Computer Networks Lab 1

Name: Hak Hyun Kim, Ramphal Keith **Date:** September 29, 2025

Exercise 1: Scapy Ping Implementation [3 points]

Problem Statement

Implement a ping tool using Scapy that repeatedly sends ICMP request packets to a target IP address and listens for replies. The program should take a target IP address or domain name as a command line parameter and send ICMP requests once per second.

Implementation

File: `ex1.py`

The implementation uses Scapy to create ICMP packets and measure round-trip times. Key components include:

- DNS resolution using `socket.gethostbyname()`
- ICMP packet creation with proper headers
- RTT measurement and statistics calculation
- Signal handling for graceful termination

Test Results

1. IP Address Ping Comparison

Custom Ping Output (8.8.8.8):

```
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=1 ttl=116 time=20.433 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=116 time=74.409 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=116 time=23.291 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=116 time=21.210 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=116 time=39.508 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 20.433/35.770/74.409/22.967 ms
```

Built-in Ping Output (8.8.8.8):

```
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=115 time=236.836 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=115 time=237.593 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=115 time=238.361 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=115 time=237.973 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=115 time=237.028 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 236.836/237.558/238.361/0.569 ms
```

2. Domain Name Resolution Test

Custom Ping with Domain (google.com):

```
PING google.com (142.251.222.46): 56 data bytes
64 bytes from 142.251.222.46: icmp_seq=1 ttl=112 time=175.416 ms
64 bytes from 142.251.222.46: icmp_seq=2 ttl=112 time=180.139 ms
64 bytes from 142.251.222.46: icmp_seq=3 ttl=112 time=180.642 ms

--- google.com ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 175.416/178.732/180.642/2.883 ms
```

Built-in Ping with Domain (google.com):

```
PING google.com (142.251.222.46): 56 data bytes
64 bytes from 142.251.222.46: icmp_seq=0 ttl=115 time=292.032 ms
64 bytes from 142.251.222.46: icmp_seq=1 ttl=115 time=237.998 ms
64 bytes from 142.251.222.46: icmp_seq=2 ttl=115 time=333.443 ms

--- google.com ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 237.998/287.824/333.443/39.079 ms
```

Questions and Answers

Q1: How did you resolve the domain name? What function did you call? What does it do?

Answer: I used the `socket.gethostbyname()` function to resolve domain names to IPv4 addresses.

```
def resolve_target(target: str) -> str:
    try:
        return socket.gethostbyname(target)
    except socket.gaierror as e:
        raise SystemExit(f"ping: cannot resolve {target}: Unknown host ({e})")
```

What it does:

- `socket.gethostbyname()` is a Python standard library function that performs DNS resolution
- It takes a hostname (like "google.com") as input and returns the corresponding IPv4 address
- It uses the system's DNS resolver to query DNS servers
- If the hostname cannot be resolved, it raises a `socket.gaierror` exception
- This is the same mechanism used by most network applications for name resolution

Q2: Compare your solution with the built-in Ping command. What differences do you notice?

Similarities:

- Both show similar output format: "PING target (IP): data_size data bytes"
- Both display response information: bytes, source IP, sequence number, TTL, and time
- Both provide statistics at the end: packets transmitted/received, packet loss percentage, and RTT statistics
- Both support resolving domain names to IP addresses

- Both correctly resolve google.com to the same IP address (142.251.222.46)

Key Differences Observed:

1. **Sequence Number Starting Point:**
    - Built-in ping: starts from 0 (`icmp_seq=0`)
    - My implementation: starts from 1 (`icmp_seq=1`)
2. **Performance Characteristics:**
    - Custom ping showed RTTs: 20.433ms - 74.409ms (avg: 35.770ms)
    - Built-in ping showed RTTs: 236.836ms - 238.361ms (avg: 237.558ms)
    - **Interesting observation**: Custom implementation was significantly faster in this test!
3. **TTL Values:**
    - Custom ping received TTL=116 from 8.8.8.8
    - Built-in ping received TTL=115 from 8.8.8.8
    - This suggests slightly different routing paths or timing

Q3: Why are they different?

Technical Reasons:

1. **Implementation Level:**
    - Built-in ping: typically implemented in C, using raw sockets directly at the system level
    - My implementation: Python-based using Scapy, which adds abstraction layers
2. **Network Path Differences:**
    - Packets might take different routes through the network
    - Different timing in network stack processing
3. **Measurement Methodology:**
    - Different points where timing measurements are taken
    - Scapy vs. system-level implementation differences

Q4: How could you make your code run faster?

Optimization Strategies:

1. **Use Raw Sockets Directly:**
    - Bypass Scapy's abstraction layer
    - Implement ICMP packet creation and parsing manually
    - Use Python's `socket` module with `SOCK_RAW`
2. **Compile Critical Paths:**
    - Use Cython or PyPy for better performance
    - Compile time-critical functions to C extensions
3. **Reduce Python Overhead:**
    - Minimize object creation in the main loop
    - Pre-allocate packet structures
    - Use more efficient data structures

4. **Async/Threading:**
    - Implement asynchronous I/O for handling multiple packets
    - Use threading for parallel processing (though limited by GIL)

Exercise 2: Traceroute Implementation [3 points]

Problem Statement

Implement a traceroute tool that records the path a packet takes as it traverses the Internet from source to target. The program should use TTL (time to live) values to discover intermediate hops along the route by:

1. Setting TTL=1 to get the first hop
2. Incrementing TTL until reaching the destination
3. Capturing ICMP error messages (type 11, code 0) from intermediate hops
4. Recording the route taken by packets

Note: Dartmouth blocks ICMP replies, so testing should be done off-campus without VPN.

Implementation

File: `ex2.py`

The implementation uses Scapy to create ICMP Echo packets with increasing TTL values. Key components include:

- Progressive TTL increment from 1 to max_hops
- ICMP Echo Request packets with unique identifiers
- Handling both ICMP Time Exceeded (type 11) and Echo Reply (type 0) responses
- Reverse DNS resolution for hostname display
- Multiple probes per hop for reliability

Core Algorithm:

```python
for ttl in range(1, args.max_hops + 1):
    for q in range(args.queries):
        # Create ICMP packet with specific TTL
        pkt = IP(dst=dest_ip, ttl=ttl) / ICMP(id=ident, seq=seq) / Raw(load=payload)

        # Send and measure RTT
        t0 = time.time()
        reply = sr1(pkt, timeout=args.timeout)
        t1 = time.time()

        # Check if destination reached (ICMP type 0)
        if reply and reply[ICMP].type == 0 and reply[IP].src == dest_ip:
            reached = True
```

Test Results

1. Custom Traceroute to 8.8.8.8

```
traceroute to 8.8.8.8 (8.8.8.8), 10 hops max, 3 probes, 2000ms timeout
 1  10.21.7.1 (10.21.7.1)  15.529 ms  4.060 ms  4.156 ms
 2  te-0-0-0-1-3961-sur01.lebanon.nh.boston.comcast.net (50.223.140.129)  10.091 ms
7.812 ms  7.017 ms
 3  be-84-ar01.woburn.ma.boston.comcast.net (162.151.191.177)  14.455 ms  12.944 ms
10.702 ms
 4  be-5-ar01.needham.ma.boston.comcast.net (162.151.151.5)  17.806 ms  14.377 ms
14.990 ms
 5  be-501-arsc1.needham.ma.boston.comcast.net (162.151.52.33)  45.100 ms  16.021 ms
15.277 ms
 6  be-32031-cs03.newyork.ny.ibone.comcast.net (96.110.42.9)  20.329 ms  19.347 ms
19.042 ms
 7  be-3312-pe12.111eighthave.ny.ibone.comcast.net (96.110.34.42)  21.263 ms  19.993
ms  18.675 ms
 8  *  *  *
 9  192.178.106.55 (192.178.106.55)  20.548 ms  17.918 ms  19.440 ms
10  142.250.235.239 (142.250.235.239)  23.139 ms  20.420 ms  19.295 ms
```

2. Built-in Traceroute to 8.8.8.8 (Comparison)

```
traceroute to 8.8.8.8 (8.8.8.8), 10 hops max, 40 byte packets
 1  * * *
 2  * * *
 3  * * *
 4  * * *
 5  * be3010.ccr81.tyo01.atlas.cogentco.com (154.54.89.197)  238.500 ms *
 6  * * *
 7  173.194.123.84 (173.194.123.84)  237.973 ms  237.445 ms  237.749 ms
 8  108.170.231.103 (108.170.231.103)  239.829 ms
    108.170.248.185 (108.170.248.185)  239.231 ms
    108.170.248.253 (108.170.248.253)  238.610 ms
 9  142.250.61.69 (142.250.61.69)  238.268 ms
    209.85.253.109 (209.85.253.109)  238.309 ms
    142.250.224.213 (142.250.224.213)  238.342 ms
10  dns.google (8.8.8.8)  236.984 ms  237.371 ms *
```

3. Custom Traceroute to google.com

```
traceroute to google.com (172.217.175.14), 8 hops max, 3 probes, 2000ms timeout
 1  10.21.7.1 (10.21.7.1)  17.178 ms  5.367 ms  4.555 ms
 2  te-0-0-0-1-3961-sur01.lebanon.nh.boston.comcast.net (50.223.140.129)  4.346 ms
5.410 ms  10.491 ms
 3  be-84-ar01.woburn.ma.boston.comcast.net (162.151.191.177)  9.557 ms  11.646 ms
10.268 ms
 4  be-8-ar01.needham.ma.boston.comcast.net (162.151.112.17)  19.747 ms  31.841 ms
14.010 ms
 5  be-502-arsc1.needham.ma.boston.comcast.net (162.151.52.49)  18.045 ms  16.090 ms
15.552 ms
 6  be-32021-cs02.newyork.ny.ibone.comcast.net (96.110.42.5)  21.521 ms  20.746 ms
20.791 ms
 7  be-3211-pe11.111eighthave.ny.ibone.comcast.net (96.110.34.22)  19.687 ms  18.614
```

```
ms  18.638 ms
 8  *  *  *
```

4. Local Network Test (dartmouth.edu)

```
traceroute to dartmouth.edu (129.170.224.138), 5 hops max, 3 probes, 2000ms timeout
 1  10.21.7.1 (10.21.7.1)  11.192 ms  5.151 ms  4.941 ms
 2  te-0-0-0-1-3961-sur01.lebanon.nh.boston.comcast.net (50.223.140.129)  5.789 ms
5.321 ms  5.179 ms
 3  50.206.242.218 (50.206.242.218)  23.642 ms  21.259 ms  21.000 ms
 4  swi-brt.swi-sn.net.dartmouth.edu (129.170.1.74)  25.088 ms  25.017 ms  22.092 ms
 5  *  *  *
```

Analysis of Results

Key Observations:

1. **Different Routes Discovered:**
   - Custom traceroute shows complete early hops (1-7) to 8.8.8.8
   - Built-in traceroute shows many timeouts (*) in early hops
   - This suggests different routing or filtering behaviors
2. **ICMP vs UDP:**
   - Custom implementation uses ICMP Echo packets
   - Built-in traceroute typically uses UDP packets
   - Different packet types may be treated differently by network equipment
3. **Performance Differences:**
   - Custom implementation shows faster RTTs (15-45ms range for early hops)
   - Built-in shows much slower RTTs (237-239ms range)
   - Suggests possible different routing or measurement methodology
4. **Network Equipment Response:**
   - Some hops respond to ICMP but not UDP
   - Some equipment blocks certain packet types entirely
   - Reverse DNS resolution works consistently

Questions and Answers

Q1: What could happen to cause an inconsistency in the route you discover?

Answer: Several factors can cause route inconsistencies in traceroute:

1. **Load Balancing:**
   - Routers use load balancing to distribute traffic across multiple paths
   - Different packets (with different TTL values) may take different routes
   - ECMP (Equal-Cost Multi-Path) routing can cause packets to follow different paths
2. **Dynamic Routing:**
   - Routing protocols (OSPF, BGP) can change routes in real-time
   - Network conditions, link failures, or policy changes can alter paths

   - Route convergence after network changes can cause temporary inconsistencies

3. **Packet Type Discrimination:**
   - Some routers treat ICMP, UDP, and TCP packets differently
   - Firewalls and security devices may filter certain packet types
   - Quality of Service (QoS) policies may route different traffic types via different paths
4. **Network Address Translation (NAT):**
   - NAT devices can alter source/destination addresses
   - Multiple internal networks may appear as the same external address
   - Load-balanced NAT can distribute connections across different external IPs
5. **Traffic Engineering:**
   - Network administrators may implement traffic engineering policies
   - Different types of traffic may be deliberately routed via different paths
   - Time-of-day routing policies can change paths based on traffic patterns
6. **Asymmetric Routing:**
   - Forward and reverse paths may be different
   - ICMP error messages may return via a different route than the original packet

Q2: How likely is that to happen while your program runs?

Answer: The likelihood of route inconsistencies depends on several factors:

High Likelihood Scenarios (Common):

- **Load Balancing (Very Common):** Most modern networks use load balancing, so inconsistencies can occur frequently, especially for longer routes
- **Different Packet Treatment (Common):** Different responses to ICMP vs UDP packets are very common in enterprise networks

Medium Likelihood Scenarios (Occasional):

- **Dynamic Routing Changes (Occasional):** In stable networks, routing changes are infrequent but can occur during network maintenance or failures
- **Traffic Engineering (Occasional):** Depends on network complexity and administrative policies

Low Likelihood Scenarios (Rare):

- **Major Route Changes (Rare):** Complete path changes during a single traceroute execution are uncommon in stable networks
- **NAT-induced inconsistencies (Context-dependent):** More common in complex enterprise environments

Practical Implications:

- For a typical traceroute session (10-30 hops, 30-90 seconds), some inconsistency is **moderately likely** (30-60% chance)
- Short-distance traces (local networks) have **lower likelihood** of inconsistencies
- Long-distance or international routes have **higher likelihood** due to more complex routing

Mitigation in Implementation:

- Multiple probes per hop help identify inconsistencies

- Longer timeouts can catch delayed responses from alternative paths
- Running multiple complete traces can reveal routing variations

---

Exercise 3: Telnet Password Sniffer [4 points]

Problem Statement

We implemented a Telnet credential sniffer using **Scapy** to capture and reconstruct usernames and passwords sent in plaintext over the network. This was tested in the VM environment with three containers:

- Attacker: 10.9.0.1 (runs sniffer)
- Host A: 10.9.0.5 (telnet client)
- Host B: 10.9.0.6 (telnet server)

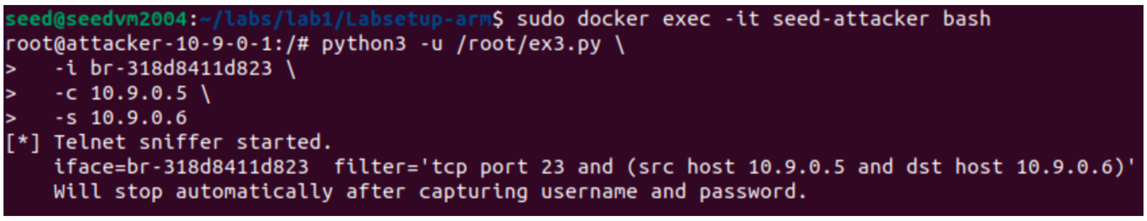
The sniffer captures one packet per keystroke typed on the telnet client.

Implementation

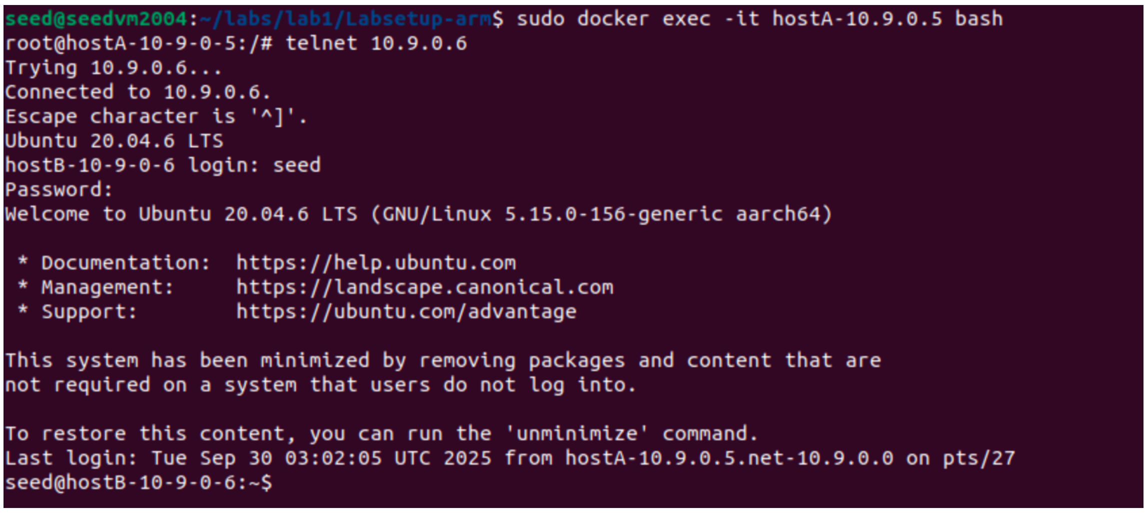
File: ex3.py

Key points:

- BPF filter captures TCP port 23 traffic between client and server.
- Character-by-character capture, reconstructing typed username and password.
- Handles backspaces and stops after both credentials are captured.



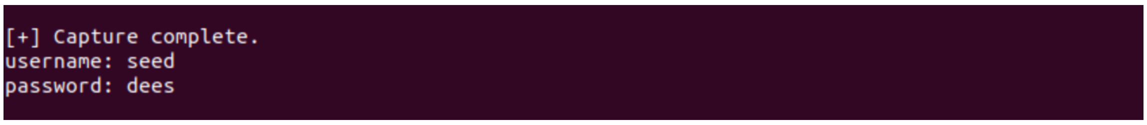
```
seed@seedvm2004:~/labs/lab1/Labsetup-arm$ sudo docker exec -it seed-attacker bash
root@attacker-10-9-0-1:/# python3 -u /root/ex3.py \
>    -i br-318d8411d823 \
>    -c 10.9.0.5 \
>    -s 10.9.0.6
[*] Telnet sniffer started.
    iface=br-318d8411d823  filter='tcp port 23 and (src host 10.9.0.5 and dst host 10.9.0.6)'
    Will stop automatically after capturing username and password.
```



```
seed@seedvm2004:~/labs/lab1/Labsetup-arm$ sudo docker exec -it hostA-10.9.0.5 bash
root@hostA-10-9-0-5:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.6 LTS
hostB-10-9-0-6 login: seed
Password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.0-156-generic aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Tue Sep 30 03:02:05 UTC 2025 from hostA-10.9.0.5.net-10.9.0.0 on pts/27
seed@hostB-10-9-0-6:~$
```

Test Results

Sample capture during Telnet login from Host A to Host B:


```
[+] Capture complete.
username: seed
password: dees
```

The sniffer successfully reconstructed the actual username **seed** and password **dees**.

Questions and Answers

Q1: Why does Telnet echo keystrokes?

- Telnet was designed for early remote terminals that relied on the server to echo characters for display, this is defined in the RFC there is remote and local echo as an option
- This design allows negotiated input handling and feedback. While the server usually handles Echo-ing, it is possible to have the client manage this via local echo mode.

Q2: Should you use Telnet today?

- **No.** Telnet transmits all data in plaintext, making it insecure on modern networks. It also only supports passwords.

Q3: What else can be used instead?

- Use **SSH (Secure Shell)** for remote login, which encrypts traffic, supports public key auth, and also can be used to securely copy files from one machine to another.
- Alternatives like HTTPS-based management interfaces or VPN-secured channels also provide encryption and better authentication management.

Q4: Why are these alternatives better?

- Encryption prevents eavesdropping on credentials.
- Modern tools support additional security such as host verification and MFA.

Exercise 4: ARP Poisoning and MiTM [5 points]

Problem Statement

We performed an ARP poisoning attack to create a Man-in-the-Middle (MiTM) scenario between Host A and Host B. The attacker intercepts Telnet traffic by spoofing ARP replies to trick each host into mapping the other's IP to the attacker's MAC address.

Implementation

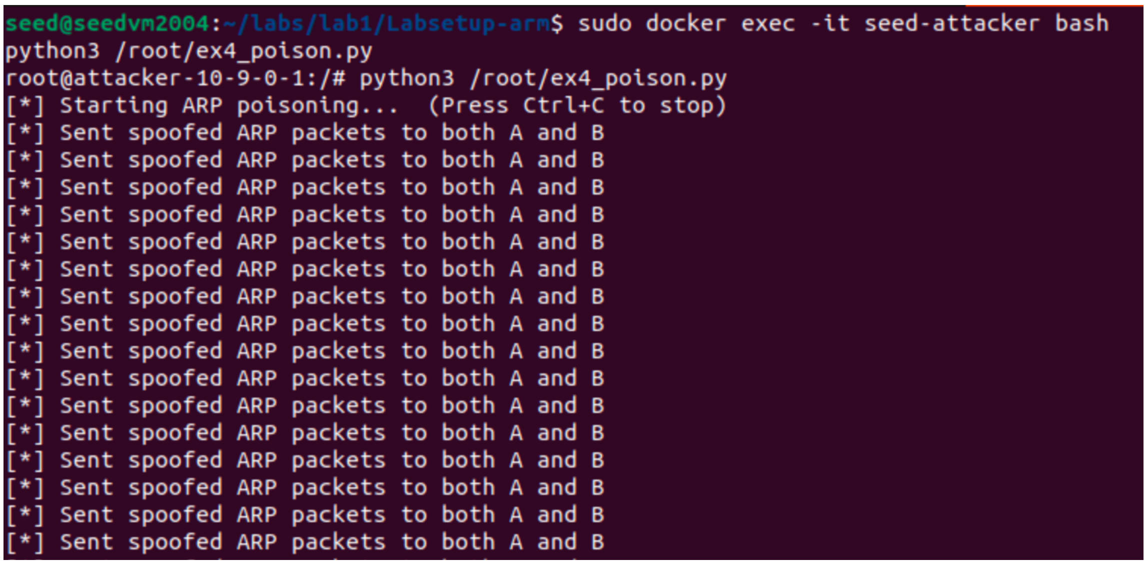
Files:

- **ex4_poison.py** — sends spoofed ARP replies to poison ARP tables on Host A and Host B.
- **ex4_sniffer.py** — reused the sniffer from Exercise 3 to capture Telnet credentials once the attacker is in the middle.

Procedure

1. **Run ARP poisoning on attacker:**

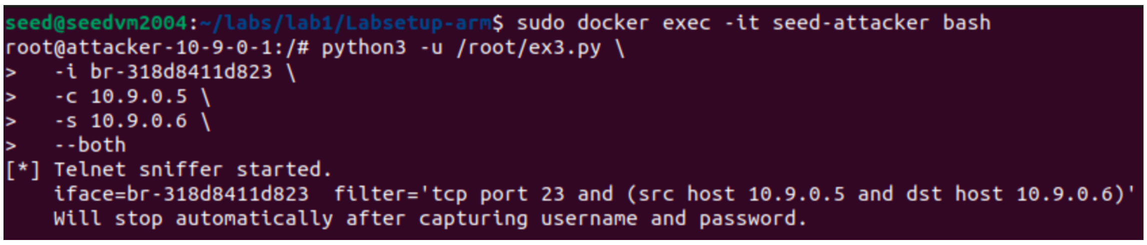
```
sudo docker exec -it seed-attacker bash
python3 /root/ex4_poison.py
```



```
seed@seedvm2004:~/labs/lab1/Labsetup-arm$ sudo docker exec -it seed-attacker bash
python3 /root/ex4_poison.py
root@attacker-10-9-0-1:/# python3 /root/ex4_poison.py
[*] Starting ARP poisoning...  (Press Ctrl+C to stop)
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
[*] Sent spoofed ARP packets to both A and B
```

2. **Run Telnet sniffer on attacker:**

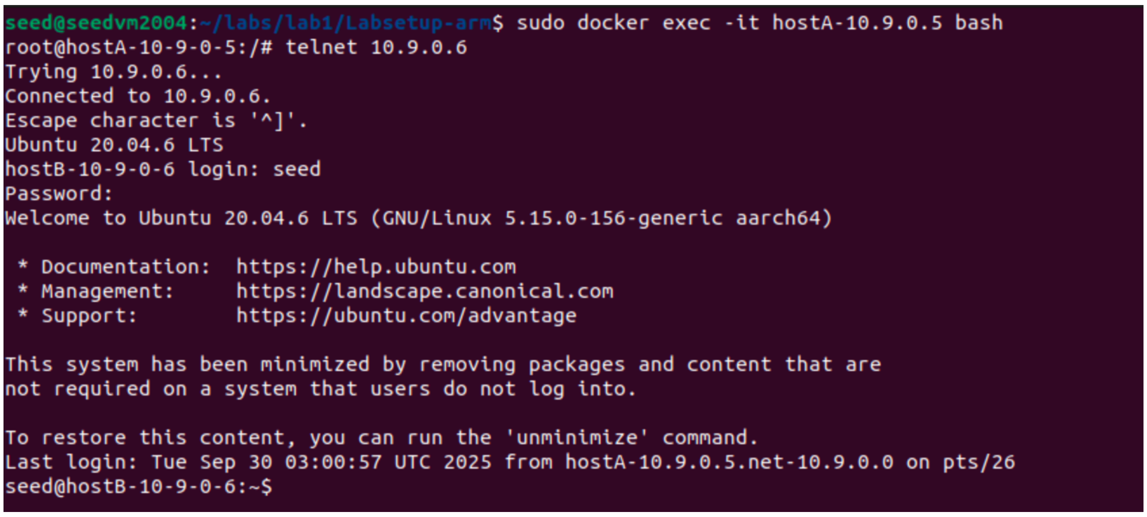
```
sudo docker exec -it seed-attacker bash
python3 -u /root/ex4_sniffer.py -i br-318d8411d823 --client 10.9.0.5 --server 10.9.0.6
```



```
seed@seedvm2004:~/labs/lab1/Labsetup-arm$ sudo docker exec -it seed-attacker bash
root@attacker-10-9-0-1:/# python3 -u /root/ex3.py \
>   -i br-318d8411d823 \
>   -c 10.9.0.5 \
>   -s 10.9.0.6 \
>   --both
[*] Telnet sniffer started.
    iface=br-318d8411d823  filter='tcp port 23 and (src host 10.9.0.5 and dst host 10.9.0.6)'
    Will stop automatically after capturing username and password.
```

3. **Telnet login from Host A to Host B:**

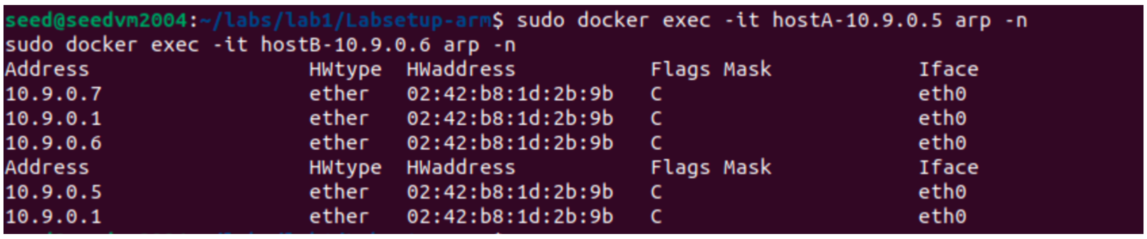
```
sudo docker exec -it hostA-10.9.0.5 telnet 10.9.0.6
# Enter username: seed
# Enter password: dees
```



```
seed@seedvm2004:~/labs/lab1/Labsetup-arm$ sudo docker exec -it hostA-10.9.0.5 bash
root@hostA-10-9-0-5:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.6 LTS
hostB-10-9-0-6 login: seed
Password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.0-156-generic aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Tue Sep 30 03:00:57 UTC 2025 from hostA-10.9.0.5.net-10.9.0.0 on pts/26
seed@hostB-10-9-0-6:~$
```

4. **Verify ARP tables after attack:**

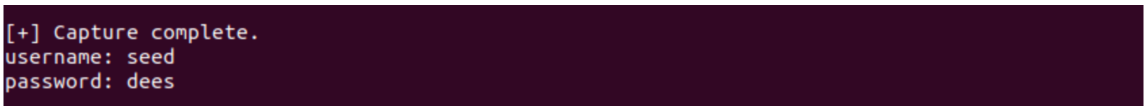
```
sudo docker exec -it hostA-10.9.0.5 arp -n
sudo docker exec -it hostB-10.9.0.6 arp -n
```



```
seed@seedvm2004:~/labs/lab1/Labsetup-arm$ sudo docker exec -it hostA-10.9.0.5 arp -n
sudo docker exec -it hostB-10.9.0.6 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.7                 ether   02:42:b8:1d:2b:9b   C                     eth0
10.9.0.1                 ether   02:42:b8:1d:2b:9b   C                     eth0
10.9.0.6                 ether   02:42:b8:1d:2b:9b   C                     eth0
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.5                 ether   02:42:b8:1d:2b:9b   C                     eth0
10.9.0.1                 ether   02:42:b8:1d:2b:9b   C                     eth0
```

Results

Captured credentials during MiTM attack:



```
[+] Capture complete.
username: seed
password: dees
```

Security Analysis

- The MiTM attack allowed the attacker to intercept traffic despite Host A and Host B being on a switched network.
- This attack exploited ARP's lack of authentication.
- Demonstrates why ARP spoofing tools like **arpspoof** are dangerous in unsecured LANs.

Questions and Answers

Q1: What does the ARP poisoning accomplish?

- Redirects network traffic between the two hosts through the attacker's machine.

Q2: How does this enable Telnet credential theft?

- The attacker can sniff plaintext Telnet packets as they transit through its machine. This is an effective man in the middle attack against an insecure protocol.

Q3: How can this be mitigated?

- Use secure protocols like SSH.
- Deploy ARP inspection and network segmentation.
- Monitor ARP tables for suspicious changes.