Report

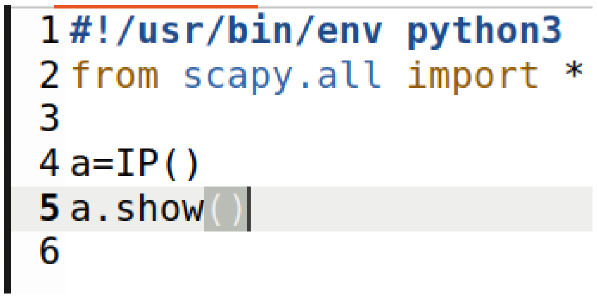
Network Security Assignment

CSG513

Submitted to:
Dr Ashutosh Bhatia

Submitted BY:
Ankit Kumar(2023H1030076P)
Sonu Parmanik(2023H1030097P)

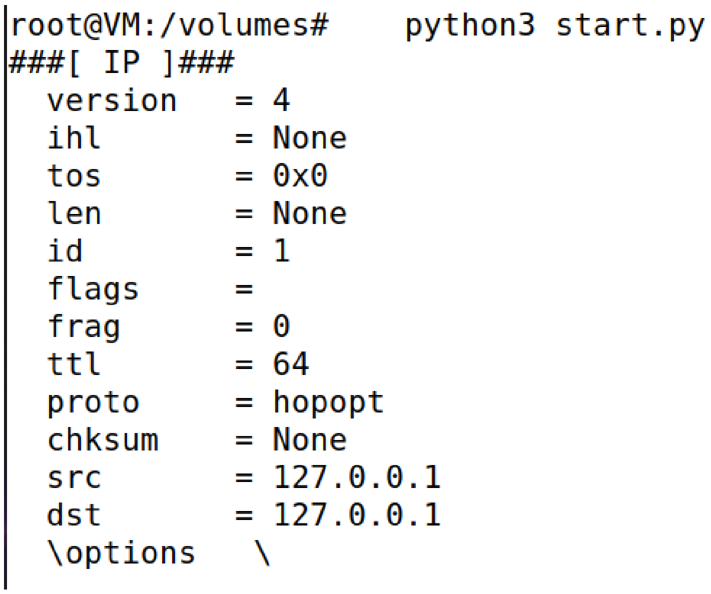
Assignment # 1
Network Security
Packet Sniffing and Spoofing



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

a=IP()
a.show()

```

IP() return a instances of iP class

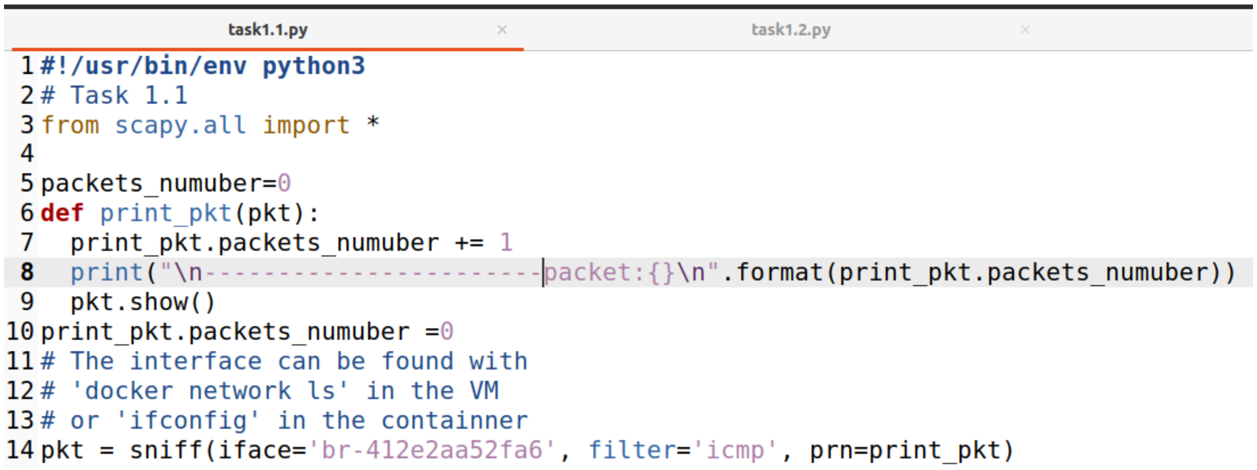
During running this script it create an Empty IP packet object than using .show() it display the content of IP() packet



```
root@VM:/volumes#    python3 start.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   \
```

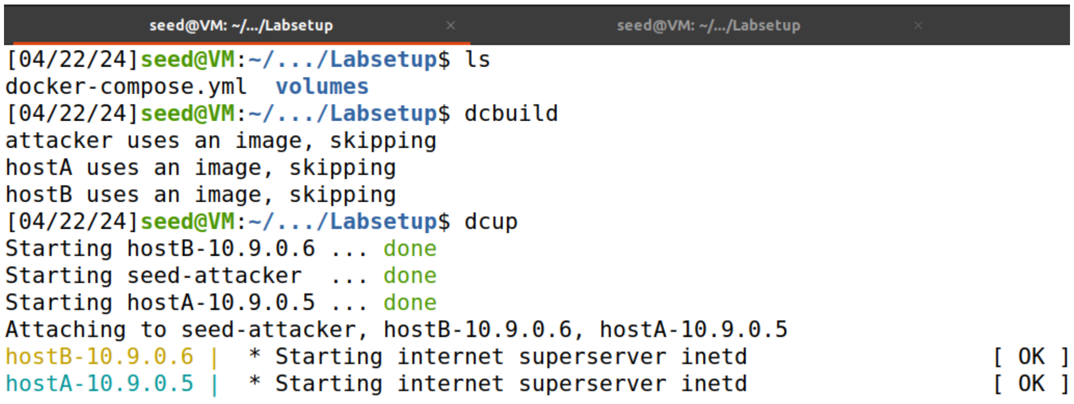
TAsk 1.1A:

Task 1.1A. In the above program, for each captured packet, the callback function print pkt() will be invoked; this function will print out some of the information about the packet. Run the program with the root privilege and demonstrate that you can indeed capture packets. After that, run the program again, but without using the root privilege; describe and explain your observations.

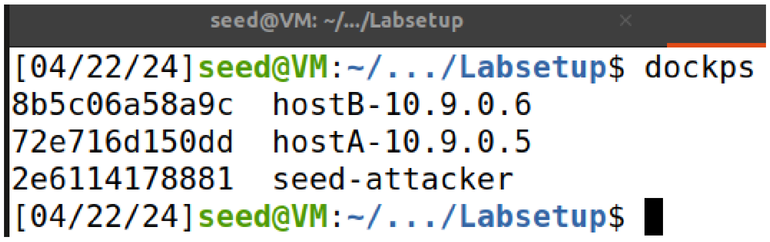

task1.1.py task1.2.py

```
#!/usr/bin/env python3
# Task 1.1
from scapy.all import *

packets_numuber=0
def print_pkt(pkt):
  print_pkt.packets_numuber += 1
  print("\n----------------------packet:{}\n".format(print_pkt.packets_numuber))
  pkt.show()
print_pkt.packets_numuber =0
# The interface can be found with
# 'docker network ls' in the VM
# or 'ifconfig' in the containner
pkt = sniff(iface='br-412e2aa52fa6', filter='icmp', prn=print_pkt)
```

Output:


seed@VM: ~/.../Labsetup seed@VM: ~/.../Labsetup

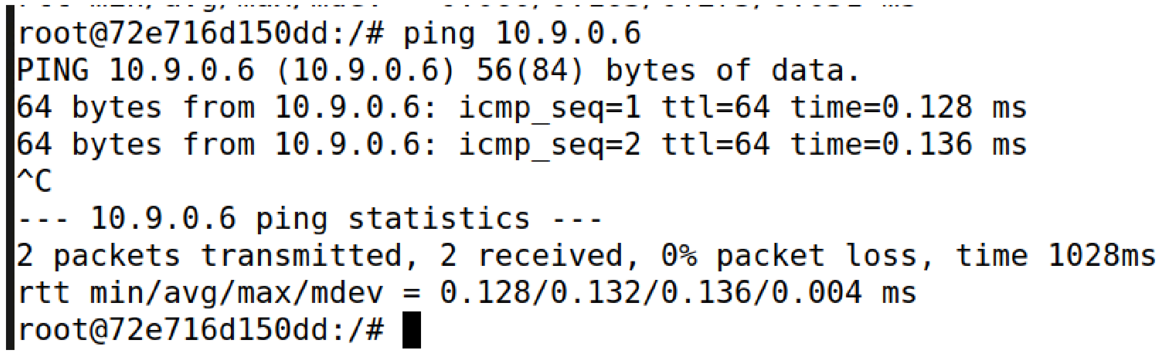
```
[04/22/24]seed@VM:~/.../Labsetup$ ls
docker-compose.yml  volumes
[04/22/24]seed@VM:~/.../Labsetup$ dcbuild
attacker uses an image, skipping
hostA uses an image, skipping
hostB uses an image, skipping
[04/22/24]seed@VM:~/.../Labsetup$ dcup
Starting hostB-10.9.0.6 ... done
Starting seed-attacker  ... done
Starting hostA-10.9.0.5 ... done
Attaching to seed-attacker, hostB-10.9.0.6, hostA-10.9.0.5
hostB-10.9.0.6 |  * Starting internet superserver inetd                    [ OK ]
hostA-10.9.0.5 |  * Starting internet superserver inetd                    [ OK ]
```


seed@VM: ~/.../Labsetup

```
[04/22/24]seed@VM:~/.../Labsetup$ dockps
8b5c06a58a9c  hostB-10.9.0.6
72e716d150dd  hostA-10.9.0.5
2e6114178881  seed-attacker
[04/22/24]seed@VM:~/.../Labsetup$
```

From A ping to B:

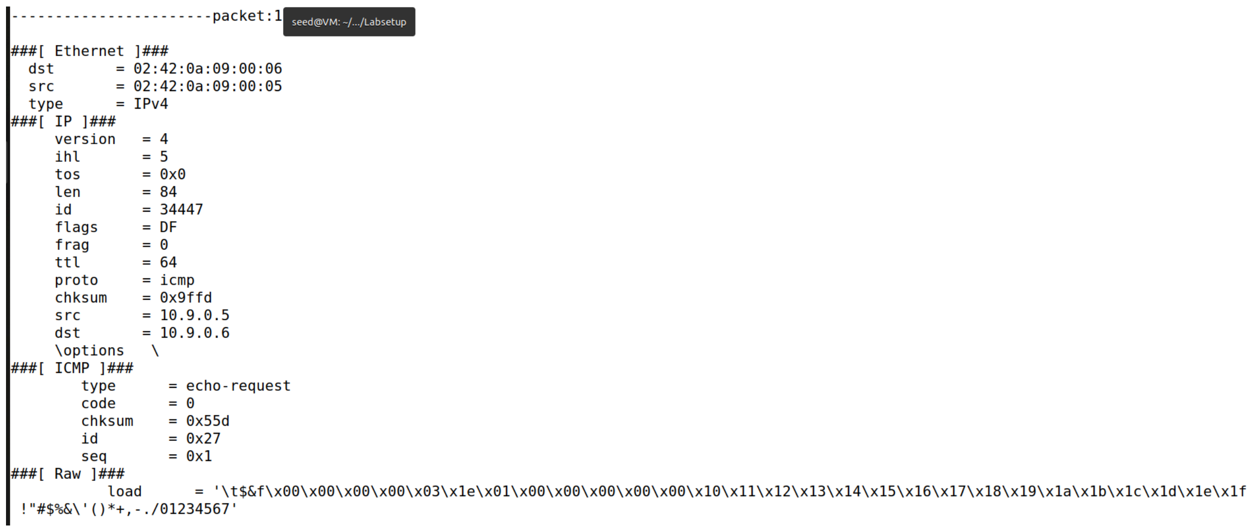
Ping for two packet only then interrupt for stop



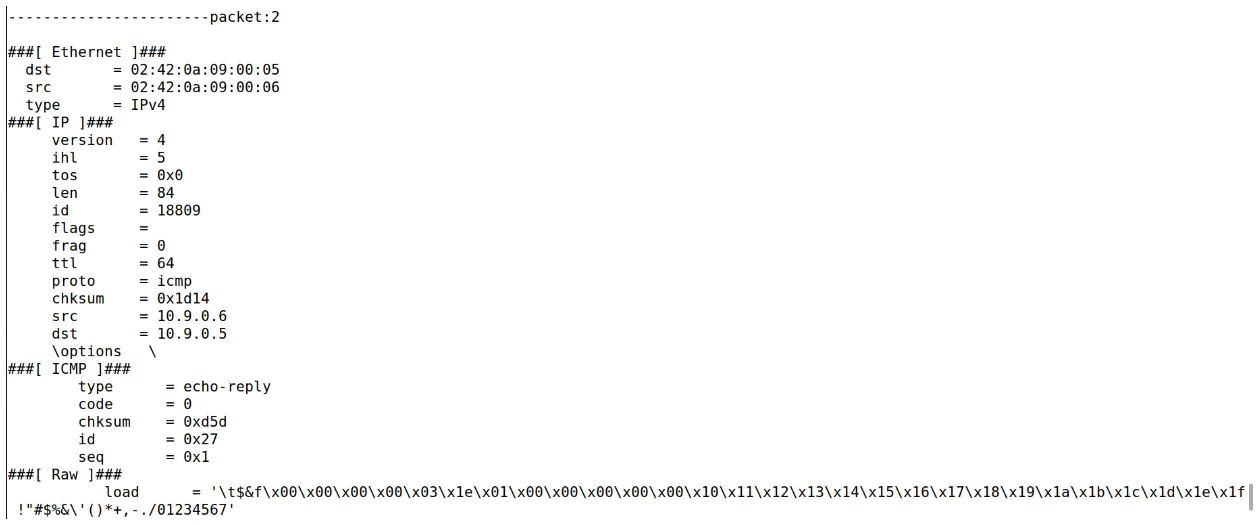
```
root@72e716d150dd:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.128 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.136 ms
^C
--- 10.9.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1028ms
rtt min/avg/max/mdev = 0.128/0.132/0.136/0.004 ms
root@72e716d150dd:/# 
```

Seed_attacker interface:

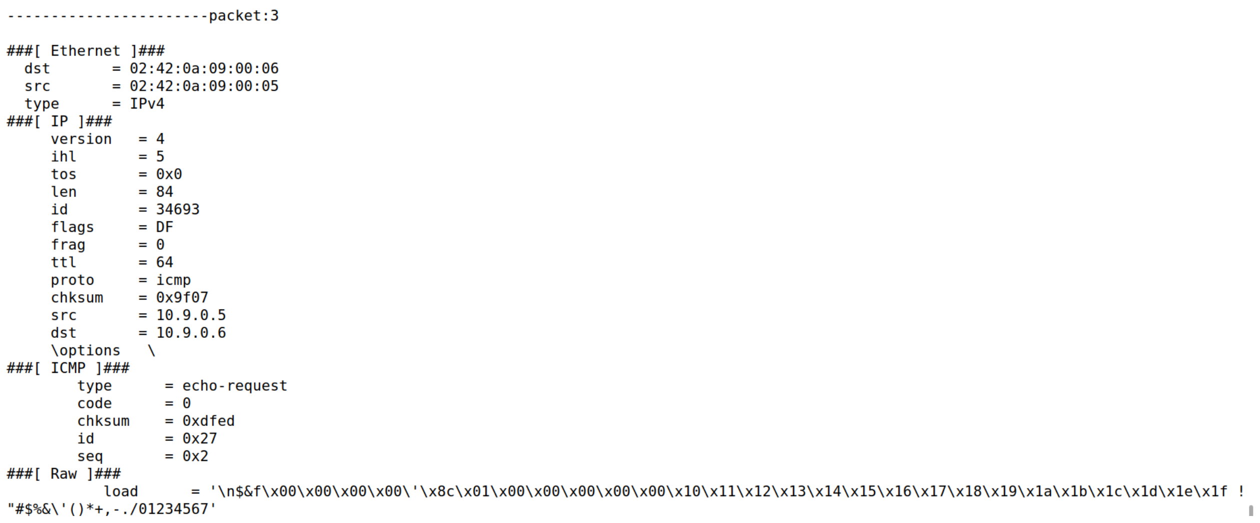
After the ping the packet sniffing is printed



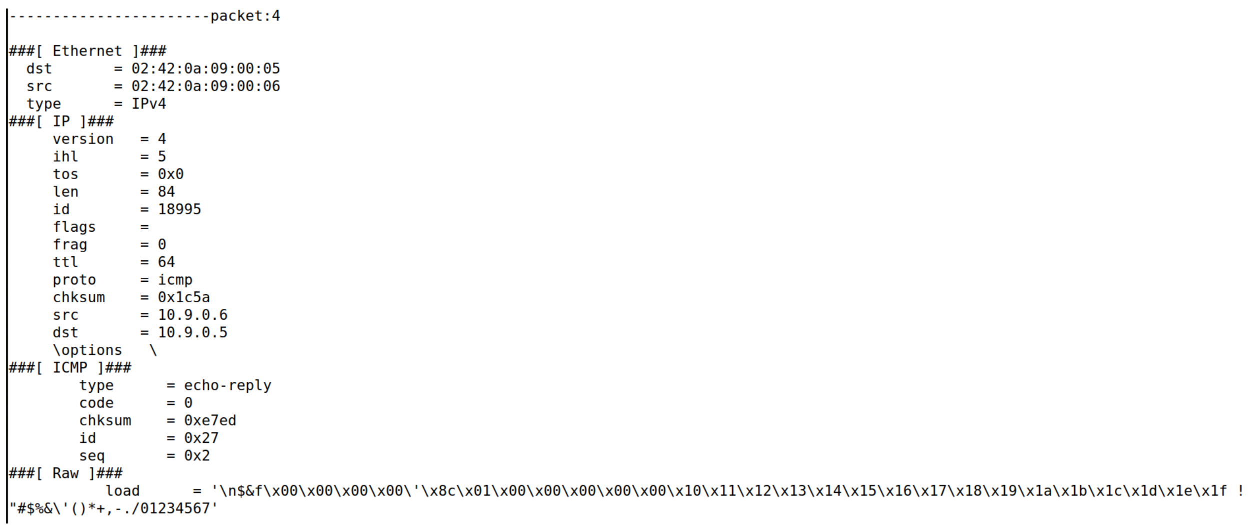
```
----------------------packet:1
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 34447
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x9ffd
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x55d
        id        = 0x27
        seq       = 0x1
###[ Raw ]###
           load      = '\t$&f\x00\x00\x00\x00\x03\x1e\x01\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
 !"#$%&\'()*+,-./01234567'
```



```
-----------------------packet:2

###[ 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        = 18809
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x1d14
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xd5d
        id        = 0x27
        seq       = 0x1
###[ Raw ]###
           load      = '\t$&f\x00\x00\x00\x00\x03\x1e\x01\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
 !"#$%&\'()*+,-./01234567'
```



```
-----------------------packet:3

###[ 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        = 34693
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x9f07
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xdfed
        id        = 0x27
        seq       = 0x2
###[ Raw ]###
           load      = '\n$&f\x00\x00\x00\x00\'\x8c\x01\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !
"#$%&\'()*+,-./01234567'
```



```
-----------------------packet:4

###[ 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        = 18995
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x1c5a
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xe7ed
        id        = 0x27
        seq       = 0x2
###[ Raw ]###
           load      = '\n$&f\x00\x00\x00\x00\'\x8c\x01\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !
"#$%&\'()*+,-./01234567'
```

'Br-412e2aa52fa6' using command **ifconfig**
To run the above program :

In the seed-attacker terminal, run the **python3 task1.1.py**
Then, two different **hostA and hostB** in different tab in the terminal using the command:
docksh hostA 10.9.0.5
docksh hostB 10.9.0.6

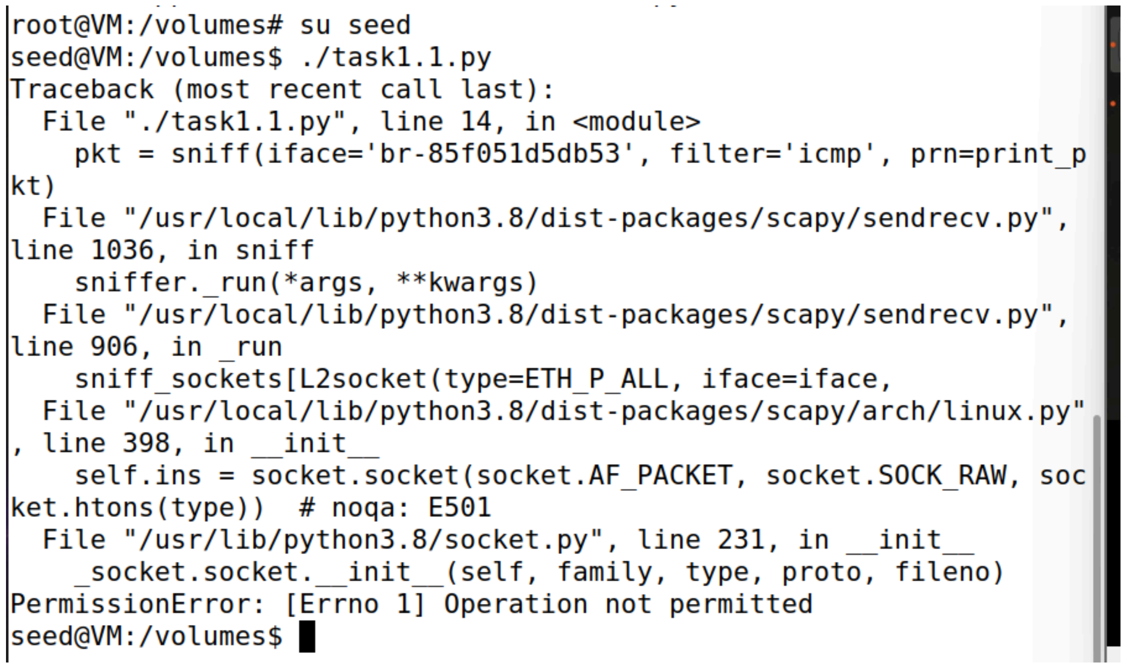
Then ping the packet from **hostA to hostB** using the command **ping 10.9.0.6**
While running the program, it captures the ICMP packet as assigned in the filter on the given inteface='br-412e2aa52fa6', and using the **.show()**, it prints the detail of the packet

Also, to sniff on multiple interfaces put all interfaces in the list and pass the argument in the above **sniff.**
Eg:: interfaces = ['br-412e2aa52fa6', 'eth0', 'eth1']

pkt= sniff(interfaces, filter='icmp', prn=print_pkt)

PermissionError in below figure:

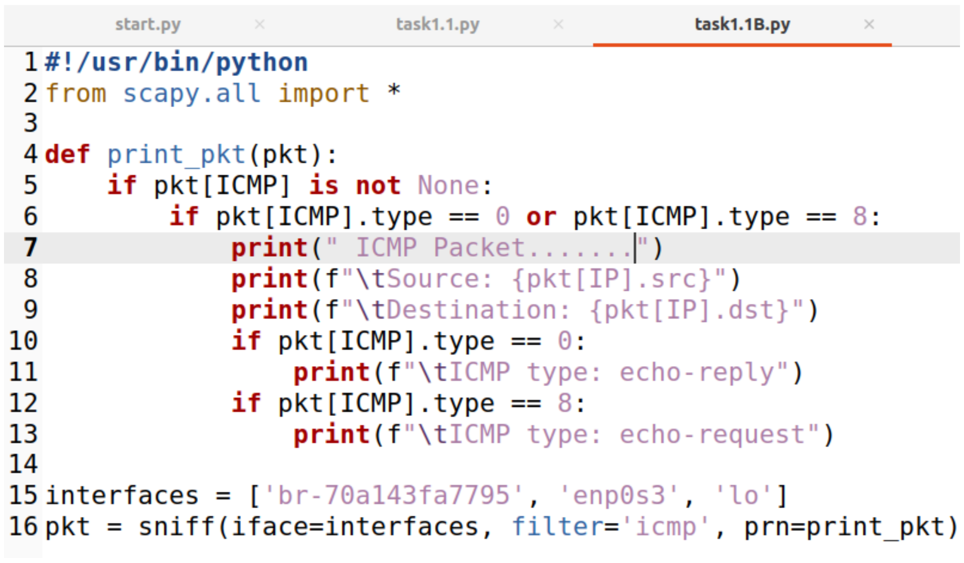
Using sudo to run the program it permit us to see complete network traffick on our given interface . we can see it in below program we got **permission error .** So if we want to sniff packets we need root privilege to see the traffic and capture the relevant packets



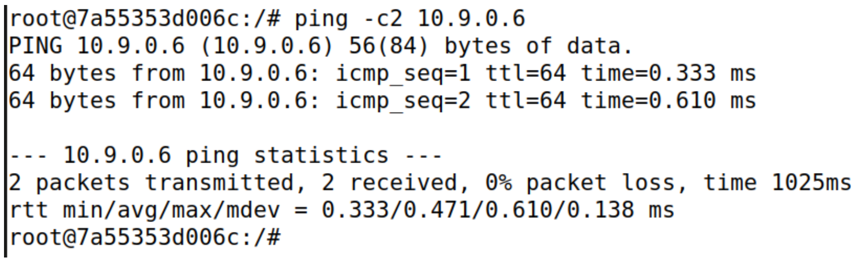
```
root@VM:/volumes# su seed
seed@VM:/volumes$ ./task1.1.py
Traceback (most recent call last):
  File "./task1.1.py", line 14, in <module>
    pkt = sniff(iface='br-85f051d5db53', filter='icmp', prn=print_p
kt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py",
line 1036, in sniff
    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, soc
ket.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. Usually, when we sniff packets, we are only interested certain types of packets. We can do that by setting filters in sniffing. Scapy's filter use the BPF (Berkeley Packet Filter) syntax; you can find the BPF manual from the Internet. Please set the following filters and demonstrate your sniffer program again (each filter should be set separately):

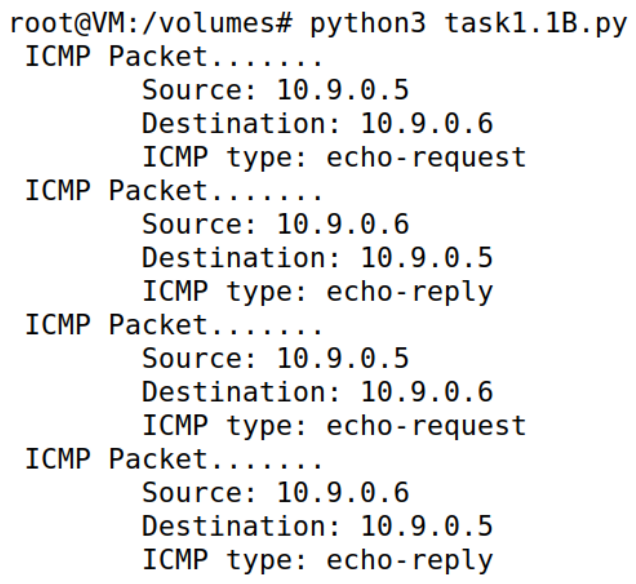
• Question: Capture only the ICMP packet



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

def print_pkt(pkt):
    if pkt[ICMP] is not None:
        if pkt[ICMP].type == 0 or pkt[ICMP].type == 8:
            print(" ICMP Packet.......")
            print(f"\tSource: {pkt[IP].src}")
            print(f"\tDestination: {pkt[IP].dst}")
            if pkt[ICMP].type == 0:
                print(f"\tICMP type: echo-reply")
            if pkt[ICMP].type == 8:
                print(f"\tICMP type: echo-request")

interfaces = ['br-70a143fa7795', 'enp0s3', 'lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```



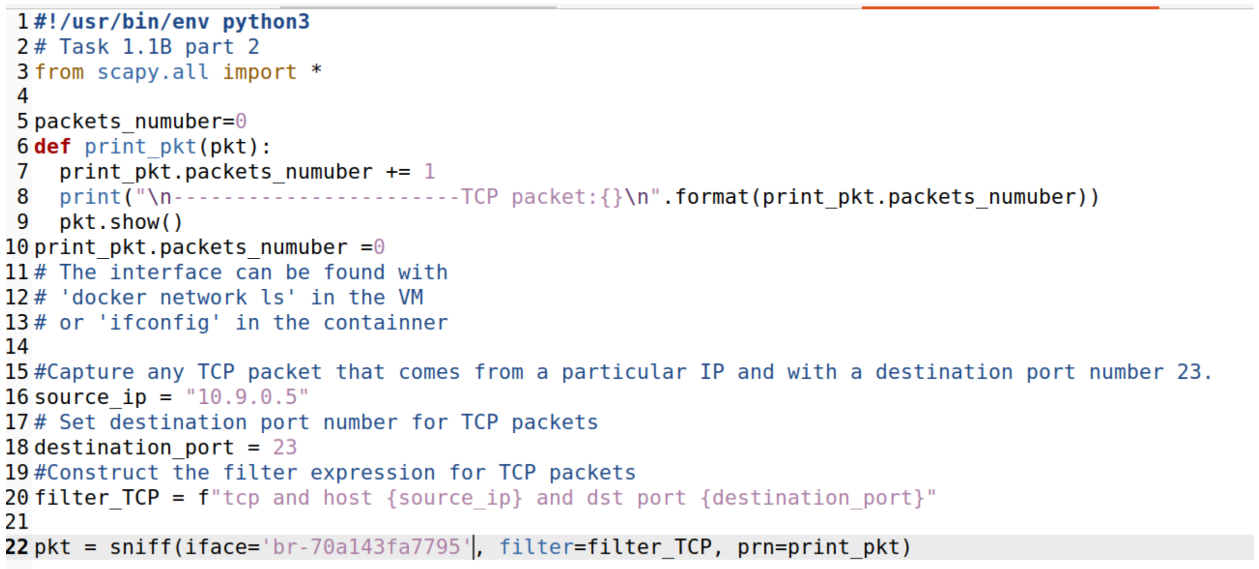
```
root@7a55353d006c:/# ping -c2 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.333 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.610 ms

--- 10.9.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1025ms
rtt min/avg/max/mdev = 0.333/0.471/0.610/0.138 ms
root@7a55353d006c:/#
```



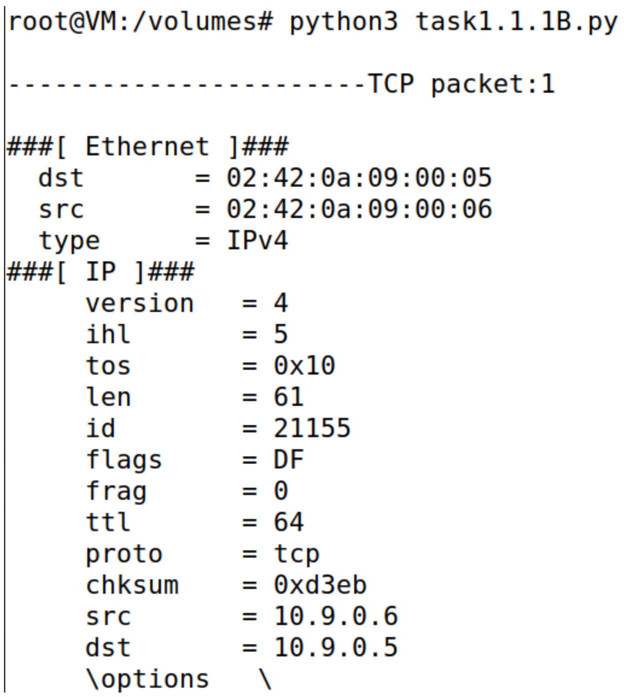
```
root@VM:/volumes# python3 task1.1B.py
 ICMP Packet.......
         Source: 10.9.0.5
         Destination: 10.9.0.6
         ICMP type: echo-request
 ICMP Packet.......
         Source: 10.9.0.6
         Destination: 10.9.0.5
         ICMP type: echo-reply
 ICMP Packet.......
         Source: 10.9.0.5
         Destination: 10.9.0.6
         ICMP type: echo-request
 ICMP Packet.......
         Source: 10.9.0.6
         Destination: 10.9.0.5
         ICMP type: echo-reply
```

Explanation: So using the filter ICMP i want to print only relevant details   source and destination IP typeof ICMP of packe**t.**

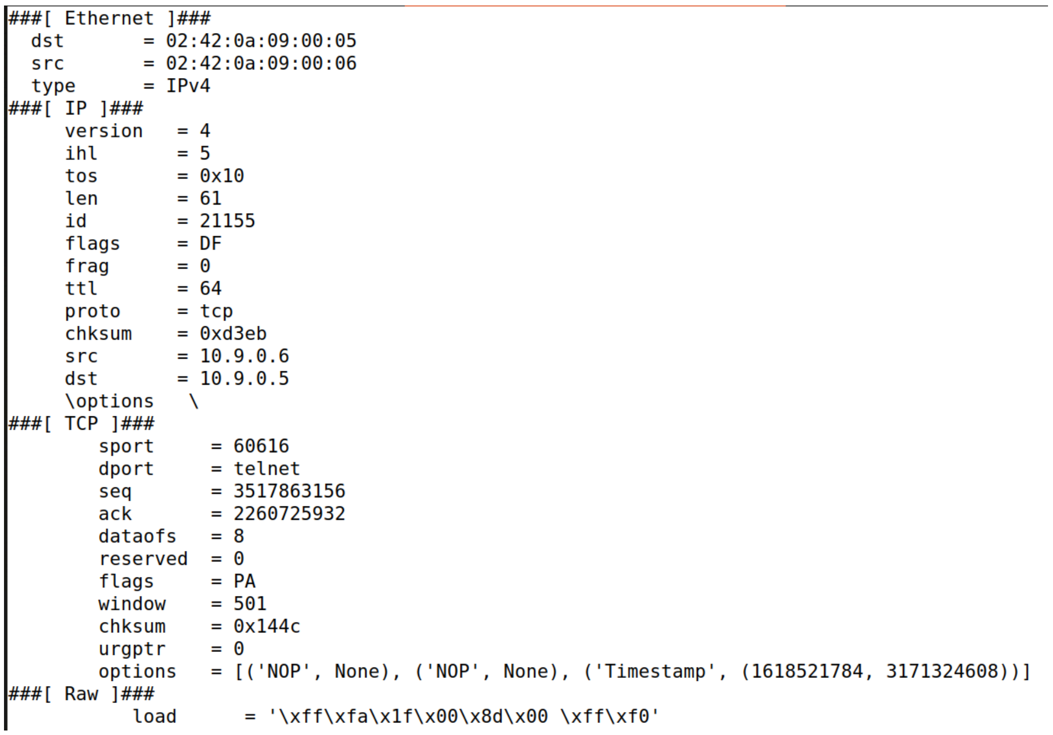
•Question: Capture any TCP packet that comes from a particular IP and with a destination port number 23.



```
#!/usr/bin/env python3
# Task 1.1B part 2
from scapy.all import *

packets_numuber=0
def print_pkt(pkt):
  print_pkt.packets_numuber += 1
  print("\n-----------------------TCP packet:{}\n".format(print_pkt.packets_numuber))
  pkt.show()
print_pkt.packets_numuber =0
# The interface can be found with
# 'docker network ls' in the VM
# or 'ifconfig' in the containner

#Capture any TCP packet that comes from a particular IP and with a destination port number 23.
source_ip = "10.9.0.5"
# Set destination port number for TCP packets
destination_port = 23
#Construct the filter expression for TCP packets
filter_TCP = f"tcp and host {source_ip} and dst port {destination_port}"

pkt = sniff(iface='br-70a143fa7795', filter=filter_TCP, prn=print_pkt)
```



```
root@VM:/volumes# python3 task1.1.1B.py

-----------------------TCP packet:1

###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 61
     id        = 21155
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xd3eb
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
```



```
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 61
     id        = 21155
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xd3eb
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ TCP ]###
        sport     = 60616
        dport     = telnet
        seq       = 3517863156
        ack       = 2260725932
        dataofs   = 8
        reserved  = 0
        flags     = PA
        window    = 501
        chksum    = 0x144c
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp', (1618521784, 3171324608))]
###[ Raw ]###
           load      = '\xff\xfa\x1f\x00\x8d\x00 \xff\xf0'
```

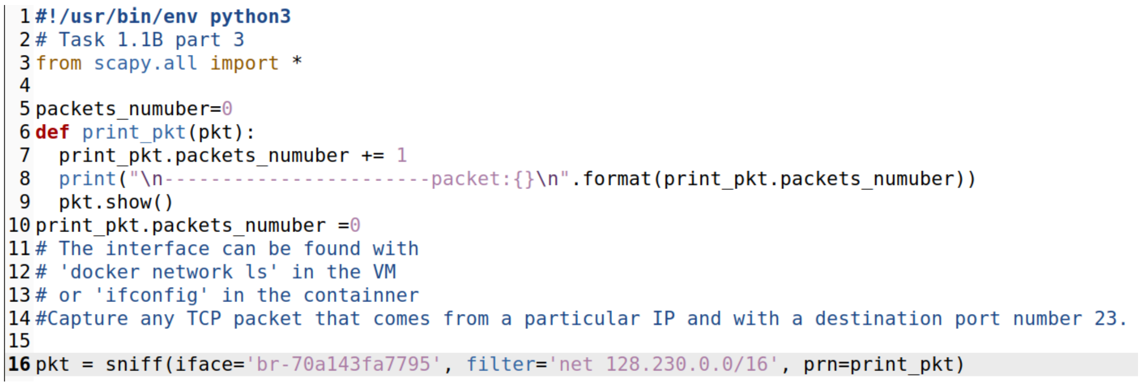
Explanation:

Above code filter the packet based on **TCP and Src ip and tcp port .** It print only the relevant detail

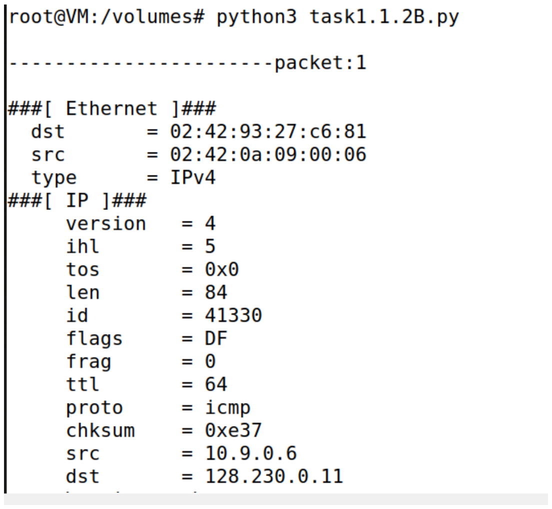
Src ip= 10.9.0.5

Telent command here used : **telenet 10.9.0.5**

• Question: Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to



```python
#!/usr/bin/env python3
# Task 1.1B part 3
from scapy.all import *

packets_numuber=0
def print_pkt(pkt):
  print_pkt.packets_numuber += 1
  print("\n-----------------------packet:{}\n".format(print_pkt.packets_numuber))
  pkt.show()
print_pkt.packets_numuber =0
# The interface can be found with
# 'docker network ls' in the VM
# or 'ifconfig' in the container
#Capture any TCP packet that comes from a particular IP and with a destination port number 23.

pkt = sniff(iface='br-70a143fa7795', filter='net 128.230.0.0/16', prn=print_pkt)
```



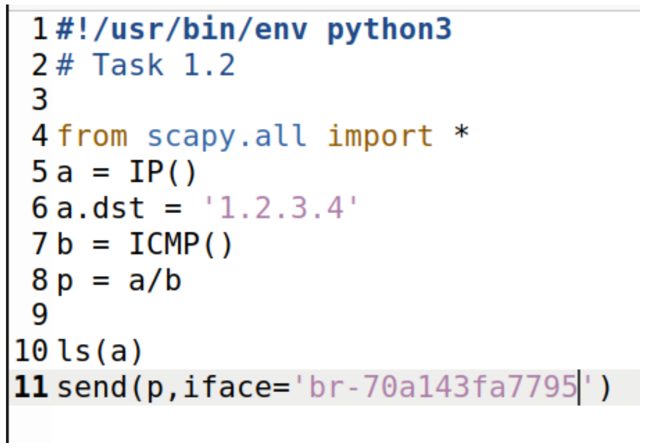
```
root@VM:/volumes# python3 task1.1.2B.py

-----------------------packet:1

###[ Ethernet ]###
  dst       = 02:42:93:27:c6:81
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 41330
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xe37
     src       = 10.9.0.6
     dst       = 128.230.0.11
```

```
-----------------------packet:1

###[ Ethernet ]###
  dst       = 02:42:93:27:c6:81
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 41330
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xe37
     src       = 10.9.0.6
     dst       = 128.230.0.11
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xd3d1
        id        = 0x25
        seq       = 0x1
###[ Raw ]###
           load      = '\xf2\xf1/f\x00\x00\x00\x00?\xdd\x03\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x
1f !"#$%&\'()*+,-./01234567'
```

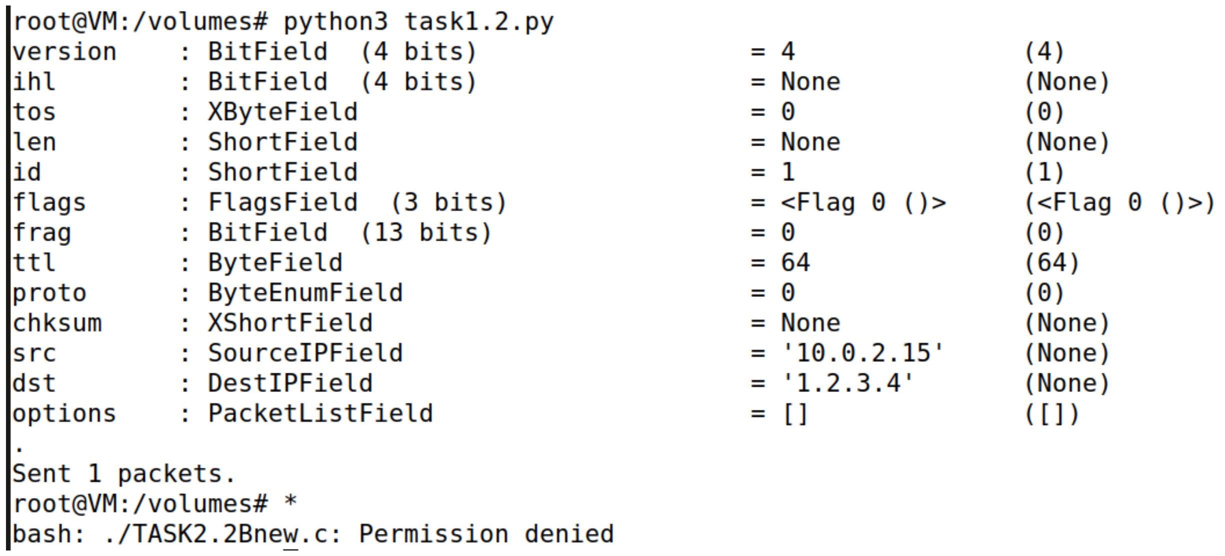
Explanation: Filter use to capture only the packet with dst net are in 128.230.0.0./16. So when packet of type dst net 128.230.0.0./16 comming it return true so it print thedetails of packet

Task 1.2: Spoofing ICMP Packets:

As a packet spoofing tool, Scapy allows us to set the fields of IP packets to arbitrary values. The objective of this task is to spoof IP packets with an arbitrary source IP address. We will spoof ICMP echo request packets, and send them to another VM on the same network. We will use Wireshark to observe whether our request will be accepted by the receiver. If it is accepted, an echo reply packet will be sent to the spoofed IP address.



```
#!/usr/bin/env python3
# Task 1.2

from scapy.all import *
a = IP()
a.dst = '1.2.3.4'
b = ICMP()
p = a/b

ls(a)
send(p,iface='br-70a143fa7795')
```

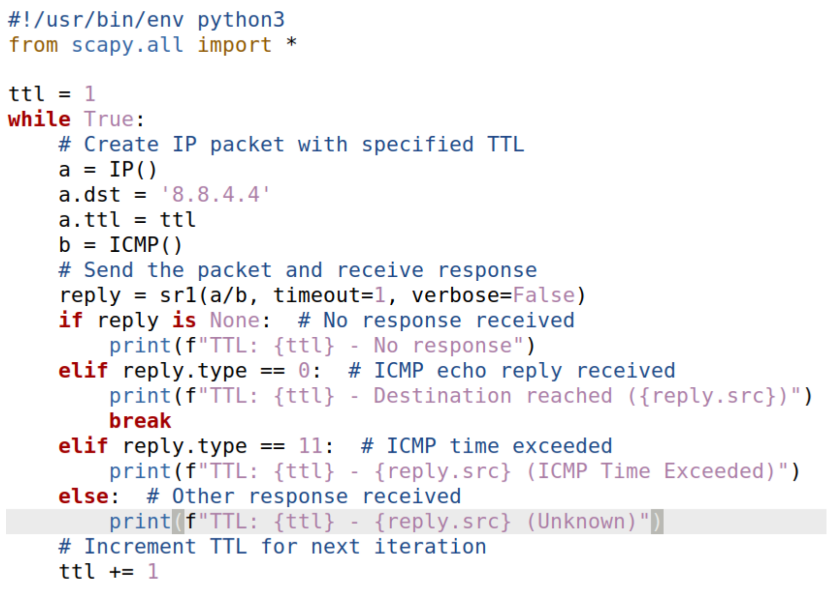


```
root@VM:/volumes# python3 task1.2.py
version    : BitField  (4 bits)                  = 4              (4)
ihl        : BitField  (4 bits)                  = None           (None)
tos        : XByteField                          = 0              (0)
len        : ShortField                          = None           (None)
id         : ShortField                          = 1              (1)
flags      : FlagsField  (3 bits)                = <Flag 0 ()>    (<Flag 0 ()>)
frag       : BitField  (13 bits)                 = 0              (0)
ttl        : ByteField                           = 64             (64)
proto      : ByteEnumField                       = 0              (0)
chksum     : XShortField                         = None           (None)
src        : SourceIPField                       = '10.0.2.15'    (None)
dst        : DestIPField                         = '1.2.3.4'      (None)
options    : PacketListField                     = []             ([])
.
Sent 1 packets.
root@VM:/volumes# *
bash: ./TASK2.2Bnew.c: Permission denied
```

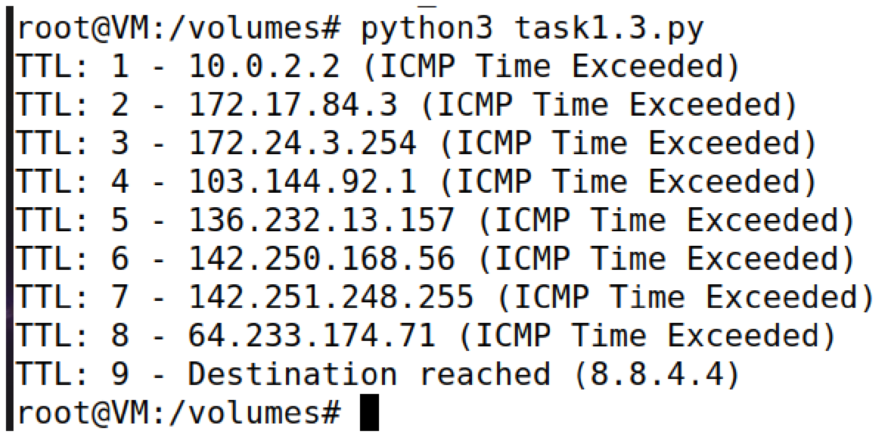
Here spoof ICMP packet with source IP address left unspecified so it allow it to set arbitrarily assign by the network.

Task 1.3: Traceroute:

The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between your VM and a selected destination. This is basically what is implemented by the traceroutetool. Inthis task, wewillwrite our own tool. The idea is quite straightforward: just send an packet (any type) to thedestination, with its Time-To-Live (TTL) field set to 1 first. This packet will be dropped by the first router,which will send us an ICMP error message, telling us that the time-to-live has exceeded. That is how we getthe IP address of the first router.



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

ttl = 1
while True:
    # Create IP packet with specified TTL
    a = IP()
    a.dst = '8.8.4.4'
    a.ttl = ttl
    b = ICMP()
    # Send the packet and receive response
    reply = sr1(a/b, timeout=1, verbose=False)
    if reply is None:  # No response received
        print(f"TTL: {ttl} - No response")
    elif reply.type == 0:  # ICMP echo reply received
        print(f"TTL: {ttl} - Destination reached ({reply.src})")
        break
    elif reply.type == 11:  # ICMP time exceeded
        print(f"TTL: {ttl} - {reply.src} (ICMP Time Exceeded)")
    else:  # Other response received
        print(f"TTL: {ttl} - {reply.src} (Unknown)")
    # Increment TTL for next iteration
    ttl += 1
```



```
root@VM:/volumes# python3 task1.3.py
TTL: 1 - 10.0.2.2 (ICMP Time Exceeded)
TTL: 2 - 172.17.84.3 (ICMP Time Exceeded)
TTL: 3 - 172.24.3.254 (ICMP Time Exceeded)
TTL: 4 - 103.144.92.1 (ICMP Time Exceeded)
TTL: 5 - 136.232.13.157 (ICMP Time Exceeded)
TTL: 6 - 142.250.168.56 (ICMP Time Exceeded)
TTL: 7 - 142.251.248.255 (ICMP Time Exceeded)
TTL: 8 - 64.233.174.71 (ICMP Time Exceeded)
TTL: 9 - Destination reached (8.8.4.4)
root@VM:/volumes# 
```

Explanation: Traceroot is used to provide any distination what number of hop far from source. Intially we try it mannually by setting ttl value: 1,2,3….

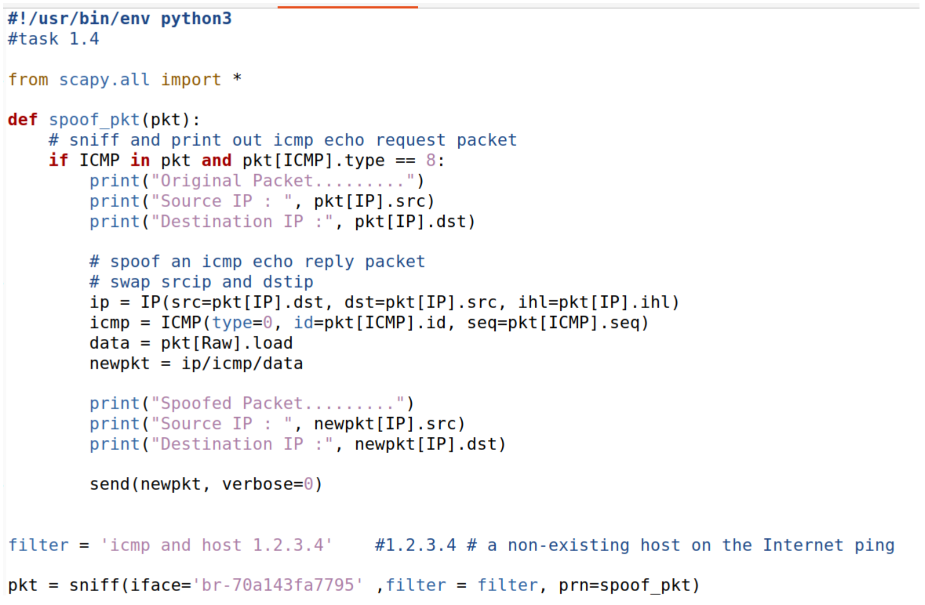
Then using loop it automatically check that any dst IP how much far from loop.

It start from ttl value 1, if destination found then it reply using ICMP packet otherwise then it iterate for next TTL value untill the destination is identified.

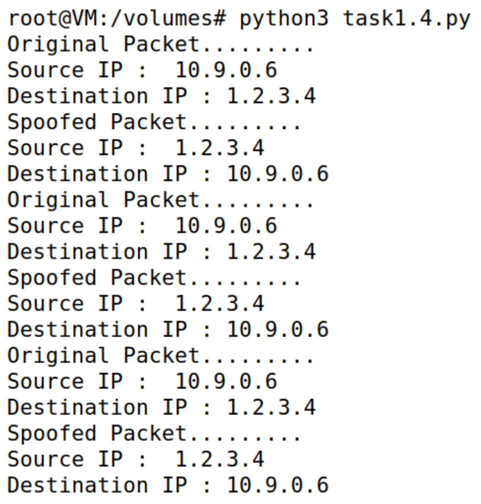
But if TTL value exceed maximum limit i,e dst host very far from host.
In above code the **dst IP =8.8.4.4** is **identified in 9 iteration**
Task 1.4: Sniffing and-then Spoofing:
In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and-then-spoof program. You need two machines on the same LAN: the VM and the user container. From the user container, you ping an IP X. This will generate an ICMP echo request packet. If X is alive, the ping program will receive an echo reply, and print out the response. Your sniff-and-then-spoof programruns on theVM, which monitors theLANthrough packet sniffing. Whenever it sees an ICMP echo request,regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regardless of whether machine X is alive or not, the ping program will always receive a reply, indicating that X is alive. You need to use Scapy to do this task. In your report,you need to provide evidence to demonstrate that your technique works.

ARP protocol: Address resolution Protocol is used to find the MAC address of a given IP address.
To find IP address the ARP protocol broadcast a packet which contain their (host) ip and MAC address and Destination device only IP address . And device which their IP address it unicast the the packet to their host.

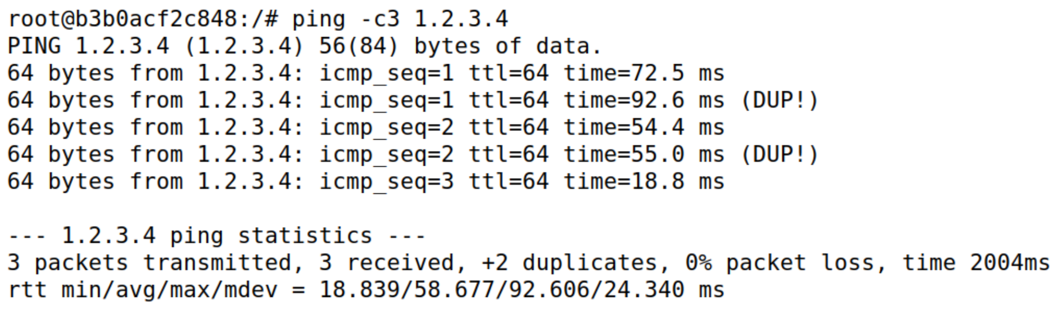
ping 1.2.3.4 # a non-existing host on the Internet



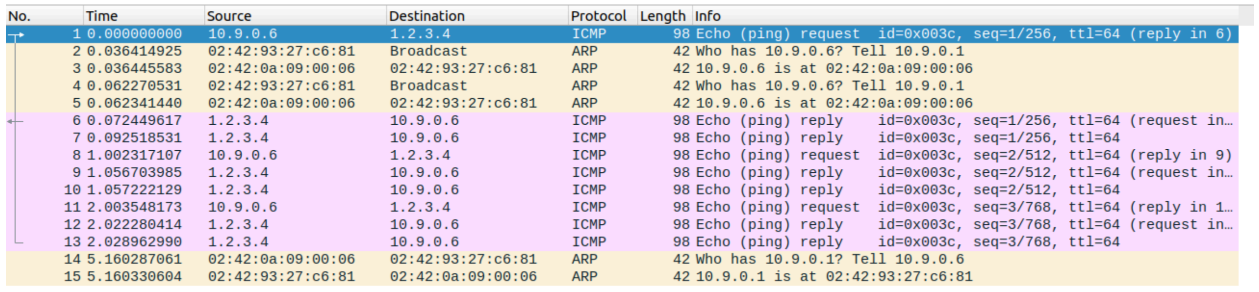
```
#!/usr/bin/env python3
#task 1.4

from scapy.all import *

def spoof_pkt(pkt):
    # sniff and print out icmp echo request packet
    if ICMP in pkt and pkt[ICMP].type == 8:
        print("Original Packet.........")
        print("Source IP : ", pkt[IP].src)
        print("Destination IP :", pkt[IP].dst)

        # spoof an icmp echo reply packet
        # swap srcip and dstip
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data

        print("Spoofed Packet.........")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)

        send(newpkt, verbose=0)


filter = 'icmp and host 1.2.3.4'    #1.2.3.4 # a non-existing host on the Internet ping

pkt = sniff(iface='br-70a143fa7795' ,filter = filter, prn=spoof_pkt)
```



```
root@VM:/volumes# python3 task1.4.py
Original Packet.........
Source IP :  10.9.0.6
Destination IP : 1.2.3.4
Spoofed Packet.........
Source IP :  1.2.3.4
Destination IP : 10.9.0.6
Original Packet.........
Source IP :  10.9.0.6
Destination IP : 1.2.3.4
Spoofed Packet.........
Source IP :  1.2.3.4
Destination IP : 10.9.0.6
Original Packet.........
Source IP :  10.9.0.6
Destination IP : 1.2.3.4
Spoofed Packet.........
Source IP :  1.2.3.4
Destination IP : 10.9.0.6
```



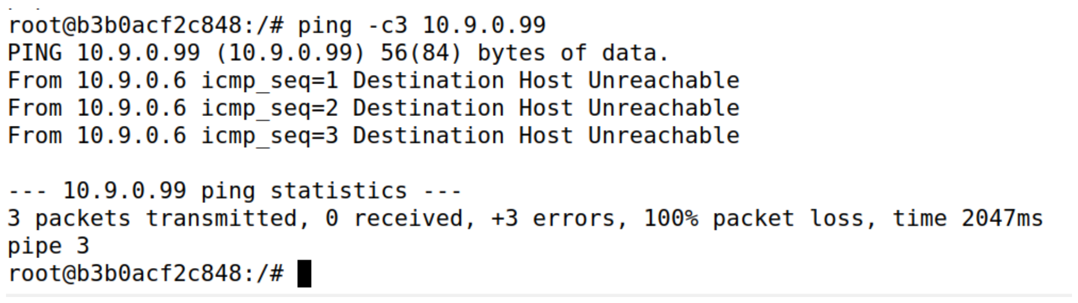
```
root@b3b0acf2c848:/# ping -c3 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=72.5 ms
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=92.6 ms (DUP!)
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=54.4 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=55.0 ms (DUP!)
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=18.8 ms

--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, +2 duplicates, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 18.839/58.677/92.606/24.340 ms
```



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.9.0.6	1.2.3.4	ICMP	98	Echo (ping) request id=0x003c, seq=1/256, ttl=64 (reply in 6)
2	0.036414925	02:42:93:27:c6:81	Broadcast	ARP	42	Who has 10.9.0.6? Tell 10.9.0.1
3	0.036445583	02:42:0a:09:00:06	02:42:93:27:c6:81	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
4	0.062270531	02:42:93:27:c6:81	Broadcast	ARP	42	Who has 10.9.0.6? Tell 10.9.0.1
5	0.062341440	02:42:0a:09:00:06	02:42:93:27:c6:81	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
6	0.072449617	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x003c, seq=1/256, ttl=64 (request in…
7	0.092518531	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x003c, seq=1/256, ttl=64
8	1.002317107	10.9.0.6	1.2.3.4	ICMP	98	Echo (ping) request id=0x003c, seq=2/512, ttl=64 (reply in 9)
9	1.056703985	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x003c, seq=2/512, ttl=64 (request in…
10	1.057222129	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x003c, seq=2/512, ttl=64
11	2.003548173	10.9.0.6	1.2.3.4	ICMP	98	Echo (ping) request id=0x003c, seq=3/768, ttl=64 (reply in 1…
12	2.022280414	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x003c, seq=3/768, ttl=64 (request in…
13	2.028962990	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x003c, seq=3/768, ttl=64
14	5.160287061	02:42:0a:09:00:06	02:42:93:27:c6:81	ARP	42	Who has 10.9.0.1? Tell 10.9.0.6
15	5.160330604	02:42:93:27:c6:81	02:42:0a:09:00:06	ARP	42	10.9.0.1 is at 02:42:93:27:c6:81

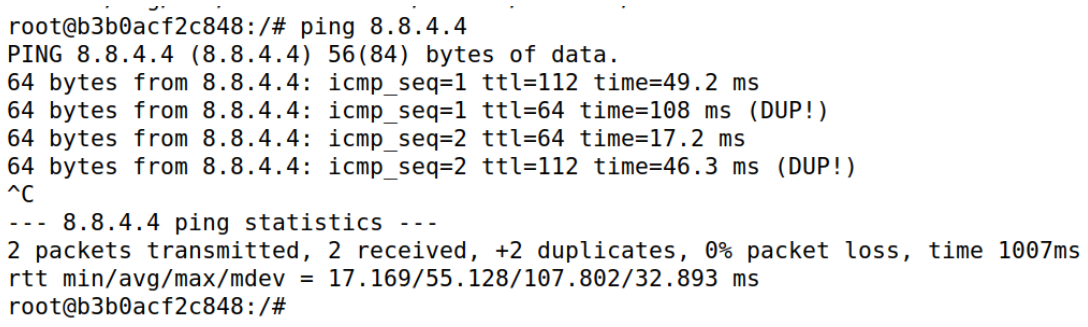
Explanation: In case when non existing host on the internet all packet get return because as we see in wireshark ARP try to find out **Who is 10.9.0.6** so in this case attacker attacker return the answer along with ICMP packet is return.

Case 2: ping 10.9.0.99 # a non-existing host on the LAN

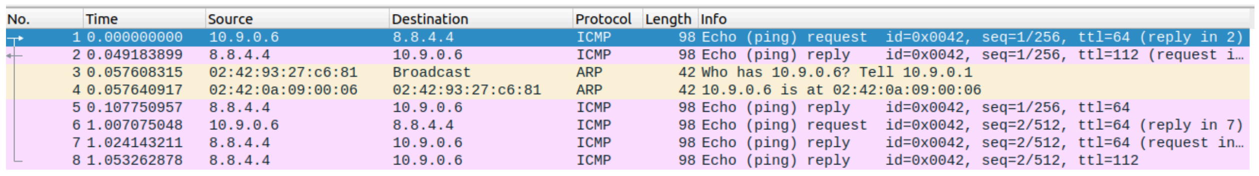


```
root@b3b0acf2c848:/# ping -c3 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.6 icmp_seq=1 Destination Host Unreachable
From 10.9.0.6 icmp_seq=2 Destination Host Unreachable
From 10.9.0.6 icmp_seq=3 Destination Host Unreachable

--- 10.9.0.99 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2047ms
pipe 3
root@b3b0acf2c848:/# 
```

Case 3: ping 8.8.8.8 # an existing host on the Internet



```
root@b3b0acf2c848:/# ping 8.8.4.4
PING 8.8.4.4 (8.8.4.4) 56(84) bytes of data.
64 bytes from 8.8.4.4: icmp_seq=1 ttl=112 time=49.2 ms
64 bytes from 8.8.4.4: icmp_seq=1 ttl=64 time=108 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=2 ttl=64 time=17.2 ms
64 bytes from 8.8.4.4: icmp_seq=2 ttl=112 time=46.3 ms (DUP!)
^C
--- 8.8.4.4 ping statistics ---
2 packets transmitted, 2 received, +2 duplicates, 0% packet loss, time 1007ms
rtt min/avg/max/mdev = 17.169/55.128/107.802/32.893 ms
root@b3b0acf2c848:/#
```



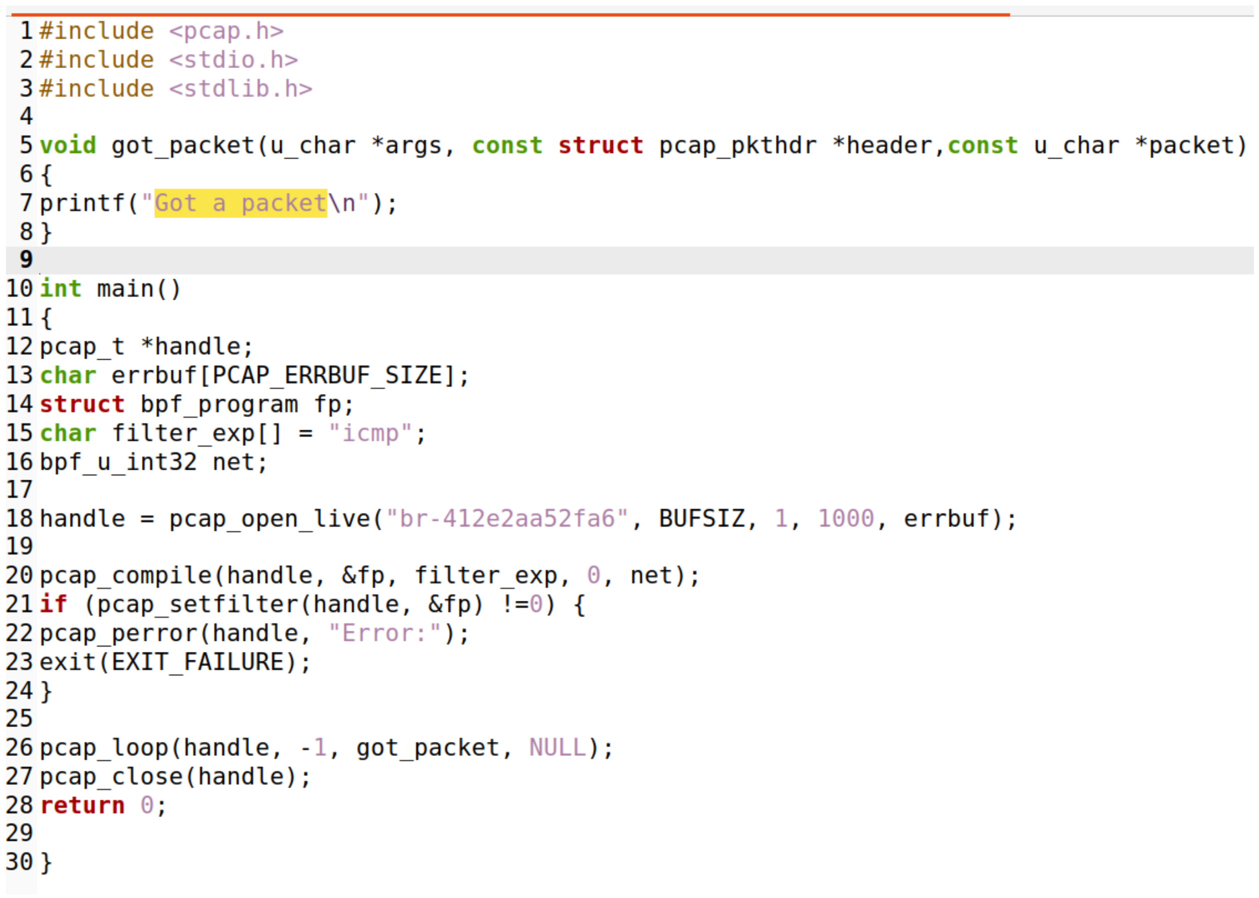
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.9.0.6	8.8.4.4	ICMP	98	Echo (ping) request id=0x0042, seq=1/256, ttl=64 (reply in 2)
2	0.049183899	8.8.4.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x0042, seq=1/256, ttl=112 (request i…
3	0.057608315	02:42:93:27:c6:81	Broadcast	ARP	42	Who has 10.9.0.6? Tell 10.9.0.1
4	0.057640917	02:42:0a:09:00:06	02:42:93:27:c6:81	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
5	0.107750957	8.8.4.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x0042, seq=1/256, ttl=64
6	1.007075048	10.9.0.6	8.8.4.4	ICMP	98	Echo (ping) request id=0x0042, seq=2/512, ttl=64 (reply in 7)
7	1.024143211	8.8.4.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x0042, seq=2/512, ttl=64 (request in…
8	1.053262878	8.8.4.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x0042, seq=2/512, ttl=112

Explanation: As ‘8.8.4.4’ are existing host on the internet we get duplicate response because it relly exist on the internet

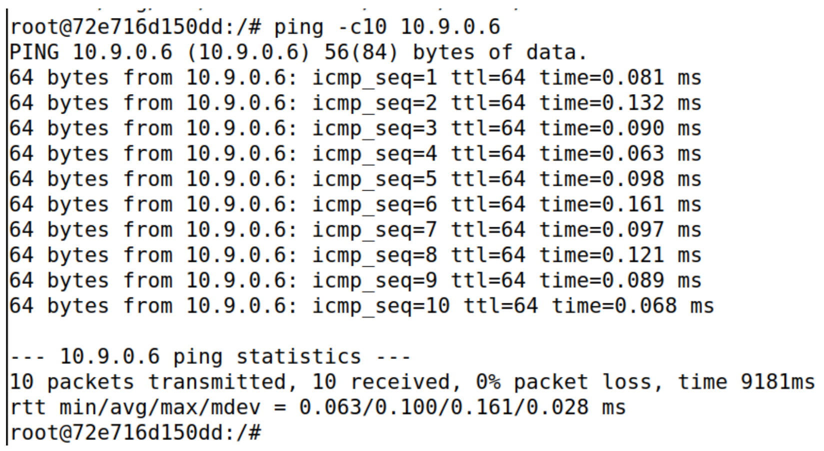
Lab Assignment Task Set 2: Writing Programs to Sniff and Spoof Packets

Writing Packet Sniffing Program

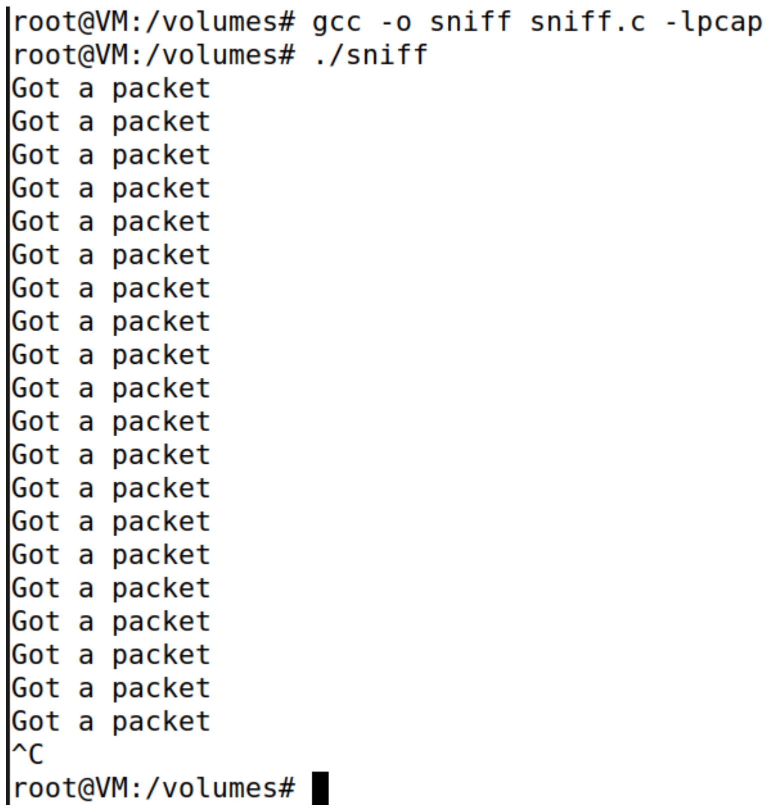
Sniffer programs can be easily written using the pcap library. With pcap, the task of sniffers becomes invoking a simple sequence of procedures in the pcaplibrary. At the end of the sequence, packets will be put in the buffer for further processing as soon as they are captured. All the details of packet capturing are handled by the pcap library.



```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>

void got_packet(u_char *args, const struct pcap_pkthdr *header,const u_char *packet)
{
printf("Got a packet\n");
}

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

handle = pcap_open_live("br-412e2aa52fa6", BUFSIZ, 1, 1000, errbuf);

pcap_compile(handle, &fp, filter_exp, 0, net);
if (pcap_setfilter(handle, &fp) !=0) {
pcap_perror(handle, "Error:");
exit(EXIT_FAILURE);
}

pcap_loop(handle, -1, got_packet, NULL);
pcap_close(handle);
return 0;

}
```



```
root@72e716d150dd:/# ping -c10 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=0.081 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.132 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.090 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.063 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.098 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.161 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.097 ms
64 bytes from 10.9.0.6: icmp_seq=8 ttl=64 time=0.121 ms
64 bytes from 10.9.0.6: icmp_seq=9 ttl=64 time=0.089 ms
64 bytes from 10.9.0.6: icmp_seq=10 ttl=64 time=0.068 ms

--- 10.9.0.6 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9181ms
rtt min/avg/max/mdev = 0.063/0.100/0.161/0.028 ms
root@72e716d150dd:/#
```



```
root@VM:/volumes# gcc -o sniff sniff.c -lpcap
root@VM:/volumes# ./sniff
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
^C
root@VM:/volumes# 
```

Instead of message got packet we can print souce and destination of the packet:

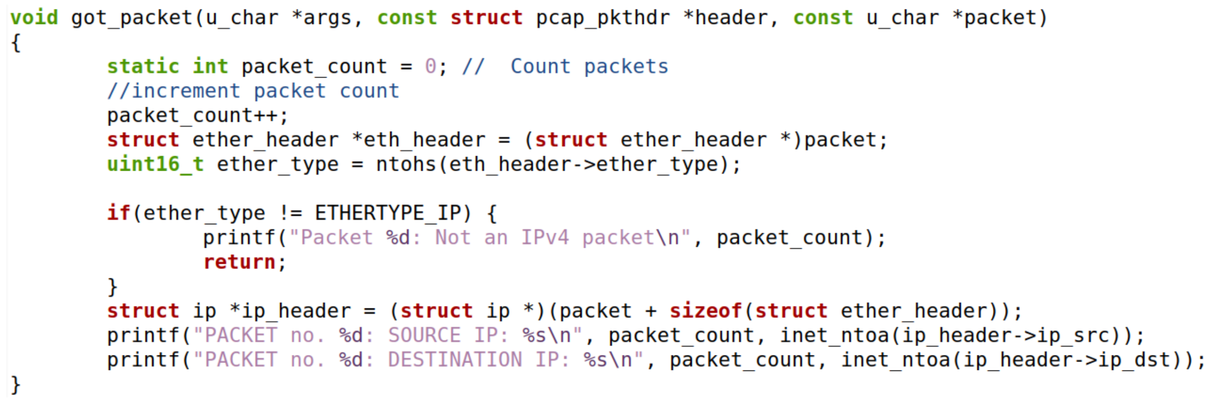
By replacring the **printf line** in **got_packet** by their line code

```
struct ethheader *eth = (struct ethheader *)packet;
if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IPv4 type
   struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));
   printf("Source: %s ", inet_ntoa(ip->iph_sourceip));
   printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
}
```

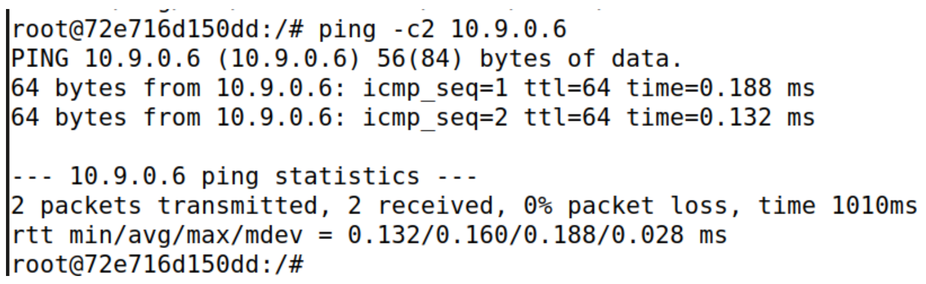
Explanation: above program capture the packets, network using pcap library and display only the message ‘Got a packet’ as request, reply .

I ping 10 packets using command ping -c10 10.9.0.6 from host A at seed-attacker Got a packet message printed.

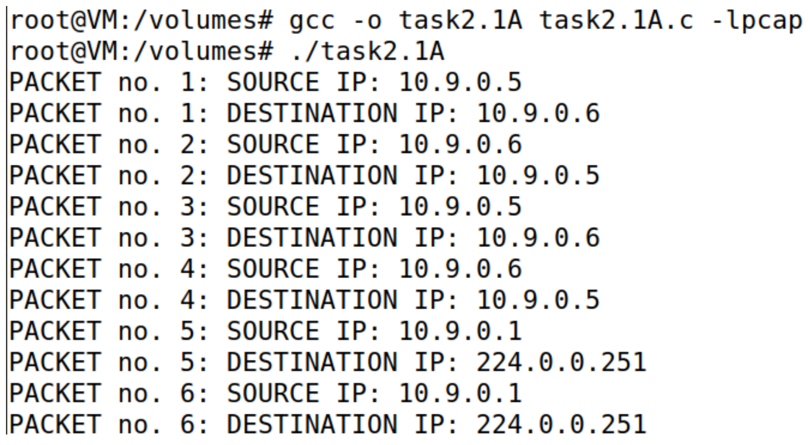
Task 2.1A: In this task, students need to write a sniffer program to print out the source and destination IP addresses of each captured packet.



```c
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
        static int packet_count = 0; //  Count packets
        //increment packet count
        packet_count++;
        struct ether_header *eth_header = (struct ether_header *)packet;
        uint16_t ether_type = ntohs(eth_header->ether_type);

        if(ether_type != ETHERTYPE_IP) {
                printf("Packet %d: Not an IPv4 packet\n", packet_count);
                return;
        }
        struct ip *ip_header = (struct ip *)(packet + sizeof(struct ether_header));
        printf("PACKET no. %d: SOURCE IP: %s\n", packet_count, inet_ntoa(ip_header->ip_src));
        printf("PACKET no. %d: DESTINATION IP: %s\n", packet_count, inet_ntoa(ip_header->ip_dst));
}
```



```
root@72e716d150dd:/# ping -c2 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=0.188 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.132 ms

--- 10.9.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1010ms
rtt min/avg/max/mdev = 0.132/0.160/0.188/0.028 ms
root@72e716d150dd:/#
```



```
root@VM:/volumes# gcc -o task2.1A task2.1A.c -lpcap
root@VM:/volumes# ./task2.1A
PACKET no. 1: SOURCE IP: 10.9.0.5
PACKET no. 1: DESTINATION IP: 10.9.0.6
PACKET no. 2: SOURCE IP: 10.9.0.6
PACKET no. 2: DESTINATION IP: 10.9.0.5
PACKET no. 3: SOURCE IP: 10.9.0.5
PACKET no. 3: DESTINATION IP: 10.9.0.6
PACKET no. 4: SOURCE IP: 10.9.0.6
PACKET no. 4: DESTINATION IP: 10.9.0.5
PACKET no. 5: SOURCE IP: 10.9.0.1
PACKET no. 5: DESTINATION IP: 224.0.0.251
PACKET no. 6: SOURCE IP: 10.9.0.1
PACKET no. 6: DESTINATION IP: 224.0.0.251
```

Explanation : this program is similar to the above program , here just print the source and destination received packet.

• **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:

Firstof all we open a live pcap session on NIC using the Pcap_open_live ,

Pcap_open_live: this function opens the specified network interface for packet capture i,e it monitors whole network traffic.

Then we set filter

pcap_compile(): it compile a packet filter expression for use with the pacp_setfilter function

pcap_setfilter():It set the filter to be applied to the incoming packets,allowing the program to capture only relevant traffic

Pcap_loop: It initiates loop to capture and process packets according to the given filter

• **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:

Root privilage is required to set the network interface in promiscous mode and create raw socket which is crucial to see and capture the network traffic in the interface.

So, when we try to run the program without root privilage , in this situation pcap_open_live function fails to access device so it give error in the complete program

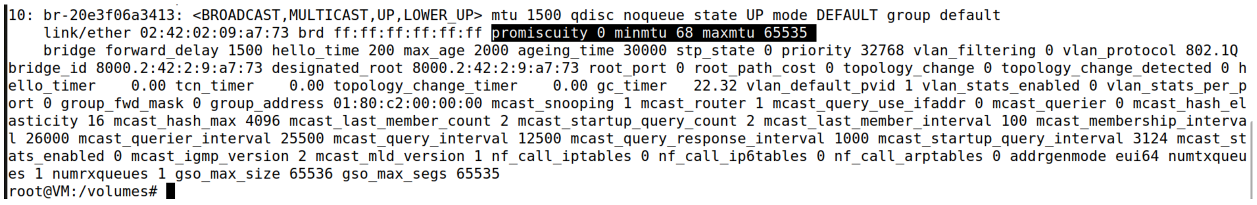
• **Question 3.** Please turn on and turn off the promiscuous mode in your sniffer program. The value 1 of the third parameter in pcap open live() turns on the promiscuous mode (use 0 to turn it off). Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this. You can use the following command to check whether an interface's promiscuous mode is on or off (look at the promiscuity's value).

Answer:

When the value 1 of the third parameter in pcap open live() turns on the promiscuous mode it will capture all packets which arrive on the network interface.

We can see through running program   where the   pc= pcap.pcap(,**promisc=True)**

Also,when the value 0 of the third parameter in pcap open live() turns off the promiscuous mode ,the program captures packets only addressed to the local machine.

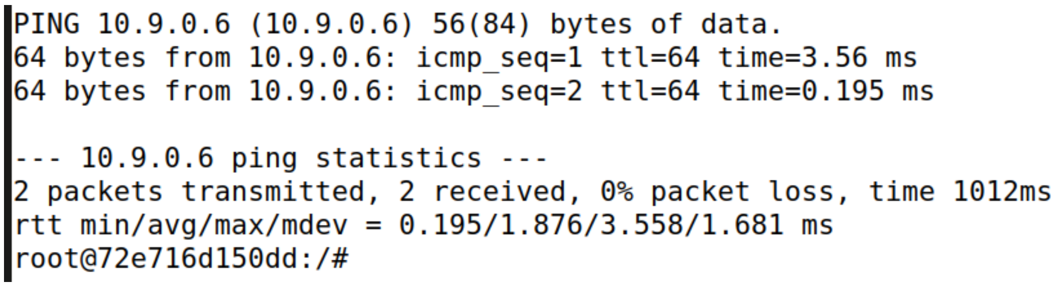


```
10: br-20e3f06a3413: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:02:09:a7:73 brd ff:ff:ff:ff:ff:ff promiscuity 0 minmtu 68 maxmtu 65535
    bridge forward_delay 1500 hello_time 200 max_age 2000 ageing_time 30000 stp_state 0 priority 32768 vlan_filtering 0 vlan_protocol 802.1Q
bridge_id 8000.2:42:2:9:a7:73 designated_root 8000.2:42:2:9:a7:73 root_port 0 root_path_cost 0 topology_change 0 topology_change_detected 0 h
ello_timer    0.00 tcn_timer    0.00 topology_change_timer    0.00 gc_timer   22.32 vlan_default_pvid 1 vlan_stats_enabled 0 vlan_stats_per_p
ort 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_el
asticity 16 mcast_hash_max 4096 mcast_last_member_count 2 mcast_startup_query_count 2 mcast_last_member_interval 100 mcast_membership_interva
l 26000 mcast_querier_interval 25500 mcast_query_interval 12500 mcast_query_response_interval 1000 mcast_startup_query_interval 3124 mcast_st
ats_enabled 0 mcast_igmp_version 2 mcast_mld_version 1 nf_call_iptables 0 nf_call_ip6tables 0 nf_call_arptables 0 addrgenmode eui64 numtxqueu
es 1 numrxqueues 1 gso_max_size 65536 gso_max_segs 65535
root@VM:/volumes#
```

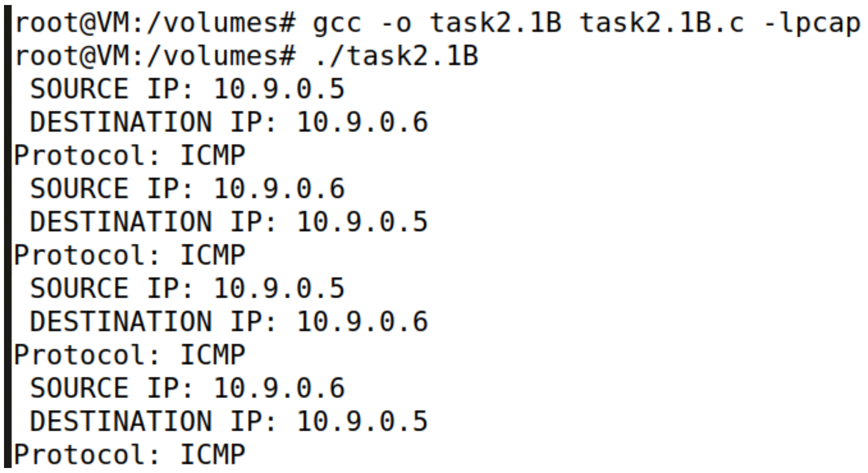
Task 2.1B: Writing Filters. Please write filter expressions for your sniffer program to capture each of the following. You can find online manuals for pcap filters. In your Lab Assignment reports, you need to include screenshots to show the results after applying each of these filters.

- **Capture the ICMP packets between two specific hosts**

```c
#include <stdlib.h>
#include <stdio.h>
#include <pcap.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 ether_header *eth_header = (struct ether_header *)packet;
    // here check  Ethernet type indicates an IPv4 packet
        if (ntohs(eth_header->ether_type) == 0x0800) {

    struct ip *ip_header = (struct ip *)(packet + sizeof(struct ether_header));

        // is protocol is ICMP  //ip->iph_protocol== IPPROTO_ICMP
        if (ip_header->ip_p == IPPROTO_ICMP) {
        printf(" SOURCE IP: %s\n",inet_ntoa(ip_header->ip_src));
        printf(" DESTINATION IP: %s\n",inet_ntoa(ip_header->ip_dst));
                printf("Protocol: ICMP\n");
                return;
        }
    }
    printf("Not an ICMP packet\n");
}


int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip"; // filter expression to capture all IP packets
    bpf_u_int32 net, mask;
    //open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("br-412e2aa52fa6", BUFSIZ, 1, 1000, errbuf);

    //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);
    }
    //capture packets
    pcap_loop(handle, -1, got_packet, NULL);
    //Close the handle
    pcap_close(handle);
    return 0;
}
```



```
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=3.56 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.195 ms

--- 10.9.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1012ms
rtt min/avg/max/mdev = 0.195/1.876/3.558/1.681 ms
root@72e716d150dd:/#
```



```
root@VM:/volumes# gcc -o task2.1B task2.1B.c -lpcap
root@VM:/volumes# ./task2.1B
 SOURCE IP: 10.9.0.5
 DESTINATION IP: 10.9.0.6
Protocol: ICMP
 SOURCE IP: 10.9.0.6
 DESTINATION IP: 10.9.0.5
Protocol: ICMP
 SOURCE IP: 10.9.0.5
 DESTINATION IP: 10.9.0.6
Protocol: ICMP
 SOURCE IP: 10.9.0.6
 DESTINATION IP: 10.9.0.5
Protocol: ICMP
```

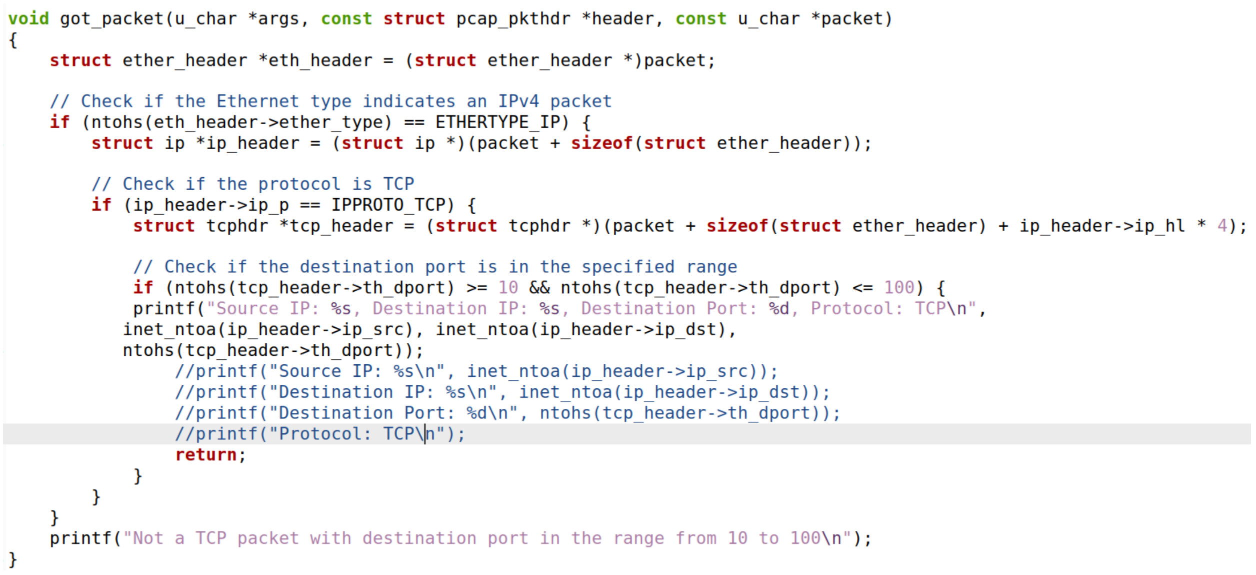
Explanation: Above program check the condition if IPV4 type is true and then it check ICMP protocol type if it true then it print the

PROTOCOL TYPE

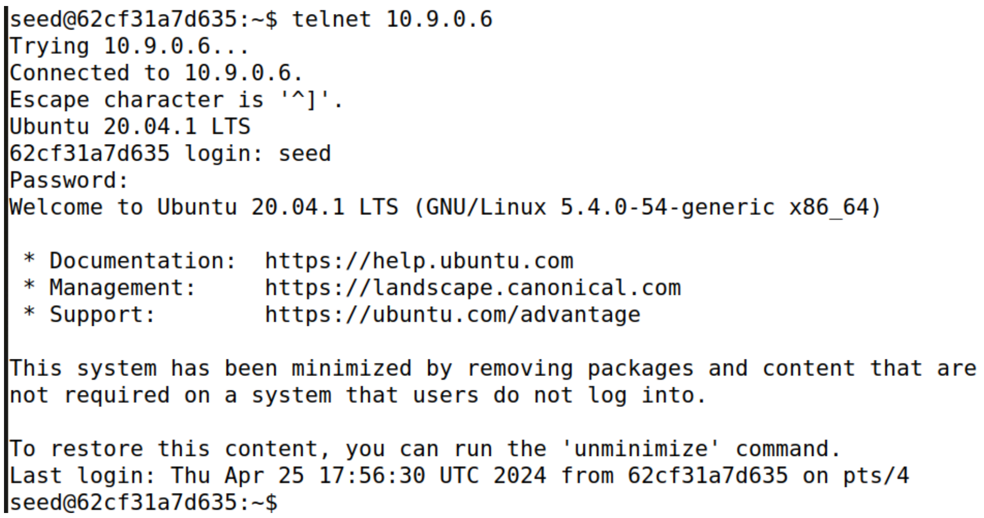
SOurce IP

Destination IP

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



```c
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    struct ether_header *eth_header = (struct ether_header *)packet;

    // Check if the Ethernet type indicates an IPv4 packet
    if (ntohs(eth_header->ether_type) == ETHERTYPE_IP) {
        struct ip *ip_header = (struct ip *)(packet + sizeof(struct ether_header));

        // Check if the protocol is TCP
        if (ip_header->ip_p == IPPROTO_TCP) {
            struct tcphdr *tcp_header = (struct tcphdr *)(packet + sizeof(struct ether_header) + ip_header->ip_hl * 4);

            // Check if the destination port is in the specified range
            if (ntohs(tcp_header->th_dport) >= 10 && ntohs(tcp_header->th_dport) <= 100) {
            printf("Source IP: %s, Destination IP: %s, Destination Port: %d, Protocol: TCP\n",
           inet_ntoa(ip_header->ip_src), inet_ntoa(ip_header->ip_dst),
           ntohs(tcp_header->th_dport));
                //printf("Source IP: %s\n", inet_ntoa(ip_header->ip_src));
                //printf("Destination IP: %s\n", inet_ntoa(ip_header->ip_dst));
                //printf("Destination Port: %d\n", ntohs(tcp_header->th_dport));
                //printf("Protocol: TCP\n");
                return;
            }
        }
    }
    printf("Not a TCP packet with destination port in the range from 10 to 100\n");
}
```



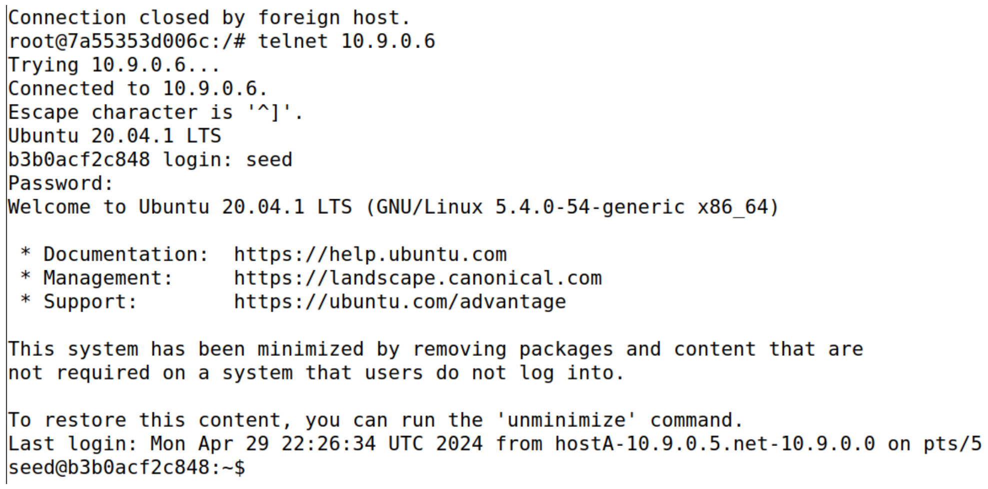
```
seed@62cf31a7d635:~$ telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
62cf31a7d635 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * 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: Thu Apr 25 17:56:30 UTC 2024 from 62cf31a7d635 on pts/4
seed@62cf31a7d635:~$
```

```
root@VM:/volumes# gcc -o sp task2.1BB.c -lpcap
root@VM:/volumes# ./sp
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
Source IP: 10.9.0.5, Destination IP: 10.9.0.5, Destination Port: 23, Protocol: TCP
```

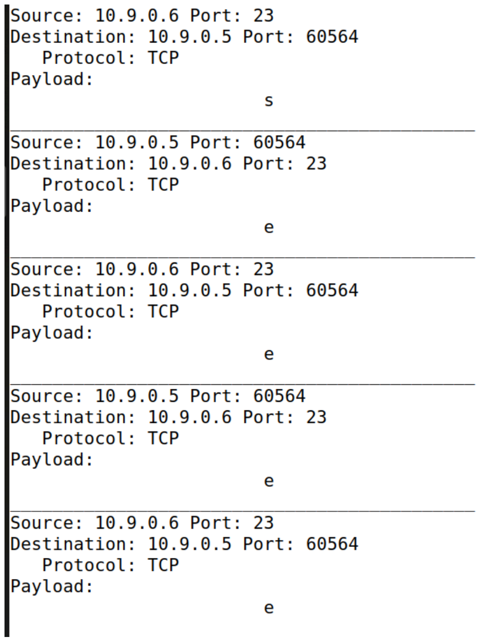
Explanation:

Here the TCP and destination Port range 10 - 100
The Packet capture by the above program if ti belong to IPV4 and its TCP type then it print
Source IP,
Destination IP,
Destination Port and
Protocol   of each packet

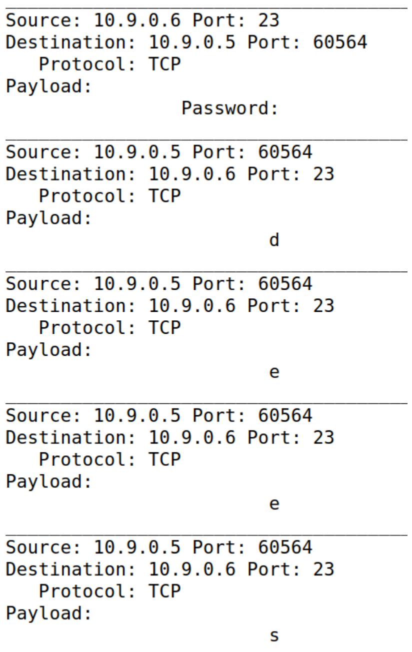
Task 2.1C: Sniffing Passwords. Please show how you can use your sniffer program to capture the password when somebody is using telnet on the network that you are monitoring. You may need to modify your sniffer code to print out the data part of a captured TCP packet (telnet uses TCP). It is acceptable if you print out the entire data part and then manually mark where the password (or part of it) is.



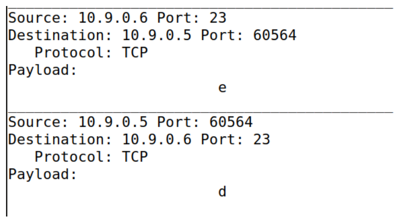
```
Connection closed by foreign host.
root@7a55353d006c:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
b3b0acf2c848 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * 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: Mon Apr 29 22:26:34 UTC 2024 from hostA-10.9.0.5.net-10.9.0.0 on pts/5
seed@b3b0acf2c848:~$
```



```
Source: 10.9.0.6 Port: 23
Destination: 10.9.0.5 Port: 60564
   Protocol: TCP
Payload:
                        s
____________________________________________
Source: 10.9.0.5 Port: 60564
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        e
____________________________________________
Source: 10.9.0.6 Port: 23
Destination: 10.9.0.5 Port: 60564
   Protocol: TCP
Payload:
                        e
____________________________________________
Source: 10.9.0.5 Port: 60564
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        e
____________________________________________
Source: 10.9.0.6 Port: 23
Destination: 10.9.0.5 Port: 60564
   Protocol: TCP
Payload:
                        e
```



```
______________________________________
Source: 10.9.0.6 Port: 23
Destination: 10.9.0.5 Port: 60564
   Protocol: TCP
Payload:
                Password:
______________________________________
Source: 10.9.0.5 Port: 60564
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        d
______________________________________
Source: 10.9.0.5 Port: 60564
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        e
______________________________________
Source: 10.9.0.5 Port: 60564
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        e
______________________________________
Source: 10.9.0.5 Port: 60564
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        s
```



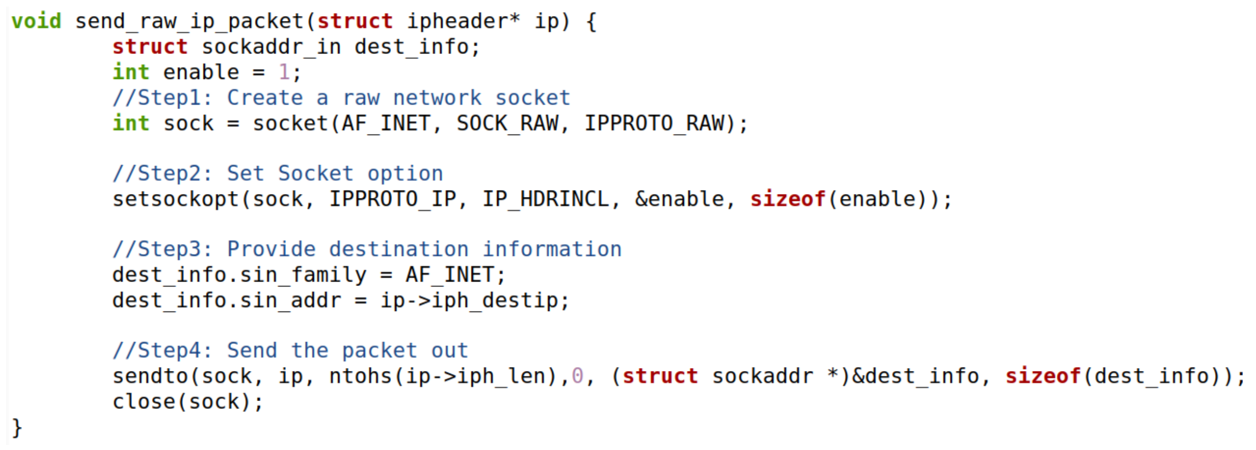
```
____________________________________________
Source: 10.9.0.6 Port: 23
Destination: 10.9.0.5 Port: 60564
   Protocol: TCP
Payload:
                        e
____________________________________________
Source: 10.9.0.5 Port: 60564
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        d
```

Explanation: Here filter used is tcp port telenet,

Above program sniff the tcp packet due to the set filters and when we execute using command: **telent 10.9.0.6** and after that i put userid and password it will capture and print at **seed-attacker terminal.** It print the payload of tcp.

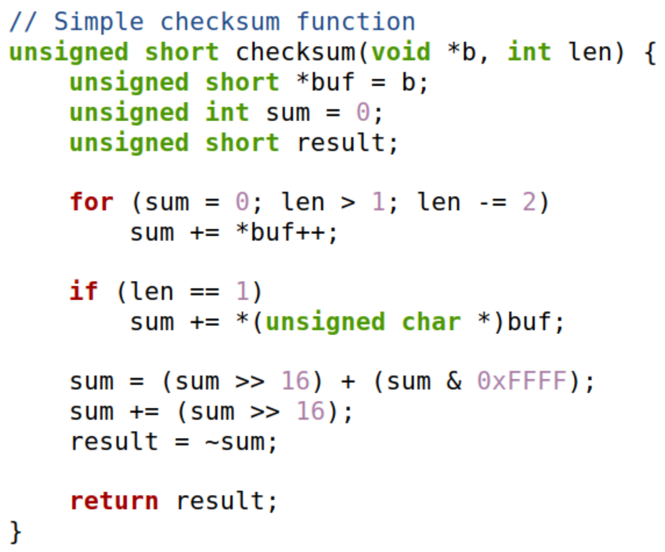
4.2 Task 2.2: Spoofing

Task 2.2A: Write a spoofing program. Please write your own packet spoofing program in C. You need to provide evidence (e.g., Wireshark packet trace) to show that your program successfully sends out spoofed IP packets.

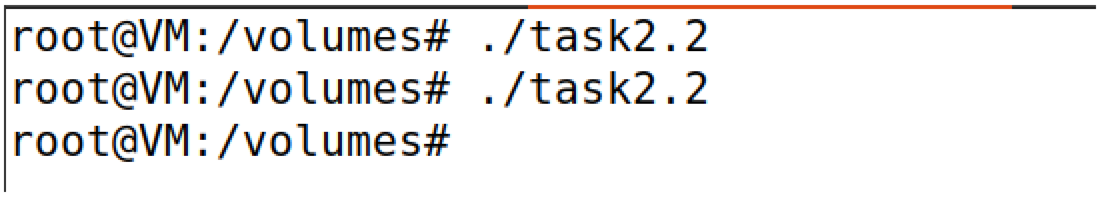


```
void send_raw_ip_packet(struct ipheader* ip) {
        struct sockaddr_in dest_info;
        int enable = 1;
        //Step1: Create a raw network socket
        int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

        //Step2: Set Socket option
        setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

        //Step3: Provide destination information
        dest_info.sin_family = AF_INET;
        dest_info.sin_addr = ip->iph_destip;

        //Step4: Send the packet out
        sendto(sock, ip, ntohs(ip->iph_len),0, (struct sockaddr *)&dest_info, sizeof(dest_info));
        close(sock);
}
```

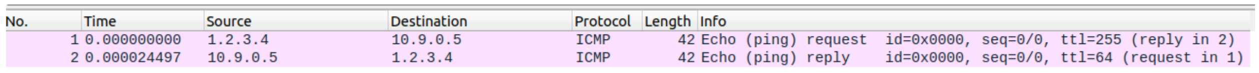
Task 2.2B: Spoof an ICMP Echo Request. Spoof an ICMP echo request packet on behalf of another machine (i.e., using another machine's IP address as its source IP address). This packet should be sent to a remote machine on the Internet (the machine must be alive). You should turn on your Wireshark, so if your spoofing is successful, you can see the echo reply coming back from the remote machine.



```c
// Simple checksum function
unsigned short checksum(void *b, int len) {
    unsigned short *buf = b;
    unsigned int sum = 0;
    unsigned short result;

    for (sum = 0; len > 1; len -= 2)
        sum += *buf++;

    if (len == 1)
        sum += *(unsigned char *)buf;

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    result = ~sum;

    return result;
}
```



```
root@VM:/volumes# ./task2.2
root@VM:/volumes# ./task2.2
root@VM:/volumes#
```



```
No.  Time          Source      Destination   Protocol  Length  Info
   1 0.000000000   1.2.3.4     10.9.0.5      ICMP          42 Echo (ping) request  id=0x0000, seq=0/0, ttl=255 (reply in 2)
   2 0.000024497   10.9.0.5    1.2.3.4       ICMP          42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64 (request in 1)
```

Questions. Please answer the following questions.

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

Answer: Yes it can be set to any arbitrary value but it should not exceed the maximum permissible IP packet size

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

Answer: Yes we calculate the checksum for IP after using the raw socket

```
// Recalculate IP checksum
int sum = 0;
for (int i = 0; i < (new_ip_len / 2); i++) {
    sum += *(unsigned short *)(spoofed_packet + sizeof(struct ether_header) + i * 2);
}
```

• 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: Raw socket allows direct access to network communication by bypassing network stack also ,raw socket enable custom packet crafting. So, for security purposes the non privileged user don't have command to change the fill ed for all this it requires root privilege .It facilitates custom change in field in the packet header.

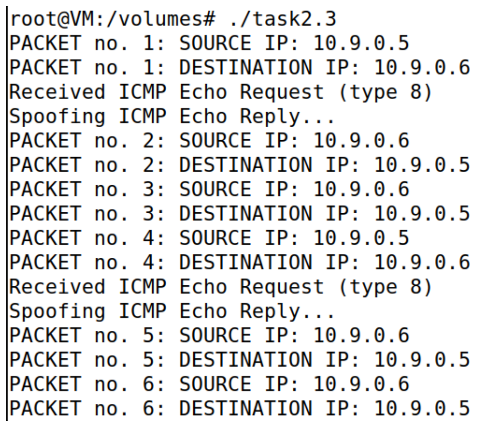
Program fails when we run without root privilege.

4.3 Task 2.3: Sniff and then Spoof

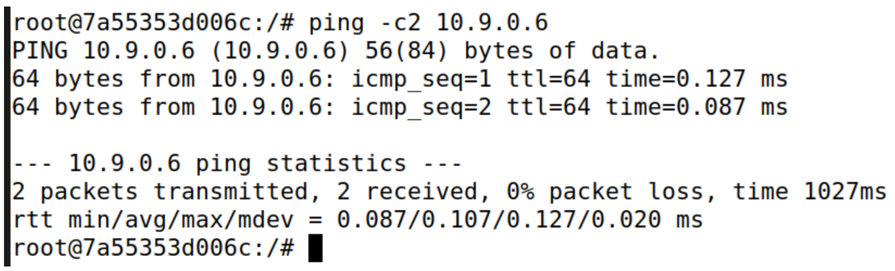
In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and-then-spoof program. You need two machines on the same LAN. From machine A, you ping an IP X. This will generate an ICMP echo request packet. If X is alive, the ping program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on the attacker machine, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IPaddress is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regardless of whether machine X is alive or not, the ping program will always receive a reply, indicating that X is alive. You need to write such a program in C, and include screenshots in your report to show that your program works. Please also attach the code (with adequate amount of comments) in your report.

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

// Structure to hold source and destination IP addresses
struct IpAddresses {
    char source[INET_ADDRSTRLEN];
    char destination[INET_ADDRSTRLEN];
};

// Function to spoof an ICMP echo reply packet
void spoof_icmp_echo_reply(const u_char *packet, int packet_len) {
    // Extract Ethernet header
    struct ether_header *eth_header = (struct ether_header *)packet;

    // Extract IP header
    struct ip *ip_header = (struct ip *)(packet + sizeof(struct ether_header));

    // Calculate the new IP total length
    int new_ip_len = ntohs(ip_header->ip_len);

    // Allocate memory for the spoofed packet
    u_char *spoofed_packet = (u_char *)malloc(packet_len);

    // Copy the original packet to the spoofed packet
    memcpy(spoofed_packet, packet, packet_len);

    // Modify the IP header to swap source and destination
    struct ip *new_ip_header = (struct ip *)(spoofed_packet + sizeof(struct ether_header));
    struct in_addr temp_addr = new_ip_header->ip_src;
    new_ip_header->ip_src = new_ip_header->ip_dst;
    new_ip_header->ip_dst = temp_addr;
    new_ip_header->ip_sum = 0; // Recalculate IP checksum later

    // Modify the ICMP type to echo reply
    u_char *icmp_payload = spoofed_packet + sizeof(struct ether_header) + (ip_header->ip_hl * 4);
    icmp_payload[0] = 0; // ICMP type (echo reply)

    // Recalculate IP checksum
    int sum = 0;
    for (int i = 0; i < (new_ip_len / 2); i++) {
        sum += *(unsigned short *)(spoofed_packet + sizeof(struct ether_header) + i * 2);
    }
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    new_ip_header->ip_sum = ~sum;

    // Send the spoofed packet
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    handle = pcap_open_live("br-70a143fa7795", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device eth0: %sn", errbuf);
        exit(EXIT_FAILURE);
    }
    if (pcap_sendpacket(handle, spoofed_packet, packet_len) != 0) {
        fprintf(stderr, "Error sending packet: %sn", pcap_geterr(handle));
    }
    pcap_close(handle);

    free(spoofed_packet);
}

// Function to process captured packets and extract source and destination IP addresses
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    static int packet_count = 0; // Count packets
```

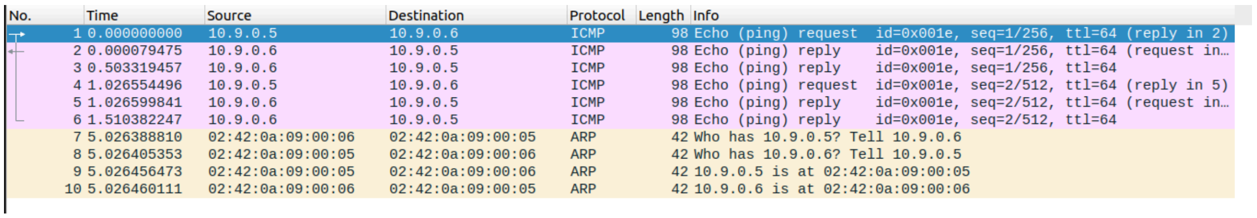
```
  packet_count++;

  // Extract Ethernet header
  struct ether_header *eth_header = (struct ether_header *)packet;
  uint16_t ether_type = ntohs(eth_header->ether_type);

  // Check if the packet is IPv4
  if (ether_type == ETHERTYPE_IP) {
    // Extract IP header
    struct ip *ip_header = (struct ip *)(packet + sizeof(struct ether_header));

    // Copy source and destination IP addresses into the IpAddresses structure
    struct IpAddresses addresses;
    inet_ntop(AF_INET, &(ip_header->ip_src), addresses.source, INET_ADDRSTRLEN);
    inet_ntop(AF_INET, &(ip_header->ip_dst), addresses.destination, INET_ADDRSTRLEN);

    // Print source and destination IP addresses
    printf("PACKET no. %d: SOURCE IP: %sn", packet_count, addresses.source);
    printf("PACKET no. %d: DESTINATION IP: %sn", packet_count, addresses.destination);

    // Check if it's an ICMP echo request (type 8)
    if (ip_header->ip_p == IPPROTO_ICMP) {
      //u_char *icmp_payload = packet + sizeof(struct ether_header) + (ip_header->ip_hl * 4);
      u_char *icmp_payload = (u_char *)(packet + sizeof(struct ether_header) + (ip_header->ip_hl * 4));

      if (icmp_payload[0] == 8) {
        printf("Received ICMP Echo Request (type 8)n");
        printf("Spoofing ICMP Echo Reply...n");
        spoof_icmp_echo_reply(packet, header->caplen);
      }
    }
  } else {
    printf("Packet %d: Not an IPv4 packetn", packet_count);
  }
}

int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;

  // Open live pcap session on NIC with name "eth0"
  handle = pcap_open_live("br-70a143fa7795", BUFSIZ, 1, 1000, errbuf);
  if (handle == NULL) {
    fprintf(stderr, "Couldn't open device eth0: %sn", errbuf);
    return 1;
  }

  // Compile filter_exp into BPF psuedo-code
  char filter_exp[] = "icmp";
  if (pcap_compile(handle, &fp, filter_exp, 0, PCAP_NETMASK_UNKNOWN) == -1) {
    fprintf(stderr, "Couldn't parse filter %s: %sn", filter_exp, pcap_geterr(handle));
    return 1;
  }

  // Apply the compiled filter
  if (pcap_setfilter(handle, &fp) == -1) {
    fprintf(stderr, "Couldn't install filter %s: %sn", filter_exp, pcap_geterr(handle));
    return 1;
  }

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

  // Close the handle
  pcap_close(handle);

  return 0;
}
```



```
root@VM:/volumes# ./task2.3
PACKET no. 1: SOURCE IP: 10.9.0.5
PACKET no. 1: DESTINATION IP: 10.9.0.6
Received ICMP Echo Request (type 8)
Spoofing ICMP Echo Reply...
PACKET no. 2: SOURCE IP: 10.9.0.6
PACKET no. 2: DESTINATION IP: 10.9.0.5
PACKET no. 3: SOURCE IP: 10.9.0.6
PACKET no. 3: DESTINATION IP: 10.9.0.5
PACKET no. 4: SOURCE IP: 10.9.0.5
PACKET no. 4: DESTINATION IP: 10.9.0.6
Received ICMP Echo Request (type 8)
Spoofing ICMP Echo Reply...
PACKET no. 5: SOURCE IP: 10.9.0.6
PACKET no. 5: DESTINATION IP: 10.9.0.5
PACKET no. 6: SOURCE IP: 10.9.0.6
PACKET no. 6: DESTINATION IP: 10.9.0.5
```



```
root@7a55353d006c:/# ping -c2 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=0.127 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.087 ms

--- 10.9.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1027ms
rtt min/avg/max/mdev = 0.087/0.107/0.127/0.020 ms
root@7a55353d006c:/# 
```



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x001e, seq=1/256, ttl=64 (reply in 2)
2	0.000079475	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x001e, seq=1/256, ttl=64 (request in…
3	0.503319457	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x001e, seq=1/256, ttl=64
4	1.026554496	10.9.0.5	10.9.0.6	ICMP	98	Echo (ping) request id=0x001e, seq=2/512, ttl=64 (reply in 5)
5	1.026599841	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x001e, seq=2/512, ttl=64 (request in…
6	1.510382247	10.9.0.6	10.9.0.5	ICMP	98	Echo (ping) reply id=0x001e, seq=2/512, ttl=64
7	5.026388810	02:42:0a:09:00:06	02:42:0a:09:00:05	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
8	5.026405353	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
9	5.026456473	02:42:0a:09:00:05	02:42:0a:09:00:06	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
10	5.026460111	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

Explanation: when we execute the above program ,the Network interface card capture all the packets that arrive at the program and it process each packet by modification and create new packet and modification as source as destination and destination as source included in the created packet and then sends it and victim received