CS255 Lab 3 Network Sniffing and Proofing Report

Hugo Wan, twan012

Due Date: 11/16/2025

Lab Environment Setup:

- What I did:

  I navigated to the Lab3setup directory after extracting the Lab 3 starter package (Lab3setup.zip). Inside this directory, I verified that the necessary files (docker-compose.yml and the volumes folder) were present, confirming that the lab environment was ready for setup.

  Inside the Lab3setup folder, I ran the following commands: “dcbuild” and “dcup”. These commands build the Docker images and start the containers for the attacker, Host A, and Host B.

  I used “dockps”, “docksh <container-id>”, and “ifconfig” to enter Host A (hostA-10.9.0.5) and check its network configuration. The <container-id> for hostA is 9a660cf14157.

```
[11/12/25]seed@VM:~$ ls
Desktop    Lab2setup      Lab3setup.zip  Public
Documents  Lab2setup.zip  Music          Templates
Downloads  Lab3setup      Pictures       Videos
[11/12/25]seed@VM:~$ cd Lab3setup/
[11/12/25]seed@VM:~/Lab3setup$ ls
docker-compose.yml  volumes
```

Figure 1 caption: Shows the user accessing Lab3setup/ after extracting the Lab 3 setup package.

```
[11/16/25]seed@VM:~/Lab3setup$ ls
docker-compose.yml  volumes
[11/16/25]seed@VM:~/Lab3setup$ dcbuild
attacker uses an image, skipping
hostA uses an image, skipping
hostB uses an image, skipping
[11/16/25]seed@VM:~/Lab3setup$ dcup
Starting seed-attacker  ... done
Starting hostA-10.9.0.5 ... done
Starting hostB-10.9.0.6 ... done
Attaching to seed-attacker, hostA-10.9.0.5, hostB-10.9.0.6
hostB-10.9.0.6 |  * Starting internet superserver inetd          [ OK ]
hostA-10.9.0.5 |  * Starting internet superserver inetd          [ OK ]
```

Figure 2 caption: Displays running dcbuild then dcup to launch seed-attacker, hostA, and hostB.

```
[11/16/25]seed@VM:~$ dockps
8aa98e623abf  seed-attacker
d19a5d9302f5  hostB-10.9.0.6
9a660cf14157  hostA-10.9.0.5
[11/16/25]seed@VM:~$ docksh 9a660cf14157
root@9a660cf14157:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 27  bytes 3217 (3.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@9a660cf14157:/# exit
```

Figure 3 caption: Shows entering hostA via “docksh” and viewing its network interface “eth0”.

- What I observed:

  The directory correctly contained the Docker Compose file and the shared volumes folder required for running the SEED containers. This matches the expected lab structure described in the manual.

  All three containers started successfully. Output confirms:

  a. seed-attacker — done
  b. hostA-10.9.0.5 — done
  c. hostB-10.9.0.6 — done

  HostA’s eth0 interface is configured with:

  a. IP address: 10.9.0.5
  b. Netmask: 255.255.255.0
  c. Broadcast: 10.9.0.255

- Short conclusion:

  The file structure confirms that the Lab 3 environment was correctly extracted and is ready for container initialization.

  The lab network environment was successfully initialized. The attacker and two hosts are running and accessible for sniffing experiments.

  The container network is functioning correctly. Each host sits on the correct isolated LAN segment, confirming that packet sniffing will only work properly from the attacker VM using the bridged interface.

Task 1.1: Sniffing Packets:

Task 1.1A:

- Objective:

  Use Scapy to sniff ICMP packets on the LAN, observe packet information printed by the callback function, and compare the program's behavior when executed with and without root privilege.

- What I did and observed:

  The following is an important step where I run ifconfig on the attacker VM to locate the Docker bridge interface (br-xxxxx) that connects the VM to all SEED lab containers. Scapy must sniff on this bridge interface because it is the only interface that can see all traffic flowing between Host A and Host B.

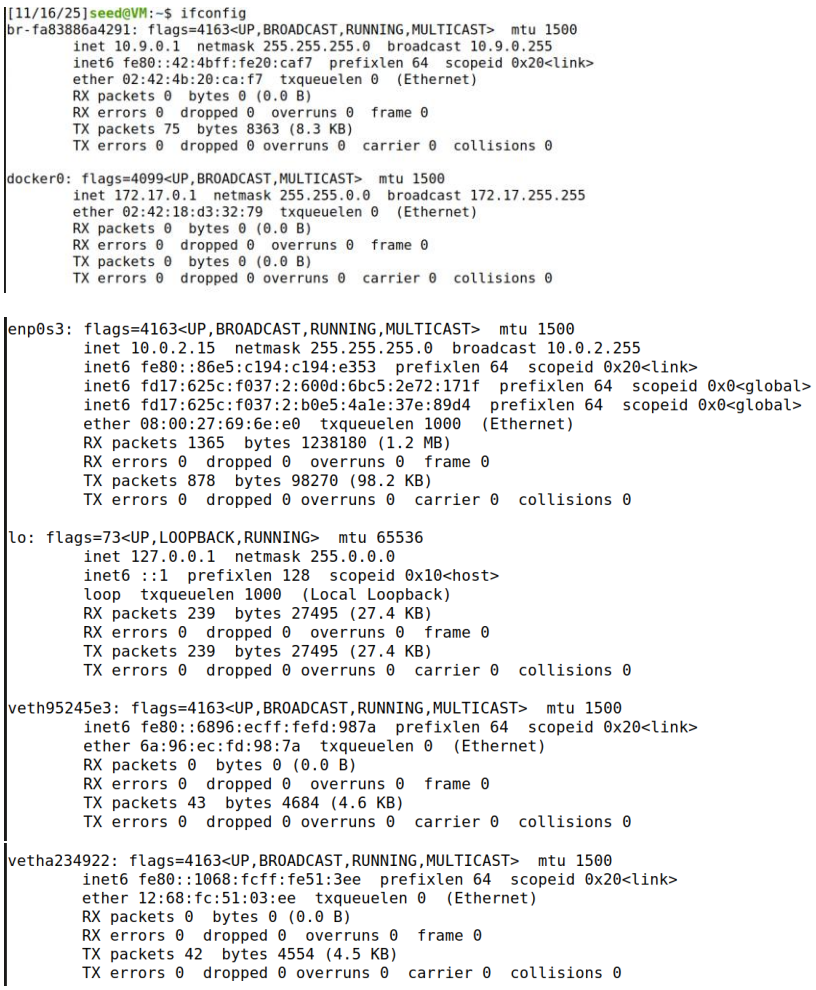

Figure 4, 5, 6 captions: These figures show the attacker VM running ifconfig and identifying the Docker bridge interface br-fa83886a4291, which holds IP address 10.9.0.1. This interface is required for Scapy packet sniffing.

- After finding the bridge interface br-fa83886a4291, navigate to Lab 3 setup folder and open the volumes directory.

```
[11/16/25]seed@VM:~$ ls
Desktop    Downloads  Lab2setup.zip  Lab3setup.zip  Pictures  Templates
Documents  Lab2setup  Lab3setup      Music          Public    Videos
[11/16/25]seed@VM:~$ cd Lab3setup/
[11/16/25]seed@VM:~/Lab3setup$ ls
docker-compose.yml  volumes
[11/16/25]seed@VM:~/Lab3setup$ cd volumes/
[11/16/25]seed@VM:~/.../volumes$ gedit sniffer.py
```

Figure 7 caption: Scapy sniffer script “sniffer.py” is created using “gedit”.

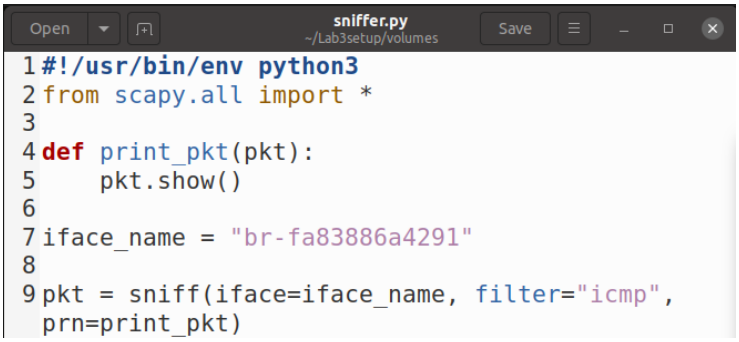

Figure 8 caption: This figure shows the completed Scapy sniffer script with the correct bridge interface (br-fa83886a4291) inserted. The program captures ICMP packets on the SEED lab network and prints their contents using the print_pkt() callback function.

- **Filter: ICMP only**:
- <u>What I did and observed</u>:

```
seed@VM: ~/.../volumes
[11/16/25]seed@VM:~/.../volumes$ ls
sniffer.py
[11/16/25]seed@VM:~/.../volumes$ chmod a+x sniffer.py
[11/16/25]seed@VM:~/.../volumes$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 30685
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xaeaf
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x915e
        id        = 0x24
        seq       = 0x1
###[ Raw ]###
           load      = '\xe8\xe7\x19i\x00\x00\x00\x00
\xa4X\x01\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15
```

Figure 9 caption: This figure shows the execution of “chmod a+x sniffer.py” and “sudo ./sniffer.py” on the attacker VM. The sniffer successfully captures and displays ICMP echo-request packets sent between Host A and Host B, including Ethernet, IP, and ICMP header fields.

```
[11/16/25]seed@VM:~$ dockps
8aa98e623abf  seed-attacker
d19a5d9302f5  hostB-10.9.0.6
9a660cf14157  hostA-10.9.0.5
[11/16/25]seed@VM:~$ docksh 9a660cf14157
root@9a660cf14157:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.488 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.243 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.242 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.227 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.383 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.186 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.139 ms
64 bytes from 10.9.0.6: icmp_seq=8 ttl=64 time=0.498 ms
64 bytes from 10.9.0.6: icmp_seq=9 ttl=64 time=0.134 ms
```

Figure 10 caption: This figure shows Host A (10.9.0.5) pinging Host B (10.9.0.6) to generate ICMP packets. These packets trigger Scapy’s callback function and allow the sniffer to capture live LAN traffic.

- Sniffer failing **without root** privilege: I switched from the root shell back to the normal seed user account. I attempted to run the sniffer without using sudo:

```
[11/16/25]seed@VM:~/.../volumes$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 9, in <module>
    pkt = sniff(iface=iface_name, filter="icmp", prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 906, in _run
    sniff_sockets[L2socket(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

Figure 11 caption: This figure shows the result of running “./sniffer.py” (without sudo in front) as a regular user. Scapy raises a “PermissionError: [Errno 1] Operation not permitted” because creating raw packet sockets requires root privileges, confirming that non-root users cannot perform packet sniffing.

- Short conclusion:
- Running the sniffer without root privilege fails because capturing raw packets requires elevated permissions. The operating system blocks normal users from placing the interface into promiscuous mode or accessing low-level packet data for security reasons. This confirms that root privileges are mandatory for packet sniffing using Scapy.

Task 1.1B:

- What I did and observed:

  In this task, I tested several BPF filters using Scapy's "sniff()" function. Each filter successfully limited the captured traffic to the desired protocols, hosts, and subnets. This demonstrates the flexibility of packet filtering and how BPF expressions can isolate specific network events for analysis.

- The ICMP-only filter requested in Task 1.1B was already completed in Task 1.1A when verifying the sniffer program. Therefore, only the TCP/port 23 and subnet filters were tested in this section.

- **Filter: Capture any TCP packet that comes from a particular IP and with a destination port number 23:**

```
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

iface_name = "br-fa83886a4291"

pkt = sniff(iface=iface_name,
            filter="tcp and src host 10.9.0.5 and dst port 23",
            prn=print_pkt)

```

Figure 12 caption: This figure shows the sniffer script updated with a BPF expression that captures only TCP packets originating from Host A (10.9.0.5) and destined for port 23.

- When I applied the filter "tcp and src host 10.9.0.5 and dst port 23" and generated ICMP traffic using ping, the sniffer displayed no packets. This is expected because ICMP packets do not match the TCP filter, and therefore Scapy correctly ignored them.

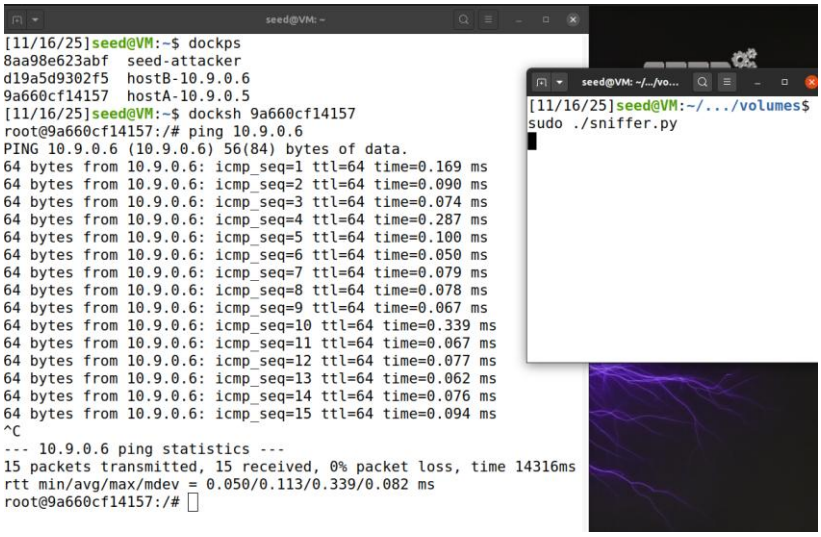

Figure 13 caption: This figure shows the sniffer running with the TCP/port 23 filter while Host A sends ICMP echo requests to Host B. Because ICMP traffic does not match the TCP filter, the sniffer correctly produces no output, demonstrating that the filter excludes unrelated packets.

- To generate the actual TCP traffic, I ran “telent” instead of “ping”. This will produce TCP packets on port 23, and how my sniffer captures them.

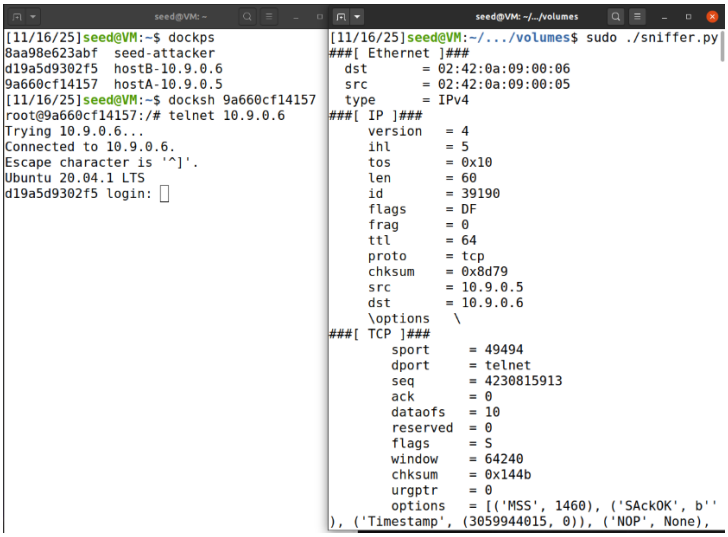

Figure 14 caption: This figure shows Host A initiating a Telnet connection to Host B (10.9.0.6) on TCP port 23, which generates packets that match the filter tcp and src host 10.9.0.5 and dst port 23. The sniffer successfully captures the TCP SYN packet, displaying Ethernet, IP, and TCP headers, confirming that the filter works as intended.

- Short conclusion:

  The lack of output confirms that the filter is functioning properly and only captures packets that meet all specified conditions. This demonstrates that Scapy’s BPF filtering works correctly by excluding unrelated traffic.

- **Filter: Subnet Filter (net 128.230.0.0/16):**
- What I did:

  This filter only captures packets from or to the subnet 128.230.0.0/16.

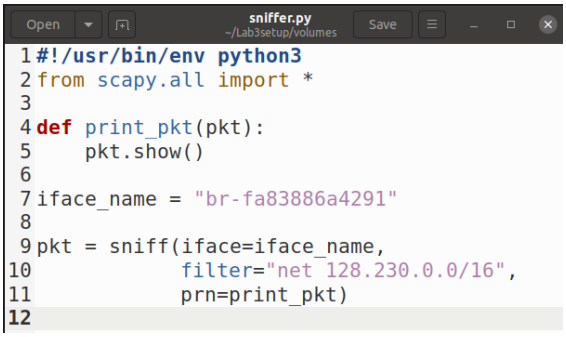

Figure 15 caption: This figure shows the sniffer script updated with a BPF subnet filter that captures only packets whose source or destination IP falls within the “128.230.0.0/16” network.

- The following figure confirms that the subnet filter correctly matches packets whose destination IP belongs to the specified network.

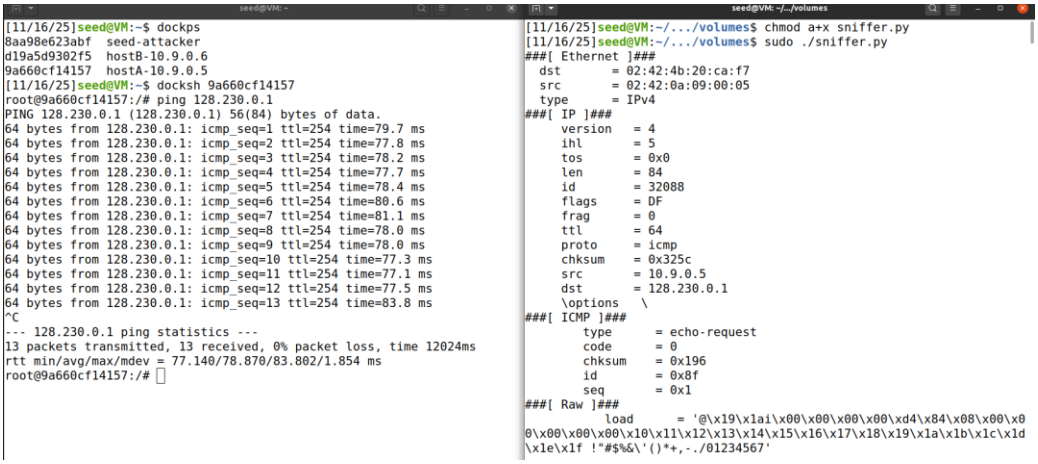

Figure 16 caption: This figure shows Host A sending ICMP echo requests to “128.230.0.1”, which generates traffic destined for the chosen subnet. The sniffer, running with the filter net “128.230.0.0/16”, successfully captures the outbound ICMP packet, displaying Ethernet, IP, and ICMP headers.

- What I observed:

  The sniffer successfully captured an outbound ICMP echo-request packet from Host A (10.9.0.5) to the destination IP 128.230.0.1.

  The captured packet included: Ethernet header, IP header with “dst = 128.230.0.1”, ICMP header (type = echo-request), and Raw payload. Only packets that matched the subnet filter were displayed.

  No internal 10.9.0.x packets or telnet traffic appeared, confirming the filter excluded unrelated traffic.

- Short conclusion:

  The subnet filter worked correctly, capturing only packets whose destination IP belonged to the specified “128.230.0.0/16” network. This demonstrates that Scapy’s BPF filtering can selectively monitor traffic based on entire IP ranges, not just individual hosts or ports. The outbound ping test verified that even unreachable destinations can produce valid packets for subnet-based filtering.

Task 1.2: Spoofing ICMP Packets:

- What I did:

  I created a new script called “icmp_spoof.py” inside the “~/Lab3setup/volumes/” directory. The script constructs an IP packet with a spoofed source IP address (1.2.3.4) and a destination of Host B (10.9.0.6). I added an ICMP Echo Request layer and sent the packet using Scapy’s “send()” function.

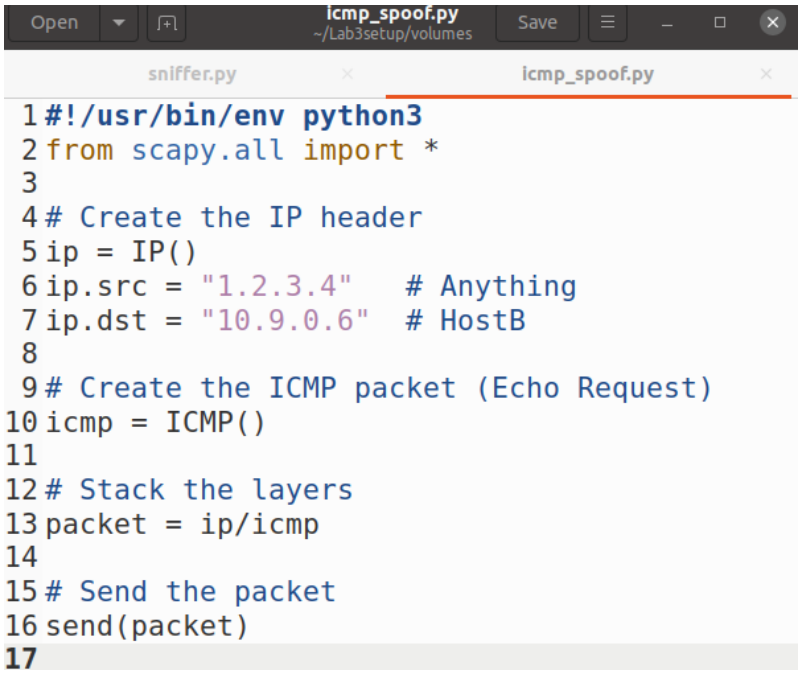

  Figure 17 caption: Shows the Scapy script used to construct a spoofed ICMP Echo Request packet. The source IP is intentionally forged (1.2.3.4), and the destination is set to Host B (10.9.0.6). The packet is built using stacked IP and ICMP layers, then transmitted using Scapy’s “send()” function.

- Then, I made the script executable by using “chmod a+x icmp_spoof.py”, and I execute it with root privileges by entering “sudo ./icmp_spoof.py”. As a result, Scapy confirmed “Sent 1 packets.”

```
[11/16/25]seed@VM:~/.../volumes$ ls
icmp_spoof.py  sniffer.py
[11/16/25]seed@VM:~/.../volumes$ chmod a+x icmp_spoof.py
[11/16/25]seed@VM:~/.../volumes$ sudo ./icmp_spoof.py
.
Sent 1 packets.
```

  Figure 18 caption: This figure shows the spoofing script being made executable and run with root privileges. Scapy confirms that one spoofed ICMP packet was successfully sent.

- In a separate terminal, I made the script executable and ran it with my sniffer (sniffer.py) with the filter set to "icmp" by entering “sudo ./sniffer.py”.

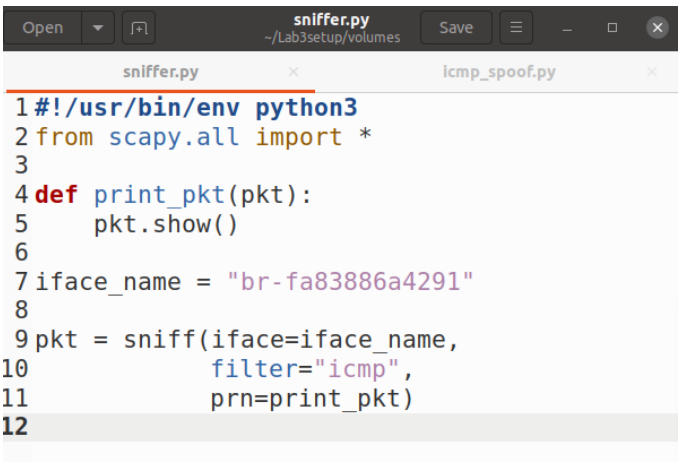

Figure 19 caption: This figure shows the “sniffer.py” script updated to use the filter “icmp”, ensuring that ICMP packets, including the spoofed Echo Request, are captured and displayed.

- What I observed:

  From the sniffer output (Figure 20), I observed that the packet used ICMP protocol, the source IP address was the spoofed value (1.2.3.4), and the destination IP was Host B (10.9.0.6). The packet type was an ICMP Echo Request (type = echo-request). The fact that the sniffer captured this packet confirms that the spoofed ICMP packet was successfully placed on the LAN. Importantly, no ICMP Echo Reply was observed in the sniffer output.

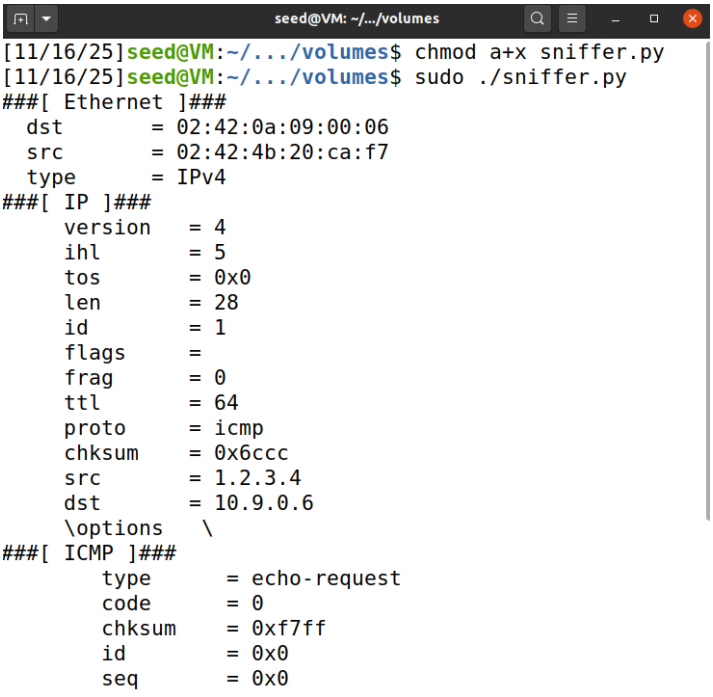

Figure 20 caption: This figure shows sniffer.py running with an ICMP filter while the spoofing script is executed in a separate terminal. The sniffer successfully captures the forged ICMP Echo Request packet, displaying the spoofed source IP (1.2.3.4) and the destination Host B (10.9.0.6). This confirms that the spoofed packet was transmitted onto the network and detected by the local sniffer.

- Short conclusion:

  In Task 1.2, I successfully crafted and transmitted a spoofed ICMP Echo Request packet using Scapy. The sniffer captured the packet exactly as intended, showing the forged source IP address (1.2.3.4). Host B received the packet, but since the reply would be sent to the spoofed address instead of my machine, I did not see any ICMP Echo Reply in my capture. This demonstrates how IP spoofing manipulates packet headers and why spoofed packets cannot receive responses unless the attacker controls the spoofed address.

Task 1.3: Traceroute:

- What I did:

  I created a new Python file "traceroute.py" (Figure 21) inside the "Lab3setup/volumes" directory. I wrote a traceroute implementation using Scapy:

  a. Target = 10.9.0.6 (Host B)
  b. TTL values from 1 to 30
  c. Sent ICMP Echo Requests with each TTL
  d. Printed the router (or host) that returned an ICMP reply

  Furthermore, I started sniffer.py in one terminal to capture ICMP traffic. In another terminal, I ran "sudo ./traceroute.py" (Figure 22). The sniffer displayed each outgoing ICMP packet and confirmed the TTL values.

Open traceroute.py ~/Lab3setup/volumes Save

sniffer.py | icmp_spoof.py | traceroute.py

```python
#!/usr/bin/env python3
from scapy.all import *

target = "10.9.0.6"       # Host B
max_hops = 30             # typical traceroute limit

for ttl in range(1, max_hops + 1):
    ip = IP(dst=target, ttl=ttl)
    icmp = ICMP()

    print(f"Sending packet with TTL = {ttl}")

    # Send packet and wait for reply
    reply = sr1(ip/icmp, timeout=2, verbose=0)

    if reply is None:
        print(f"{ttl}: Request timed out")
        continue

    # Print the hop IP address
    print(f"{ttl}: reply from {reply.src}")

    # If the destination replies directly, stop the traceroute
    if reply.type == 0:  # Echo Reply
        print("Destination reached.")
        break
```

Figure 21 caption: This figure shows the Python script used to implement a simplified traceroute. The script sends ICMP Echo Requests with increasing TTL values, listens for ICMP replies, and prints the hop address for each TTL.

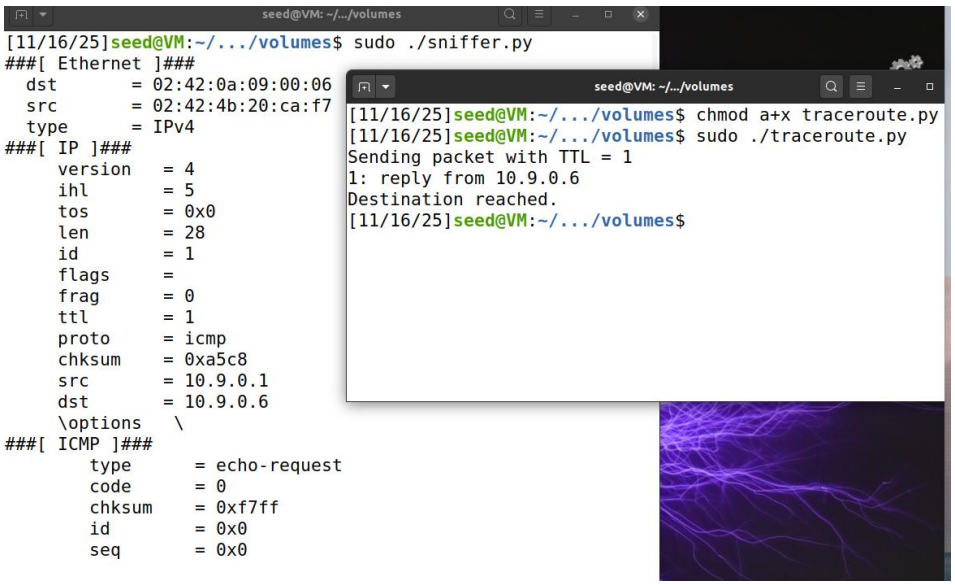

Figure 22 caption: This figure shows the sniffer running on br-fa83886a4291 and displaying ICMP packets with TTL = 1. On the right, the traceroute script is executed, sending packets with increasing TTL values. The destination (10.9.0.6) replies immediately because it is only one hop away.

- What I observed:

  In figure 22's left terminal window, the sniffer showed packets with:

  a. TTL = 1
  b. Source IP = 10.9.0.1 (hostA)
  c. Destination IP = 10.9.0.6 (hostB)
  d. ICMP type = echo-request

  In figure 22's right terminal window, the traceroute script printed:

  "Sending packet with TTL = 1"
  "1: reply from 10.9.0.6"
  "Destination reached."

  These tell me that HostA and HostB are directly connected on the same Docker network. Therefore, the first hop is already the destination.

- Short conclusion:

  In Task 1.3, I successfully implemented a basic traceroute tool using Scapy. By incrementing the TTL value and observing the ICMP replies, I confirmed that HostB (10.9.0.6) is only one hop away from HostA. The sniffer output matched the traceroute results, showing ICMP packets with TTL = 1 destined for HostB. This demonstrates how traceroute identifies each hop by leveraging ICMP Time Exceeded and Echo Reply messages.

Task 1.4: Sniffing and-then Spoofing:

- What I did:

  I wrote a sniff-and-spoof script using Scapy (Figure 23, sniff_and_spoof.py). The script listens on the attacker VM's interface (br-xxxx) and checks for ICMP Echo Requests (type = 8). When an Echo Request appears, the script immediately constructs an ICMP Echo Reply with fields swapped: "src = request.dst", "dst = request.src", and "type = 0" (Echo Reply). I then sent pings from Host A (10.9.0.5) to the three required destinations where "ping 1.2.3.4" for a non-existing host on the Internet (Figure 24), "ping 10.9.0.99" for a non-existing host on the LAN (Figure 25), and "ping 8.8.8.8" for an existing host on the Internet (Figure 26). I captured both the ping results and the attacker's spoof output as evidence.

Open sniff_and_spoof.py ~/Lab3setup/volumes Save

sniffer.py | icmp_spoof.py | traceroute.py | sniff_and_spoof.py

```python
#!/usr/bin/env python3
from scapy.all import *

iface_name = "br-fa83886a4291"

def spoof_reply(pkt):

    # Only handle ICMP Echo Requests (type 8)
    if ICMP in pkt and pkt[ICMP].type == 8:
        print(f"[+] Captured ICMP Echo Request from {pkt[IP].src} to {pkt[IP].dst}")

        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)

        reply = ip/icmp
        send(reply, verbose=0)

        print(f"[+] Spoofed Reply Sent: {pkt[IP].dst} → {pkt[IP].src}")

print("[*] Sniffing for ICMP Echo Requests...")
sniff(iface=iface_name, filter="icmp", prn=spoof_reply)
```

Figure 23 caption: The sniff_and_spoof.py script shows how ICMP Echo Requests are detected and instantly forged into spoofed Echo Reply packets using the victim's source/destination fields.

- What I observed:

1. Pinging 1.2.3.4 (Nonexistent External Host):

   Host A received only the spoofed replies, not real ones since 1.2.3.4 doesn't exist. Ping output showed success, even though the host is unreachable.

seed@VM: ~/.../volumes

```
[11/16/25]seed@VM:~/.../volumes$ dockps
8aa98e623abf  seed-attacker
d19a5d9302f5  hostB-10.9.0.6
9a660cf14157  hostA-10.9.0.5
[11/16/25]seed@VM:~/.../volumes$ docksh 9a660cf14157
root@9a660cf14157:/# ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
^C
--- 1.2.3.4 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 200
3ms

root@9a660cf14157:/#
```

seed@VM: ~/.../volumes

```
[11/16/25]seed@VM:~/.../volumes$ sudo ./sniff_and_spoof.py
[*] Sniffing for ICMP Echo Requests...
[+] Captured ICMP Echo Request from 10.9.0.5 to 1.2.3.4
[+] Spoofed Reply Sent: 1.2.3.4 → 10.9.0.5
[+] Captured ICMP Echo Request from 10.9.0.5 to 1.2.3.4
[+] Spoofed Reply Sent: 1.2.3.4 → 10.9.0.5
[+] Captured ICMP Echo Request from 10.9.0.5 to 1.2.3.4
[+] Spoofed Reply Sent: 1.2.3.4 → 10.9.0.5
```

Figure 24 caption: Ping to nonexistent external IP 1.2.3.4. Normally unreachable (no ARP resolution), but the attacker still captures Host A's ICMP request and returns spoofed Echo Replies, making it appear alive.

2. Pinging 10.9.0.99 (Nonexistent LAN Host):

   Host A received “Destination Host Unreachable” errors from router 10.9.0.5. At the same time, my spoof program generated fake Echo Replies, even though 10.9.0.99 does not exist.

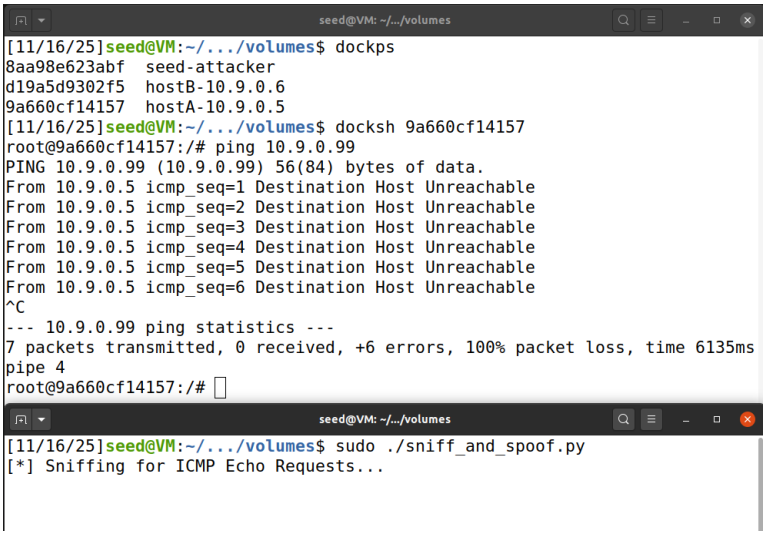

Figure 25 caption: Ping to a nonexistent LAN host (10.9.0.99). Host A receives “Destination Host Unreachable” from router 10.9.0.5, but simultaneously the attacker intercepts the Echo Request and sends spoofed Echo Replies pretending 10.9.0.99 is alive.

3. Pinging 8.8.8.8 (Existing Internet Host):

   Host A received both real and spoofed replies. This produced duplicate ICMP responses. The attacker printed captured packets and spoofed replies.

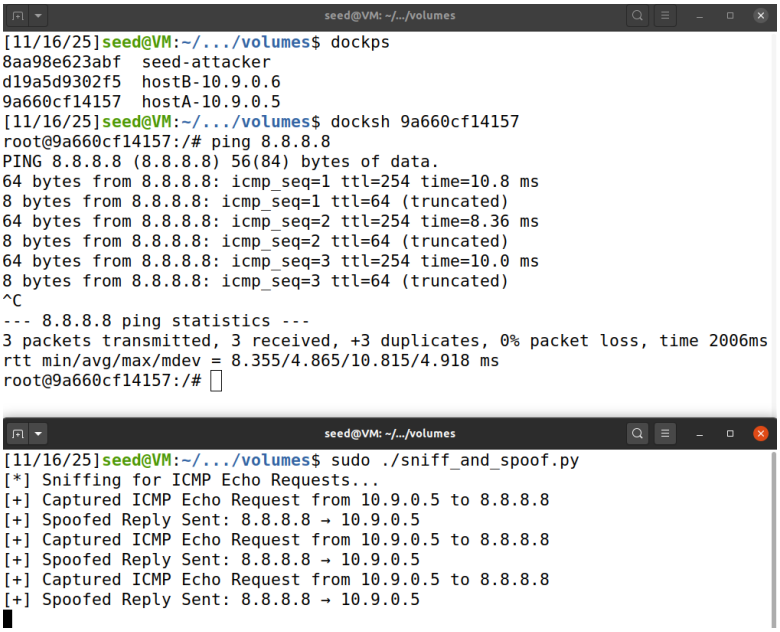

Figure 26 caption: Ping to real host 8.8.8.8 from Host A, while the attacker VM captures each ICMP Echo Request and injects spoofed Echo Replies. Both real replies and spoofed replies appear, causing duplicates.

- Short conclusion:

  In Task 1.4, I did my sniff-and-spoof program correctly as it detected all ICMP Echo Requests on the LAN, sent forged Echo Replies regardless of whether the target host exists, made unreachable hosts appear to be alive, and produced duplicate replies when pinging an actual reachable host (8.8.8.8). This demonstrates how an attacker on the same LAN can manipulate ping results by injecting fake packets, exploiting how ICMP and ARP behave.