Homework 4

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 willingly their traffic. 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. Create a file `docker-compose.yml` with the following contents:

```yaml
version: '3.3'

services:
  client:
    container_name: client
    image: com402/hw4ex1_client
    privileged: true
  mitm:
    container_name: mitm
    image: mitm
    image: com402/hw4ex1_mitm
    privileged: true
    volumes:
      - '.:/app'
```

As you see, 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 --build` (the `--build` flags instructs docker-compose to rebuild images whenever there is a change, which is handy).

You will notice that nothing happens: 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.

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. Make sure the `mitm` container is forwarding traffic through the flag `net.ipv4.ip_forward` (the flag should be set to `1`)
5. Enable Network Address Translation (NAT) so the `mitm` also receives the replies, using `iptables`

The commands for point 5 are:

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

At this point, the `client` machine should have Internet connectivity, and the traffic should go through `mitm`: check this using Wireshark.

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 his 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 `docker-compose.yml`, create a file `mitm.py`. Notice that due to the volume declared in the `docker-compose.yml`, 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
```

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!

Additionally, the client sends random-looking data but also misleading information. You should automatically process the data with Regex.

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(packet); ip.haslayer(Raw)` and `http = ip[Raw].load.decode()` useful.

Responsible disclosure

If we remove the fantasy context of this exercise, what actions should you take if you see an unsecured 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 has *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 a server: in other terms, make them use a (slightly) less secure 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 is a mechanism to prevent these kinds of downgrade attacks.

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).

Create a file `docker-compose.yml` with the following contents:

```yaml
version: '3.3'

services:
  client:
    container_name: client
```

```
    image: com402/hw4ex2_client
    privileged: true
  mitm:
    container_name: mitm
    image: mitm
    image: com402/hw4ex2_mitm
    privileged: true
    volumes:
      - '.:/app'
```

And start them with `docker-compose up --build`. 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

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)`.

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

Create a folder `server`, with inside the following `Dockerfile` :

```
FROM nginx:1.15-alpine

RUN apk update
RUN apk add openssl
```

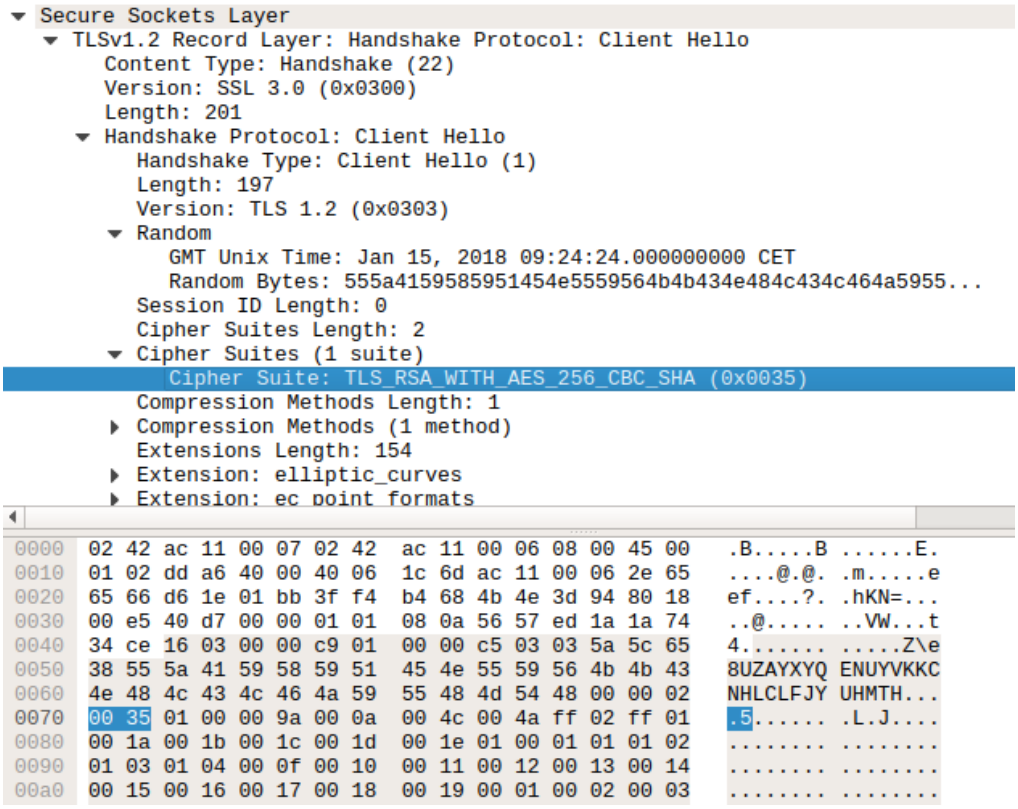

Figure 2: Byte explanation

```
RUN rm -f /etc/nginx/conf.d/*
RUN mkdir /certs/
RUN mkdir /www

COPY index.html /www
COPY default.conf /etc/nginx/conf.d/default.conf
```

Next to `Dockerfile`, create an empty `default.conf` file (this will be the configuration of nginx), and a file `index.html` with any recognizable content, say "Hello World" (this will be served by nginx).

Now, next to the `server` folder, create the following `docker-compose.yml`:

```
version: '3.3'
services:
  client:
    container_name: server
    image: server
    build:
      context: ./server
      dockerfile: Dockerfile
    ports:
      - "127.0.0.1:80:80"
      - "127.0.0.1:443:443"
  verifier:
    container_name: verifier
    image: com402/hw4ex3_verifier
    links:
    - client
    volumes:
      - '.:/current_dir'
```

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. Right now, the container will immediately crash: you haven't provided a valid configuration for nginx in the aforementioned `default.conf`.

This is your first task: add some basic configuration to `default.conf` to serve the `\www` folder over port 80. After rebuilding the image, you can manually test that this is working by opening your browser at the URL `127.0.0.1:80`.

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, then Step 2 and further of this tutorial. 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).

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 docker, 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`.

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`.

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

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.

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.

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