Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

# Network Security

## Exercise Sheet 04 Transport layer security

Prof. Dr. Mathias Fischer, August See, Finn Sell

Summer semester 2024

**Goals and Objectives** For programming tasks, submit the code as an extra file. Please write for each (sub) task how much time you need. You should reach at least 50%. This exercise is rather short, thus try to reach 100%.

## 1 Preliminaries 0%

Note that you can use any Linux system. It should also work under macOS as well as Windows with WSL. You likely do not need a VM.

### 1.1 Prepare your Linux VM

[1] Before creating the virtual machine, install a virtualization tool of your choice to run the VM. We recommend VirtualBox[1]. Another popular choice is VMWare Workstation Player[2].
We provide you two variants to get your personal Ubuntu 20.04 VM running.

**Variant 1: Fully installed VM:** The easiest way for you to get started with a working Ubuntu VM is to download and import a VM that we already prepared for you. The get this VM running on your machine, follow these instructions:

1. Download the fully installed VM (2.5GB file size) from our servers:
   `https://svs.informatik.uni-hamburg.de/teaching/rn/exercise/iss-rn.ova`
2. Assuming you use VirtualBox, select `File` -> `Import Appliance` and follow the import guide. This might take several minutes.

[1] Taken from Resilient Networks resources in Winter 2021/22.
[1] `https://www.virtualbox.org/wiki/Downloads`
[2] `https://www.vmware.com/de/products/workstation-player/workstation-player-evaluation.html`

**Variant 2: Autoinstall** If you do not want to use the fully installed VM provided by us, you can follow the same installation procedure we used. In this case, please refer to the brief instructions on our website at `https://svs.informatik.uni-hamburg.de/teaching/rn/exercise/iss-rn-vm/`. In summary, this requires you to:

1. Download the official live server installation image of Ubuntu 20.04 (914MB file size) from the ubuntu.com website: `https://releases.ubuntu.com/20.04/`
2. Download the given `seed.iso` from `https://svs.informatik.uni-hamburg.de/teaching/rn/exercise/iss-rn-vm/seed.iso`.
3. Download and import the empty VM `iss-rn-empty.ova` downloaded from `https://svs.informatik.uni-hamburg.de/teaching/rn/exercise/iss-rn-vm/iss-rn-empty.ova`. Go to the VM settings and choose the two Virtual Optical Disk Files according to Figure 1.
4. Start the VM with both the Ubuntu installation image and the `seed.iso` loaded. If done correctly, the interactive Ubuntu installer is skipped and the predefined options in `seed.iso` are used instead. Once these options have been loaded, confirm to start the installation.

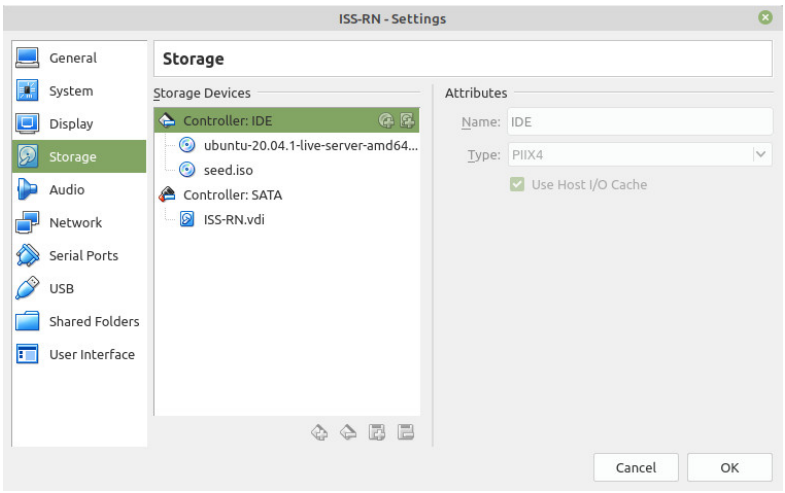


Figure 1: Order of loaded images in VirtualBox to run the automated installer.

## 1.2 Run the VM

Afterwards, you find **ISS-RN** in the list of your VMs. Select and start this VM. The login credentials are:

Username: `stud`

Password: `stud`

For better VM experiences, optionally install the integration tools of your virtualization tool. In case of VirtualBox, first make sure to install the following packages using the package manager `APT`. Open a terminal and type:

```
sudo apt-get update
```

Follow the instructions on the screen. Afterwards, focus the window of the running VM and select `Devices -> Insert Guest Additions CD image`. Again, follow the instruction on the screen.

# 2 TLS Basics 45%

## 2.1 TLS Client

In this task, we will incrementally build a simple TLS client program. As a first step TLS handshake is necessary. With that several things such as what encryption algorithm and key will be used, what MAC algorithm will be used, what algorithm should be used for the key exchange, etc. will be agreed upon by the client and the server. In this task, we focus on the TLS handshake protocol.

To begin you need to create a new socket, create an SSLContext with the appropriate configuration, and then use this to wrap the socket into an SSLSocket. You may want to check the socket and SSL documentation of python. [2] [3]

- To create a new socket, use the *socket.socket*() function. Make sure it's of the $AF\_INET$ type. (#1 in code)
- Obtain the default ssl context via the $ssl.SSLContext()$ function and set its purpose accordingly to authenticate servers. (#2 in code)
- Continuing with the context, load the CA certificate via the $context.load\_verify\_locations()$ function. This ensures that the client knows which root certificate to validate against. (#3 in code)
- Now we need a connection. Wrap the original socket into an SSLSocket using *context.wrap_socket*(). Note: You'll need to set the *server_hostname* parameter which needs to be specified as the first command-line argument. (#4 in code)

Once the client code is done, use the code above to communicate with a real HTTPS-based web server (e.g., www.example.com). Then answer following questions;

1. What is the cipher used between the client and the server?

[2] `https://docs.python.org/3.6/library/socket.html`
[3] `https://docs.python.org/3.6/library/ssl.html`

2. Print out the server certificate in the program.

3. Explain the purpose of /etc/ssl/certs.

In the given code template, the certificates in the */etc/ssl/certs* folder were used automatically to verify the server's certificate. Now, we will create our own certificate folder, and copy those certificates into a folder of our own to do the verification.
For that create a folder called certs, and change the cadir line in the client code as *cadir =′ ./certs′*. Then, you need to copy the appropriate CA's certificate into your certs folder. 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.

Copying the CA's certificate to your *./certs* 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 *./certs* folder. Therefore, we need to make a symbolic link to the certificate 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. By convention, a '.0' is appended to the hash value.

```
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
```

Run your client program again. If you have done everything correctly, your client program should be able to talk to the server. Name the certificate that you copy into *′./certs′* folder and print out the certificate.

## 2.2 Certificate Authority

In this task, we need to create digital certificates, but unlike the commercial CAs, we will become a root CA ourselves, and then use this CA to issue certificate for others (e.g., servers). To do this, we will generate our own certificate for this CA. Unlike other certificates, which are usually signed by a more trusted CA, our root CA's certificates are self-signed. Such CA's are not usually trusted, but for this lab it will be fine.

**Openssl configuration** In order to use OpenSSL to create certificates, you have to have a configuration file. You can get a copy of the configuration file from */usr/lib/ssl/openssl.cnf*. After copying this file into your current directory, you need to create several subdirectories as specified in the configuration file.
You will see an entry *certs.database = dir/index.txt* which is a database index file. For that, simply create an empty file. For the serial file which holds the current serial

number, put a single number in string format (e.g. 1000) in the file. Once you have set up the configuration file openssl.cnf, you can create and issue certificates.

**Certificate Authority** Now, we can generate a self-signed certificate for our CA. This means that this CA is totally trusted, and its certificate will serve as the root certificate. You can run the following command to generate the self-signed certificate for the CA:

```
$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf
```

You will be prompted for information and a password. Do not lose this password, because you will have to type the passphrase each time you want to use this CA to sign certificates for others. You will also be asked to fill in some information, such as the Country Name, Common Name, etc. The command results in the creation of two files: ca.key and ca.crt. The file ca.key contains the CA's private key, while ca.crt contains the public-key certificate.
Print out them by using *openssl x509* and *openssl rsa* commands with a proper optional fields. [1] [2]

**Creating a Certificate for example.com** Now we have a root CA and are ready to sign digital certificates for our customers. Our first customer is a company called example.com. For this, company to get a digital certificate from a CA, it needs to go through following steps.

- The company needs to first create its own public/private key pair. For that, run the following command to generate an RSA 2048-bit key pair. You will also be required to provide a password to encrypt the private key as is specified in the command option).

  ```
  $ openssl genrsa -aes128 -out server.key 2048
  ```

  The keys will be stored in the file server.key which is an encoded text file (also encrypted), so you will not be able to see the actual content, such as the modulus, private exponents, etc. Use *openssl rsa* command in a proper format to see those fields and include results in your report.

- The company should generates a Certificate Signing Request (CSR), which basically includes the company's public key. The CSR will be sent to the CA, who will generate a certificate for the key (usually after ensuring that identity information in the CSR matches with the server's true identity). Please use example.com as the common name of the certificate request.

  ```
  $ openssl req -new -key server.key -out server.csr -config openssl.cnf
  ```

  Use *openssl req* command in a proper format to see the content of .csr file and include it in your report. [1]

[1] https://www.openssl.org/docs/man1.0.2/man1/x509.html
[2] https://www.openssl.org/docs/man1.0.2/man1/openssl-rsa.html
[1] https://www.openssl.org/docs/man1.0.2/man1/openssl-req.html

- The CSR file needs to have the CA's signature to form a valid certificate. In the real world, the CSR files are usually sent to a trusted CA for their signature. In this lab, we will use our own trusted CA to generate certificates. The following command turns the certificate signing request (server.csr) into an X509 certificate (server.crt), using the CA's ca.crt and ca.key:

```
$ openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key -config
    openssl.cnf
```

  Then display that new certificate using the *openssl x509* command and include the output in your report.

### 2.3 TLS Server 45%

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 ). The server code has been given in *server.py* file.

Use the client program developed previously to test this server program. Remember that, 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 adding the CA's certificate to that folder, because that will affect the entire system. Instead, store the CA's certificate in the "*./certs*" folder.
Test your program first using the */etc/ssl/certs* folder and then with the *./certs* folder. Describe and explain your observations.

## 3 TLS Security: 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 a 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 that integrates the client and server programs from previous tasks. The basic working principle is shown in Figure 2. 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, you should be able to get a basic proxy working.

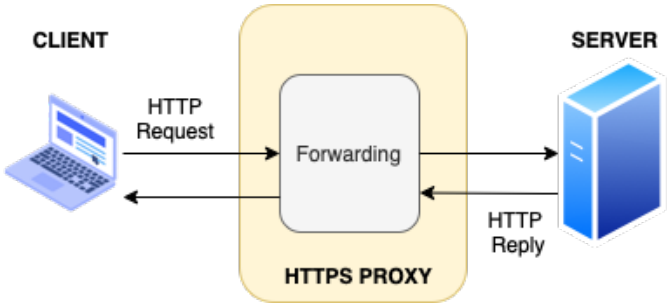


Figure 2: Order of loaded images in VirtualBox to run the automated installer.

The purpose of this task is to use this simple proxy to understand how a 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 tasksheet is on TLS, you can choose two different servers and demonstrate that your proxy works for those servers. If you are interested in a product-quality HTTPS proxy, one example is the open-source mitmproxy. [1]

**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 best for the proxy to spawn a thread to process that request, so it 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 re- quest 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
```

[1] https://mitmproxy.org

```
        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()
```

**Implementing HTTP proxy** : You should implement the simple HTTPS proxy. 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.

Popular servers, such as Facebook, have complicated login mechanisms, so feel free to find a server that has a simple login mechanism. The assumption of this MITM attack is that the attacker has compromised a trusted CA, and is able to generate a 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 a certificate for any web server.

**Hint:** You may want to run the victim browser on one VM (victim VM), and run your proxy on another VM (attacker VM). In a 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. Instead, 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) to simulate redirection attacks. So that, the victim's traffic to www.example. com will be redirected to the attacker's VM, where your proxy is waiting for HTTP requests. It should be noted that on the attacker's VM, the hostname www.example.com should point to the real web server.

Report your findings in detail with screenshots, to describe what you have done and what you have observed. You also need to provide an explanation for your observations. Together with your report, also submit your HTTPS proxy code.

# 4 Certificate Pinning 10%

**Basics** What is certificate pinning, what are its uses, and what are its limitations? What should you pin, and why?

**Implementation** Write a small script that makes an HTTPS request. Then, implement certificate pinning. Try to use your proxy with and without HTTPS pinning in your script. What is the result? (Include some screenshots here)