

## Network Sniffing and TLS Downgrade

- Exercise 1: [attack] Sniffing credit cards numbers
  - MitM
  - Using Wireshark to inspect traffic inside a docker network
  - Using the mitm container as a router
  - Building the MitM script
  - Parsing the packets for Credit Cards and Passwords
  - Going Further... (Optional)
  - Responsible disclosure
- Exercise 2: [attack] TLS Downgrade
  - Understanding the handshake
  - Performing the attack
  - Understanding which bytes to change
- Exercise 3: [defense] Secure NGINX configuration
  - Part A: A Docker with nginx
  - Part B: Self-signed HTTPS
  - Part C: Signed certificate
    - \* Part C bis: HSTS

### Exercise 1: [attack] Sniffing credit cards numbers

In this **unethical** exercise (which you should certainly not reproduce outside this class), you are in the cafeteria of Evil Corp. After poking around the network, you had a good surprise: their router had the default login `root:1234` which allowed you to log in and gain control of the device.

You realize that a device nearby is sending credit card and password information in plaintext, over HTTP. It is time to punish Evil Corp! Find the 3 sensitive information sent by the machine by performing a Man-in-the-middle attack.

#### MitM

The goal of a man-in-the-middle is to intercept the traffic; in our scenario, the client is sending the traffic to the router over WPA2, so you can't see the traffic in plaintext over this link. However, by declaring your machine as the "default route" of the LAN (in other terms, the exit router), you can make clients send you their traffic willingly. Since, in this scenario, the traffic is only encrypted at the link layer but not at the transport layer, you will receive it in plaintext. What you do with it is up to you: in our case, we simply want to take a look at it, and then forward it to its real destination, so the victim does not get suspicious.

This exercise uses `docker compose`, a wrapper around `docker` which helps with the orchestration of multiple containers. If you don't have it installed on your system yet, you should install it.

As you see in the provided `compose.yaml` which specifies the container setup, we provide you with two containers: the `client`, which generates the secrets and sends them to some webserver, and the `mitm` container which will capture the traffic. For the sake of this exercise, do not read the secrets from the code in the client (this is trivial and uninteresting).

To start both containers, simply run `docker compose up`. This command will create a `client` and `mitm` as per the description in the `compose.yaml` file.

## Interpreting output

After the containers are started, you will see the output of both containers on the terminal. In fact, each line of output will be prepended by the name of the container.

After ~10 seconds, you will see a lot of lines printed by the client. Some of these lines are shown below:

```
client      | Client booting... (10 seconds)
client      | Sending .....
client      | <html>
```

These are printed by the client container, which tries requests to `http://neverssl.com`.

You will notice that the `mitm` container prints nothing: the client sends its secrets (among other things) periodically to a webserver, and the `mitm` machine sees nothing of this, as it would be the case in a normal LAN. Verify this using Wireshark (see instructions below).

## Using Wireshark to inspect traffic inside a docker network

The containers created with `docker compose` are located in a docker network (more about docker network [here](#)). In order to be able to see the traffic in this network from your host, you will first need to find which interface from your host is used to connect to this network. For this purpose, do the following:

- Find the `mitm` IP address in the docker network by running from your host: `docker inspect mitm | grep "IPAddress"`. Consider the IP address printed and copy the prefix denoted by the first two bytes (if the address is `172.19.0.3`, copy `172.19`).
- Find the interface on your host connected to this network with the following command `ip neigh | grep "<the copied prefix>"` (with the previous example, it would be `ip neigh | grep "172.19"`). The value on the 3rd column, starting with `br-` is the interface you are looking for.

When you start Wireshark on your host, you should see and be able to select this interface. It will display the traffic inside the docker network, from the point of view of your host.

## Using the mitm container as a router

The first task for this exercise is to make the traffic of `client` transit through the `mitm` container. Conceptually, you will need to:

1. Find out the local IP address of the `mitm` container;
2. Open a shell in the client, [remove](#) the *default route*;
3. Open a shell in the client, add a new *default route* with the IP of the `mitm` container as the destination;
4. Enable Network Address Translation (NAT) in the `mitm` container so it also receives the replies, using `iptables` (see commands below).

Hint: You can open a shell in a running container named `client` by running the command `docker compose exec client /bin/bash`.

The commands for point 4 are:

```
iptables -A FORWARD -i eth0 -j ACCEPT
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

In this context, what does the `sysctl`s part in the `compose.yaml` achieve? Why is it required?

Other useful commands :

- `docker inspect`
- `route`
- `ip route`

At this point, the `client` machine should have Internet connectivity, and the traffic should go through `mitm`: check this using Wireshark or `traceroute`. Note that this is a crude attack, and more sophisticated ones may not be detectable by those methods. Also, if you ever stop/restart the containers, you will need to re-do the configuration. You can put them in a script and execute it when you start the container (for `mitm`) or copy-paste (for `client`).

**Isn't this cheating?** You may have noticed that steps 2-3 require to open a shell on the client machine, which is theoretically impossible without knowing its password, etc. Actually, those commands are to make your life simpler in this exercise. However, you could certainly achieve the same *effect* without opening a shell on the client; do you see how?

*Hint:* how are those values set in the first place?

### Building the MitM script

In the same directory as `compose.yaml`, create a file `mitm.py`. Notice that due to the volume declared in the `compose.yaml`, this file also co-exists in the `mitm` machine and is automatically synced: you can develop on your host, and simply run the code from within the container.

To process the packets in Python, we will use [NetfilterQueue](#).

First, in the `mitm` container, route the traffic to the Queue (for later processing):

```
iptables -D FORWARD -i eth0 -j ACCEPT
iptables -A FORWARD -j NFQUEUE --queue-num 1
```

**Note:** After that, all packets will be put in a queue until you accept them with your script. If you don't have a script listening, *client* won't have internet access anymore.

Then, follow the first example from the [NetfilterQueue website](#) to bind on the queue and receive packets in the function `print_and_accept()`.

If you open a shell in the `mitm` machine, you can now run `mitm.py` with `python3` and you should see packets.

### Parsing the packets for Credit Cards and Passwords

The client machine is using the HTTP headers to send sensitive information; more precisely, you should find a combination of:

- `cc --- number`, where `number` is `xxxx.xxxx.xxxx.xxxx`,
- `pwd --- password`, where `password` is a mix of 0-9 digits, *uppercase* letters A-Z, symbols  `;<=>?@`.

The client sends those among other data at random intervals; once your capture script is working, simply wait for the client to send out the secrets. You should find **3** different secrets!

Finally, notice that the capture format of `NetfilterQueue` is raw bytes, which is not really handy. How can we filter HTTP packets for instance? We recommend you to use the [scapy](#) library to parse the raw packets: a starting point could be [here](#), section "Stealing Email Data". Notably, you'll find the snippets `ip = IP(pkt.get_payload()); ip.haslayer(Raw)` and `http = ip[Raw].load.decode()` useful.

### Going Further... (Optional)

Additionally, the client sends random-looking data but also misleading information. You can automatically process the data with [Regex](#) to filter out the misleading information.

### Responsible disclosure

What actions should you take if you see an insecure router in a cyber-cafe in real life? Is it illegal to access the device if, for instance, it does not even have a password set?

## Exercise 2: [attack] TLS Downgrade

This is a follow-up exercise: suppose now that some client in the cafe is using [TLS](#) to secure their communications. TLS is end-to-end encrypted, and if the client knows the public key of the server (for instance, because it visited it before), then a third-party is unable to man-in-the-middle the connection.

**Ask yourself:** Why is the knowledge of the public key necessary for secure communications? What attack is possible if neither the client nor the server have *any* prior knowledge of the other?

However, a TLS connection can use various [cipher suites](#): notably, the server and the client agree on a common encryption algorithm, hash function, etc. Since this happens at the very beginning of the TLS connection, this part of the protocol (called the *handshake*) is unencrypted. In this exercise, you are asked to **downgrade** the quality of a TLS connection between a client and a server: in other terms, make them use a (slightly) less secure encryption or hash function, while both could have used the most secure variant.

More precisely, you will prevent the use of AES256 and force the use of AES128. To prevent these kinds of attacks (which can very well happen in the real world), the TLS standards *deprecates* weak and old cipher suites: hence, you couldn't downgrade "all the way" down to a very weak cipher, e.g., [RC4](#): the client and server would simply refuse to establish such connection.

Note that, perhaps ironically, you could drop completely the TLS packets: this would result in a denial-of-service, but some clients would then try the insecure version over HTTP instead of HTTPS. In real life, [HSTS](#), a mechanism to prevent these kinds of downgrade attacks is used.

### Understanding the handshake

To perform this exercise, you will need a basic understanding of the TLS handshake: the [RFC](#) is the most precise resource available, or you can read some [more user-friendly explanation](#).

### Performing the attack

This part will be extremely similar to Exercise 1, with the difference that you will modify packets on-the-fly (rather than just read them).

Start the containers described in the `compose.yaml` with `docker compose up` just like in Exercise 1. Use the same *default route*/forwarding/NAT trick as before to route the traffic towards the `mitm` machine. Start with the same python script (which uses `NetfilterQueue` and `scapy`) as before.

*Note:* sometimes `iptables` gets messed up, and requires a reboot. To avoid all unnecessary frustration, check every step carefully with Wireshark.

## Understanding which bytes to change

The simplest approach here is to fire up Wireshark, and explore the traffic manually. You'll see that Wireshark explains the meaning of each byte when you hover over them.

7.065590020	172.17.0.6	46.101.101.102	53392 → 443	[SYN]	Seq=1477480218 Win=29200 Len=0 MSS=1460 SACK_PER...	74 TCP
7.097596874	172.17.0.6	46.101.101.102	53392 → 443	[ACK]	Seq=1477480219 Ack=2596816304 Win=29312 Len=0 TS...	66 TCP
7.098453821	172.17.0.6	46.101.101.102		Client Hello		272 TLSv1.2
7.151935163	172.17.0.6	46.101.101.102	53392 → 443	[ACK]	Seq=1477480425 Ack=2596817712 Win=32128 Len=0 TS...	66 TCP
7.152054641	172.17.0.6	46.101.101.102	53392 → 443	[ACK]	Seq=1477480425 Ack=2596818876 Win=34944 Len=0 TS...	66 TCP
7.152762117	172.17.0.6	46.101.101.102	53392 → 443	[FIN, ACK]	Seq=1477480425 Ack=2596818876 Win=34944 Len...	66 TCP
7.153301245	172.17.0.6	46.101.101.102	53394 → 443	[SYN]	Seq=3749450197 Win=29200 Len=0 MSS=1460 SACK_PER...	74 TCP
7.184796792	172.17.0.6	46.101.101.102	53394 → 443	[ACK]	Seq=3749450198 Ack=4141274764 Win=29312 Len=0 TS...	66 TCP
7.184991379	172.17.0.6	46.101.101.102		Client Hello		272 TLSv1.2
7.207314426	172.17.0.6	46.101.101.102	53392 → 443	[ACK]	Seq=1477480426 Ack=2596818877 Win=34944 Len=0 TS...	66 TCP
7.245272740	172.17.0.6	46.101.101.102	53394 → 443	[ACK]	Seq=3749450404 Ack=4141277336 Win=34432 Len=0 TS...	66 TCP
7.248003188	172.17.0.6	46.101.101.102	53394 → 443	[FIN, ACK]	Seq=3749450404 Ack=4141277336 Win=34432 Len...	66 TCP
7.295470579	172.17.0.6	46.101.101.102	53394 → 443	[ACK]	Seq=3749450405 Ack=4141277337 Win=34432 Len=0 TS...	66 TCP

Figure 1: Wireshark

▼ Secure Sockets Layer		
▼ TLSv1.2 Record Layer: Handshake Protocol: Client Hello		
Content Type: Handshake (22)		
Version: SSL 3.0 (0x0300)		
Length: 201		
▼ Handshake Protocol: Client Hello		
Handshake Type: Client Hello (1)		
Length: 197		
Version: TLS 1.2 (0x0303)		
▼ Random		
GMT Unix Time: Jan 15, 2018 09:24:24.000000000 CET		
Random Bytes: 555a4159585951454e5559564b4b434e484c434c464a5955...		
Session ID Length: 0		
Cipher Suites Length: 2		
▼ Cipher Suites (1 suite)		
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)		
Compression Methods Length: 1		
► Compression Methods (1 method)		
Extensions Length: 154		
► Extension: elliptic_curves		
► Extension: ec_point_formats		
0000	02 42 ac 11 00 07 02 42 ac 11 00 06 08 00 45 00	.B....B.....E.
0010	01 02 dd a6 40 00 40 06 1c 6d ac 11 00 06 2e 65	....@.@..m.....e
0020	65 66 d6 1e 01 bb 3f f4 b4 68 4b 4e 3d 94 80 18	ef....?.hKN=...
0030	00 e5 40 d7 00 00 01 01 08 0a 56 57 ed 1a 1a 74	..@.....VW...t
0040	34 ce 16 03 00 00 c9 01 00 00 c5 03 03 5a 5c 65	4.....Z\ve
0050	38 55 5a 41 59 58 59 51 45 4e 55 59 56 4b 4b 43	8UZAYXYQ ENUYVKKC
0060	4e 48 4c 43 4c 46 4a 59 55 48 4d 54 48 00 00 02	NHLCFLFY UHMT...
0070	00 35 01 00 00 9a 00 0a 00 4c 00 4a ff 02 ff 01	.5.....L.J....
0080	00 1a 00 1b 00 1c 00 1d 00 1e 01 00 01 01 01 02	.....
0090	01 03 01 04 00 0f 00 10 00 11 00 12 00 13 00 14	.....
00a0	00 15 00 16 00 17 00 18 00 19 00 01 00 02 00 03	.....

Figure 2: Byte explanation

Using this knowledge, edit the payload of the message (which you can get using `ip["Raw"].load`) at the correct position, and set the new payload in the packet with `pkt.set_payload(new_payload)`.

Finally, observe the new handshake in Wireshark again. What changed? How did your active MitM script affect the security of the TLS connection?

## Exercise 3: [defense] Secure NGINX configuration

In this exercise, you will set up a simple [nginx](#) web server. In real life, this would happen if you want to host your CV online on your own server, for instance.

The goal is to have a *somewhat* secure server: do not serve HTTP, but rather immediately redirect to HTTPS, with a correctly signed certificate. By “somewhat”, we wish to emphasize that keeping a fully-secure infrastructure is more of a full-time job but this should show you the basics.

### Part A: A Docker with nginx

The server directory contains a `default.conf` file (this will be the configuration of nginx), and a file `index.html` with any recognizable content (this will be served by nginx).

Now, if you start the containers with `docker compose up --build`, your server will be created, and the verifier image will test if it behaves correctly. The `compose.yaml` maps ports 80 and 443 on your host to ports 80 and 443 in the server container, respectively. These are the default HTTP(S) ports.

Apart from the verifier, you can manually check that the server is working correctly by browsing to `http://localhost:80`.

### Part B: Self-signed HTTPS

We will now add HTTPS support. For this first variant, you will use a *self-signed certificate*. In short, this will provide confidentiality, but clients who visit your website will have no idea who generated this certificate (ask yourself: why would this be a problem?).

The procedure is explained in Step 1 of [this tutorial](#) to create the keys. Do not blindly follow the steps, your configuration is slightly different; notably, your nginx is in a docker, so restarting nginx in your case means rebuilding the container, also you need `openssl` which is given to you *in the container* (depending on your host OS, you might also have it outside). Put the key and the certificate in the `/certs` directory in the container.

N.B. This is for ease of use; best practices on certificates locations is to put the key in `/etc/ssl/private/` and the certificate in `/etc/ssl/certs/`.

**Important:** as a “Common Name”, use `server`. (if you feel good today, ask yourself why it is `server` and what it would be in the real world. This requires a decent understanding of how docker works.)

Once done, you can restart the container, and manually test your setup at `127.0.0.1:443` or equivalently `https://127.0.0.1`. You should see a warning sign: indeed, your certificate is not trustworthy.

**Hint:** If you're lost; the procedure is: generate keypairs, add them to the docker image (e.g., change `Dockerfile` and add `COPY` instructions), change `default.conf` to use these keys and to serve TLS. Rebuild the image.

As a final step, we want to redirect the HTTP traffic towards HTTPS. Add the correct 301 redirection in your `default.conf` to forward visitors to the HTTPS version of your website. To manually test your setup, browse to `http://127.0.0.1`: you should be redirected to `https://127.0.0.1`.

When setting up a server with `ssl`, do not forget to specify the `ssl` certificate and private key path in the nginx server config `default.conf`.

### Part C: Signed certificate

The problem with the above certificate is that it was self-signed, or in other terms, *not* signed by anyone trustworthy. In the real world, the [Certificate Authorities](#) are the “trustworthy” entities.

In this exercise, we implemented a fake CA which will sign any certificate you send, so you see the process. (In real life, what would a CA check?)

First, you need to create a Certificate Signing Request `.csr`. Read the section “Create a certificate” [here](#). Remember, the “Common Name” still needs to be `server`. Similar to

above, you can start a shell in the `verifier` container (which already shares a directory with your host) and use `openssl` in the container if you don't have it installed on your host system already.

Then, notice that our `verifier` container automatically signs any file named `request.csr` next to the `compose.yaml`. Place your `.csr` there, and restart the containers. The freshly-signed `.crt` should appear.

The next step is to put the `request.key` and `request.crt` in `./server`, and copy them on the docker, to `/certs`. Then adapt the `default.conf` file accordingly.

**Note:** In real life, this process can be automatized by using the excellent tool `certbot` from [Let's Encrypt](#). Unfortunately for you, this requires a public-facing webserver and does not apply in this exercise.

Now, swap the previous self-signed certificate with this new pair of certificates, and re-run the containers.

**Part C bis: HSTS** As a final step, add [HSTS](#) to your `nginx`, to prevent the downgrade attack seen in Exercise 2. This will make visitors “remember” that there exists an HTTPS version, and that the HTTP version should not be used.

**Debugging tips:** Once the HSTS header has been set, your browser will *refuse* to connect to the HTTP version. This shouldn't be a problem, but if you want to remove this setting, follow [this tutorial](#).