and private key (`server.crt` and `server.key`).

## 3.1 Task 2.a. Implement a simple TLS server

In this task, we will implement a simple TLS server. We will use the client program from Task 1 to test this server program. A sample server code is provided in the following.

Listing 2: `server.py`

```
#!/usr/bin/python3

import socket
import ssl

html = """
HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n
<!DOCTYPE html><html><body><h1>Hello, world!</h1></body></html>
"""

SERVER_CERT    = './certs/server.crt'
SERVER_PRIVATE = './certs/server.key'


# context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER) # For Ubuntu 20.04 VM
context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)      # For Ubuntu 16.04 VM
context.load_cert_chain(SERVER_CERT, SERVER_PRIVATE)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
sock.bind(('0.0.0.0', 4433))
sock.listen(5)

while True:
   newsock, fromaddr = sock.accept()
   ssock = context.wrap_socket(newsock, server_side=True)

   data = ssock.recv(1024)              # Read data over TLS
   ssock.sendall(html.encode('utf-8'))  # Send data over TLS

   ssock.shutdown(socket.SHUT_RDWR)     # Close the TLS connection
   ssock.close()
```

**Testing.** We will use the client program developed in Task 1 to test this server program. In Task 1, the client program loads the trusted certificates from the `/etc/ssl/certs` folder. In this task, the CA is created by us, and its certificate is not stored in that folder. We do not recommend students to add this CA to that folder, because that will affect the entire system. Students should store the CA's certificate in the `"./certs"` folder, and then follow the instruction in Task 1 to set up the folder. Please test your program using the `/etc/ssl/certs` folder and the `./certs` folder, respectively. Please describe your observation and explain why.

## 3.2 Task 2.b. Testing the server program using browsers

In this task, we will test our TLS server program using a browser, such as Firefox. First, point your browser to the server, report what you see from the browser and explain why. The server listens to port `4433`. If you change it to the standard HTTPS port `443`, you need to run the server program using the root privilege.

In order for browsers to communicate with your TLS server, the browser needs to verify your server's certificate. It has to use the certificate issuer CA's certificate to do the verification, but since this CA is created in our lab, the browser does not have it on its trusted certificate list. We need to manually add our CA's certificate to it. For the Firefox browser, click the following menu sequence:

```
Edit -> Preference -> Privacy & Security -> View Certificates.
```

You will see a list of certificates that are already accepted by Firefox. From here, we can "import" our own certificate. Please import `ca.crt`, and select the following option: "Trust this CA to identify web sites". You will see that our CA's certificate is now in Firefox's list of the accepted certificates.

Please demonstrate that your browser can successfully communicate with your TLS server, and can display the content returned by the server.

## 3.3 Task 2.c. Certificate with multiple names

Many websites have different URLs. For example, `www.example.com`, `www.example.org`, `example.com` all point to the same web server. Due to the hostname matching policy enforced by most TLS client programs, the common name in a certificate must match with the server's hostname, or TLS clients will refuse to communicate with the server.

To allow a certificate to have multiple names, the X.509 specification defines extensions to be attached to a certificate. This extension is called Subject Alternative Name (SAN). Using the SAN extension, it's possible to specify several hostnames in the `subjectAltName` field of a certificate.

To generate a certificate signing request with such a field, we need to use a configuration file, and put all the necessary information in this file. The following configuration file gives an example. It specifies the content for the subject field and add a `subjectAltName` field in the extension. The field specifies several alternative names, including a wildcard name `*.bank32.com`.

Listing 3: `server_openssl.cnf`

```
[ req ]
prompt             = no
distinguished_name = req_distinguished_name
req_extensions     = req_ext

[ req_distinguished_name ]
C  = US
ST = New York
L  = Syracuse
O  = XYZ LTD.
CN = www.bank32.com

[ req_ext ]
subjectAltName = @alt_names

[alt_names]
DNS.1   = www.bank32.com
DNS.2   = www.example.com
DNS.3   = *.bank32.com
```

We can use the following `"openssl req"` command to generate a pair of public/private keys and a certificate signing request:

```
openssl req -newkey rsa:2048 -config ./server_openssl.cnf -batch \
            -sha256  -days 3650 -keyout  server.key -out server.csr
```

When the CA signs a certificate, for the security reason, by default, it does not copy the extension field from the certificate signing request into the final certificate. In order to allow the copying, we need to change the `openssl`'s configuration file. By default, `openssl` uses the configuration file `openssl.cnf` from the `/usr/lib/ssl` directory. Inside this file, the `copy_extensions` option is disabled (commented out). We do not want to modify this system-wide configuration file. Let us copy it file to our own folder, and rename it as `myopenssl.cnf`. We then uncomment the following line from this file:

```
# Extension copying option: use with caution.
copy_extensions = copy
```

Now, we can use the following program to generate the certificate (`server.crt`) for the server from the certificate signing request (`server.csr`), and all the extension fields from the request will be copied to the final certificate.

```
openssl ca -md sha256 -days 3650 -config ./myopenssl.cnf -batch \
           -in server.csr -out server.crt \
           -cert ca.crt -keyfile ca.key
```

Students need to demonstrate that their server can support multiple hostnames, including any hostname in the `example.com` domain.

## 4 Task 3: A Simple HTTPS Proxy

TLS can protect against the Man-In-The-Middle attack, but only if the underlying public-key infrastructure is not compromised. In this task, we will demonstrate the Man-In-The-Middle attack against TLS servers if the PKI infrastructure is compromised, i.e., some trusted CA is compromised or the server's private key is stolen.

We will implement a simple HTTPS proxy called `mHTTPSproxy` (`m` stands for `mini`). The proxy program simply integrates the client and server programs from Task 1 and 2 together. How it works is illustrated in Figure 1.

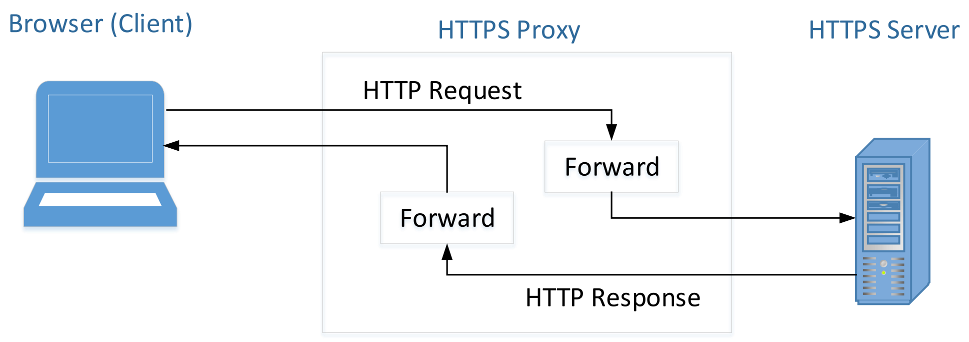

Figure 1: How `mHTTPSproxy` works

The proxy is actually a combination of the TLS client and server programs. To the browser, the TLS proxy is just a server program, which takes the HTTP requests from the browser (the client), and return HTTP responses to it. The proxy does not generate any HTTP responses; instead, it forwards the HTTP requests to the actual web server, and then get the HTTP responses from the web server. To the actual web server, the TLS proxy is just a client program. After getting the response, the proxy forwards the response to the browser, the real client. Therefore, by integrating the client and server programs implemented in the previous two tasks, students should be able to get a basic proxy working.

It should be noted that the purpose of this task is to use this simple proxy to understand how the Man-In-The-Middle attack works when the PKI infrastructure is compromised. It is not intended to implement a product-quality HTTPS proxy, because making the proxy work for every web server is not an easy job, as many aspects of the HTTP protocol need to be considered. Since the focus of this lab is on TLS, students can choose two different servers, and demonstrate that their proxy works for those servers. Students who are interested in product-quality HTTPS proxy, can find that from the Internet, such as the open-source `mitmproxy`.

**Handling multiple HTTP requests.** A browser may simultaneously send multiple HTTP requests to the server, so after receiving an HTTP request from the browser, it is better to spawn a thread to process that request, so the proxy program can handle multiple simultaneous requests. The following code snippet shows how to create a thread to handle each TLS connection.

```
import threading

while True:
    sock_for_browser, fromaddr = sock_listen.accept()
    ssock_for_browser = context_srv.wrap_socket(sock_for_browser,
                                                server_side=True)
    x = threading.Thread(target=process_request, args=(ssock_for_browser,))
    x.start()
```

The thread will execute the code in the `process_request` function, which forwards the HTTP request from the browser to the server, and then forward the HTTP response from the server to the browser. A code skeleton is provided in the following:

```
def process_request(ssock_for_browser):
    hostname = 'www.example.com'

    # Make a connection to the real server
    sock_for_server  = socket.create_connection((hostname, 443))
    ssock_for_server = ... # [Code omitted]: Wrap the socket using TLS

    request = ssock_for_browser.recv(2048)

    if request:
        # Forward request to server
        ssock_for_server.sendall(request)

        # Get response from server, and forward it to browser
        response = ssock_for_server.recv(2048)
        while response:
            ssock_for_browser.sendall(response) # Forward to browser
            response = ssock_for_server.recv(2048)
    ssock_for_browser.shutdown(socket.SHUT_RDWR)
    ssock_for_browser.close()
```

**Task.** Students should implement the simple `mHTTPSproxy`. To demonstrate it, pick a real HTTPS website as your targeted server, and then launch the Man-In-The-Middle attack against the server. Your victim is a user inside another virtual machine. Find a web server that requires login, and then use your MITM proxy to steal the password. Many popular servers, such as facebook, have complicated login mechanisms, so feel free to find a server that has simple login mechanisms. Please remember to hide your real password in your lab report.

The assumption of this MITM attack is that the attacker has compromised a trusted CA, and is able to generate fake (but valid) certificate using this CA's private key, for any hostname. In this lab, we assume that the CA you used to sign your server's certificate is compromised, and you can use it to forge certificate for any web server.

**Setup (Option 1: using two VMs).** You may want to run the victim browser on one VM (victim VM), and run your proxy on another VM (attacker VM). In the real-world attack, when the victim tries to visit a web server (say `www.example.com`), we will launch attacks to redirect the victim to our proxy. This is usually done by DNS attacks, BGP attacks, or other redirection attacks. We will not actually do such attacks. We simply modify the `/etc/hosts` file, and add the following entry to the victim VM (`10.0.2.12` is the IP address of the attacker VM).

```
10.0.2.12   www.example.com
```

By doing the above, we simulates the result of redirection attacks: the victim's traffic to `www.example.com` will be redirected to the attacker's VM, where your `mHTTPSproxy` is waiting for HTTP requests. It should be noted that on the attacker VM, the hostname `www.example.com` should point to the real web server.

**Setup (Option 2: using one VM).** If you want to use one VM for this lab, i.e., using one VM for both victim and attacker, this is doable. To do that, add the following entry to `/etc/hosts` to simulate the DNS attack.

```
127.0.0.1   www.example.com
```

Since the hostname `www.example.com` is already mapped to the localhost, we cannot use this name in the proxy code to connect to the real web server. We will get the IP address of this domain, and directly use the IP address in our proxy code, instead of using the hostname. See the following modification.

```
def process_request(ssock_for_browser):
    hostname = 'www.example.com'
    real_ip_address = "93.184.216.34"

    # Make a connection to the real server
    sock_for_server  = socket.create_connection((real_ip_address, 443))
    ...
```

**Cleanup.** After finishing this task, please remember to remove the CA's certificate from your browser, and also remove any entry that you have added to `/etc/hosts` for this lab.

## 5 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.