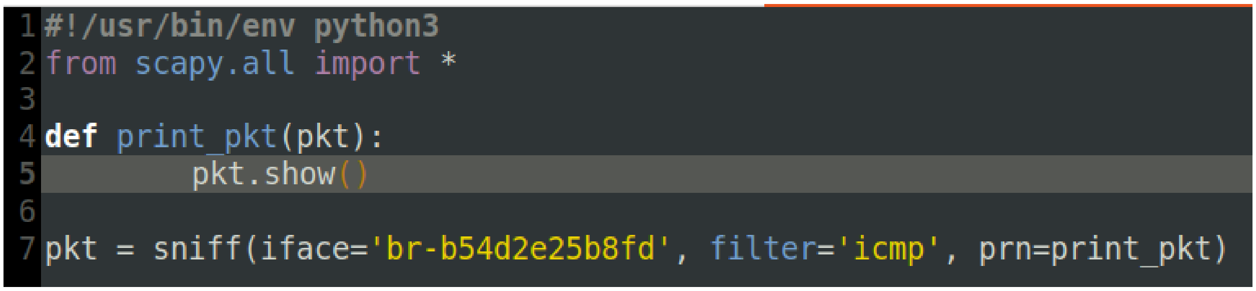
John Sikes

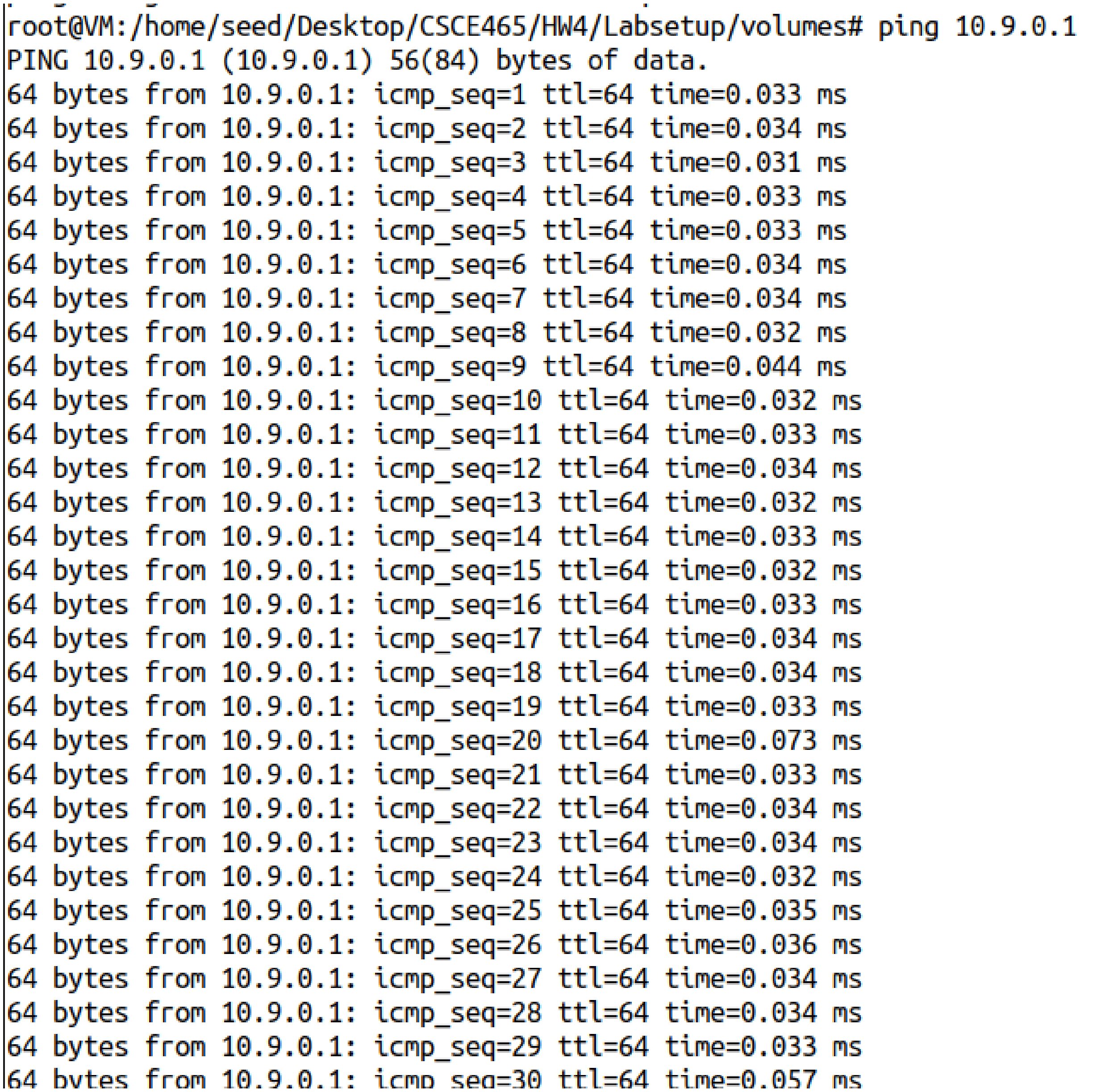
CSCE 465 - HW 4

Note: Much of the code designed and used in this homework was referenced from *Computer & Internet Security: A Hands-on Approach* textbook, as well as the homework manual itself.

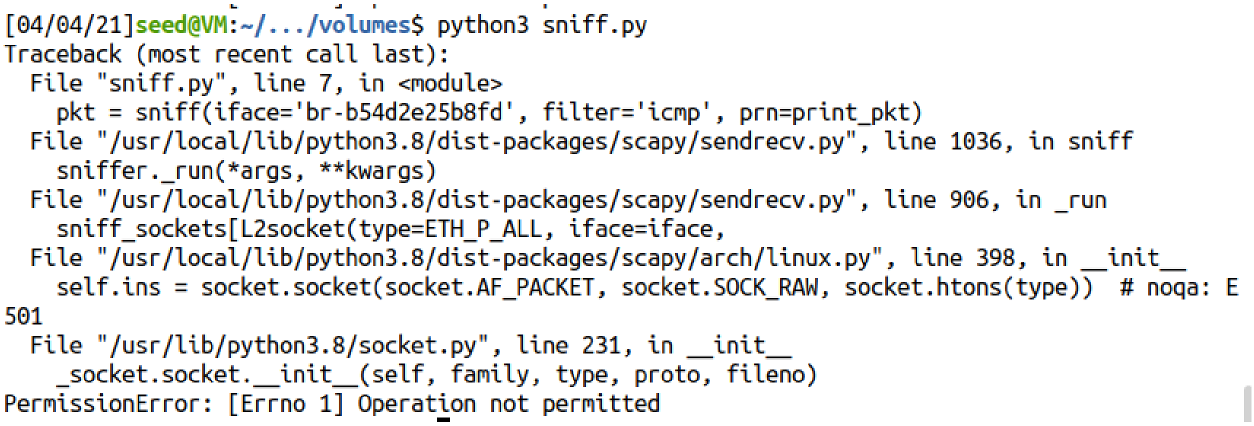
Task 1.1A:



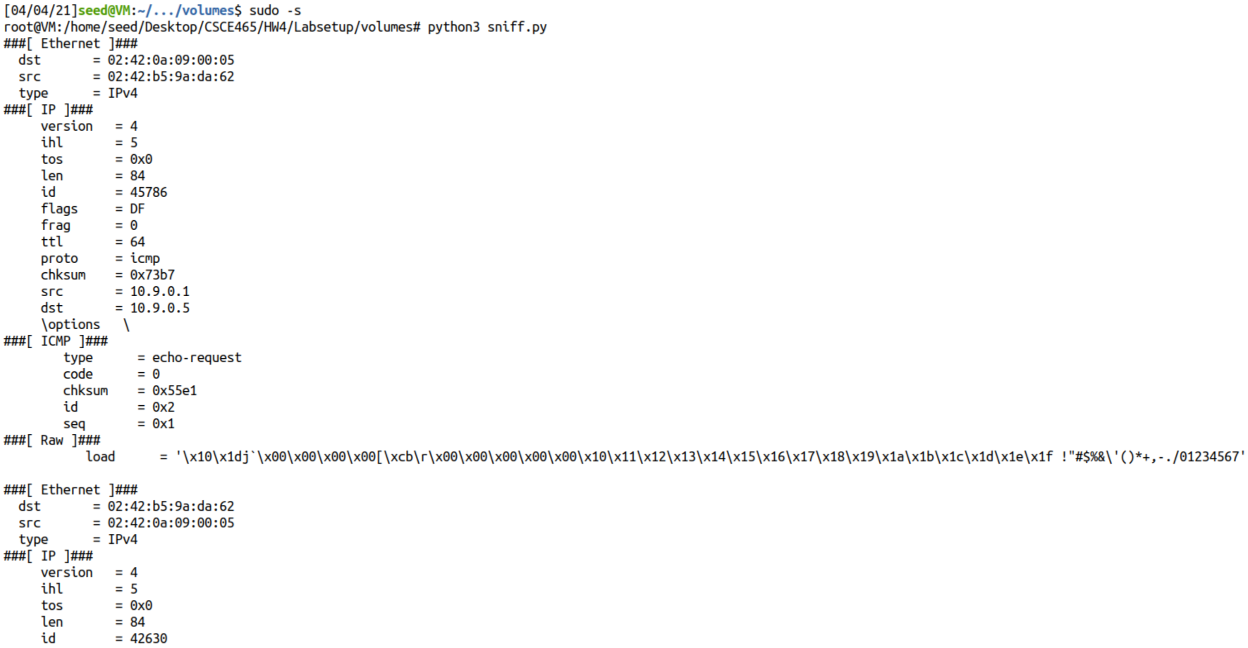
```
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-b54d2e25b8fd', filter='icmp', prn=print_pkt)
```



```
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.033 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.031 ms
64 bytes from 10.9.0.1: icmp_seq=4 ttl=64 time=0.033 ms
64 bytes from 10.9.0.1: icmp_seq=5 ttl=64 time=0.033 ms
64 bytes from 10.9.0.1: icmp_seq=6 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=7 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=8 ttl=64 time=0.032 ms
64 bytes from 10.9.0.1: icmp_seq=9 ttl=64 time=0.044 ms
64 bytes from 10.9.0.1: icmp_seq=10 ttl=64 time=0.032 ms
64 bytes from 10.9.0.1: icmp_seq=11 ttl=64 time=0.033 ms
64 bytes from 10.9.0.1: icmp_seq=12 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=13 ttl=64 time=0.032 ms
64 bytes from 10.9.0.1: icmp_seq=14 ttl=64 time=0.033 ms
64 bytes from 10.9.0.1: icmp_seq=15 ttl=64 time=0.032 ms
64 bytes from 10.9.0.1: icmp_seq=16 ttl=64 time=0.033 ms
64 bytes from 10.9.0.1: icmp_seq=17 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=18 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=19 ttl=64 time=0.033 ms
64 bytes from 10.9.0.1: icmp_seq=20 ttl=64 time=0.073 ms
64 bytes from 10.9.0.1: icmp_seq=21 ttl=64 time=0.033 ms
64 bytes from 10.9.0.1: icmp_seq=22 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=23 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=24 ttl=64 time=0.032 ms
64 bytes from 10.9.0.1: icmp_seq=25 ttl=64 time=0.035 ms
64 bytes from 10.9.0.1: icmp_seq=26 ttl=64 time=0.036 ms
64 bytes from 10.9.0.1: icmp_seq=27 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=28 ttl=64 time=0.034 ms
64 bytes from 10.9.0.1: icmp_seq=29 ttl=64 time=0.033 ms
64 bytes from 10.9.0.1: icmp_seq=30 ttl=64 time=0.057 ms
```



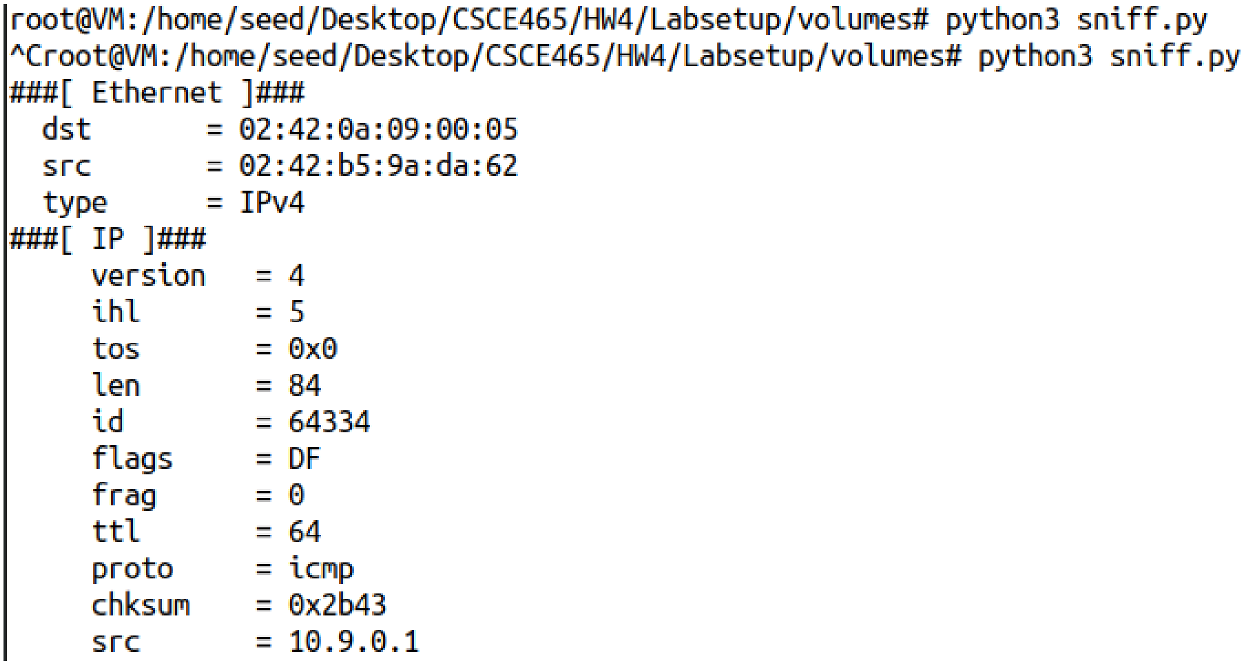
```
[04/04/21]seed@VM:~/.../volumes$ python3 sniff.py
Traceback (most recent call last):
  File "sniff.py", line 7, in <module>
    pkt = sniff(iface='br-b54d2e25b8fd', filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 906, in _run
    sniff_sockets[L2socket(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  # noqa: E
501
  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
```



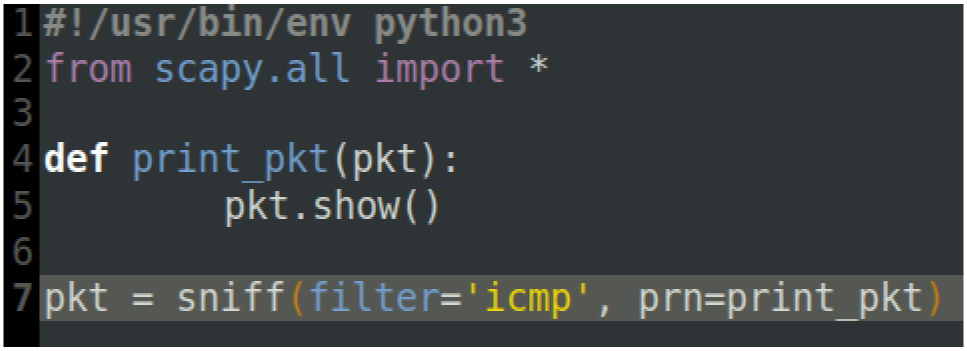
```
[04/04/21]seed@VM:~/.../volumes$ sudo -s
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# python3 sniff.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:b5:9a:da:62
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 45786
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x73b7
     src       = 10.9.0.1
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x55e1
        id        = 0x2
        seq       = 0x1
###[ Raw ]###
           load      = '\x10\x1dj`\x00\x00\x00\x00[\xcb\r\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 02:42:b5:9a:da:62
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 42630
```

In this task, I first ran the provided code in both with and without root privilege. When using root privilege, the code executes perfectly fine and packets are captured accordingly. However, when executed through the seed user the code doesn't execute due to it not having the permissions necessary to use raw sockets (which scapy utilizes in this case).

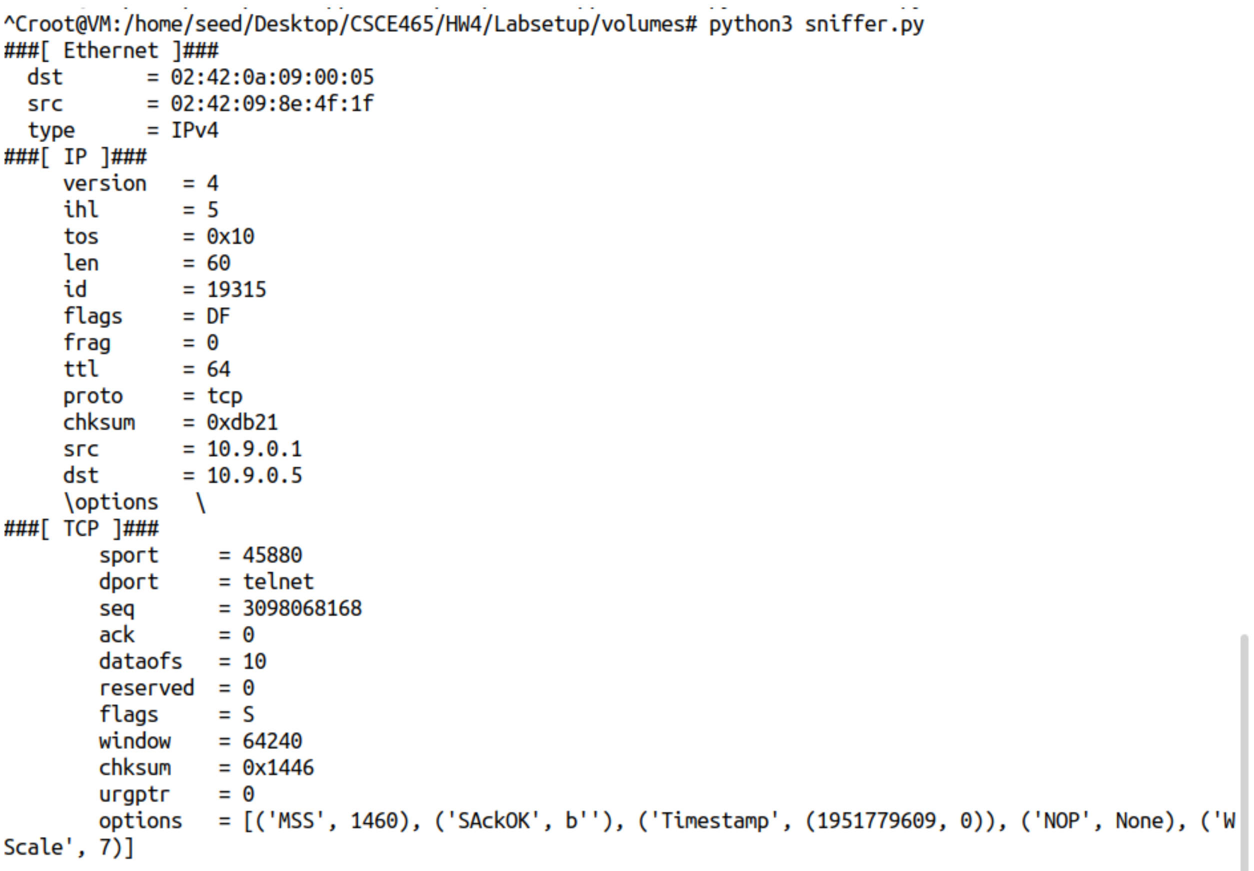
Task 1.1B:



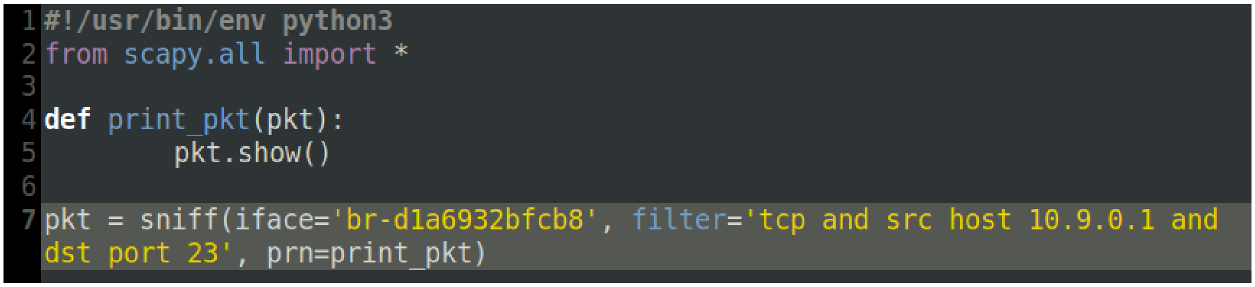
```
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# python3 sniff.py
^Croot@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# python3 sniff.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:b5:9a:da:62
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 64334
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x2b43
     src       = 10.9.0.1
```



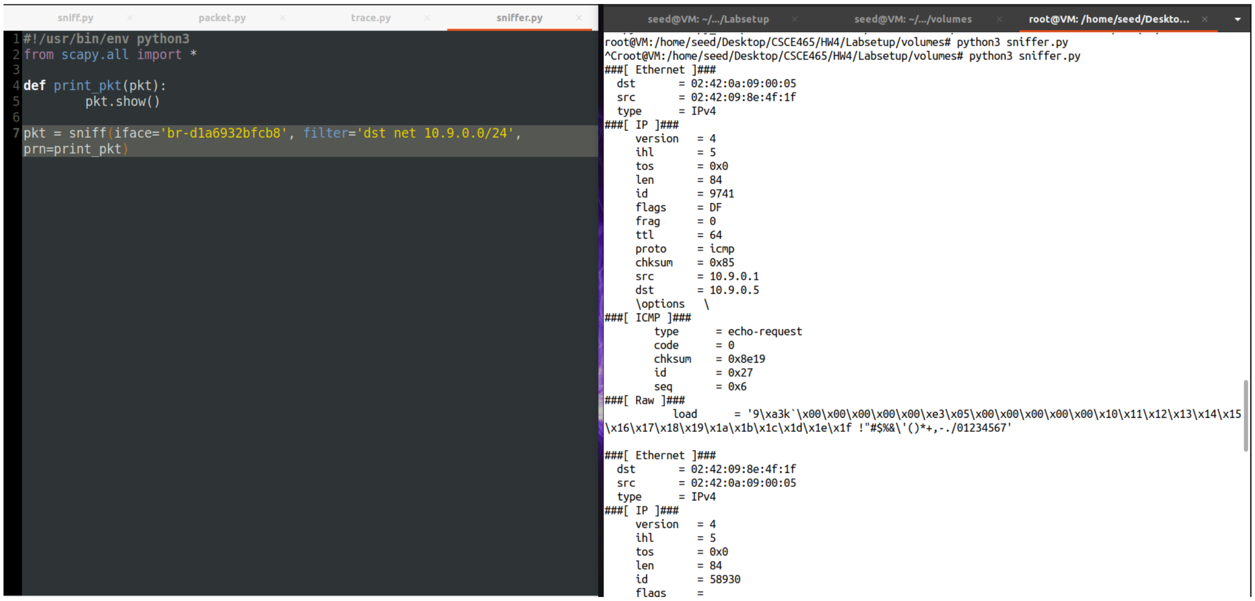
```python
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
           pkt.show()

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



```
^Croot@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# python3 sniffer.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:09:8e:4f:1f
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 19315
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xdb21
     src       = 10.9.0.1
     dst       = 10.9.0.5
     \options   \
###[ TCP ]###
        sport     = 45880
        dport     = telnet
        seq       = 3098068168
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 64240
        chksum    = 0x1446
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (1951779609, 0)), ('NOP', None), ('W
Scale', 7)]
```



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

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-d1a6932bfcb8', filter='tcp and src host 10.9.0.1 and dst port 23', prn=print_pkt)
```

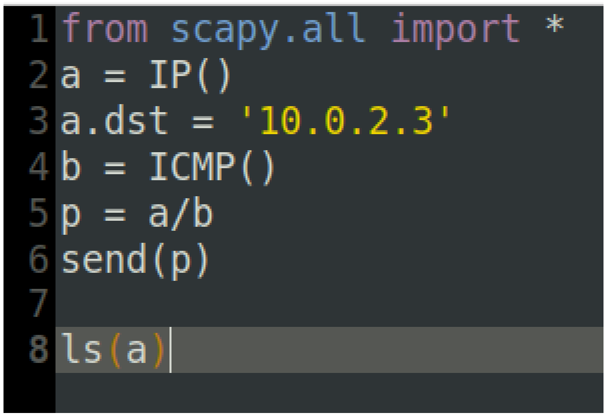


```python
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-d1a6932bfcb8', filter='dst net 10.9.0.0/24',
prn=print_pkt)
```

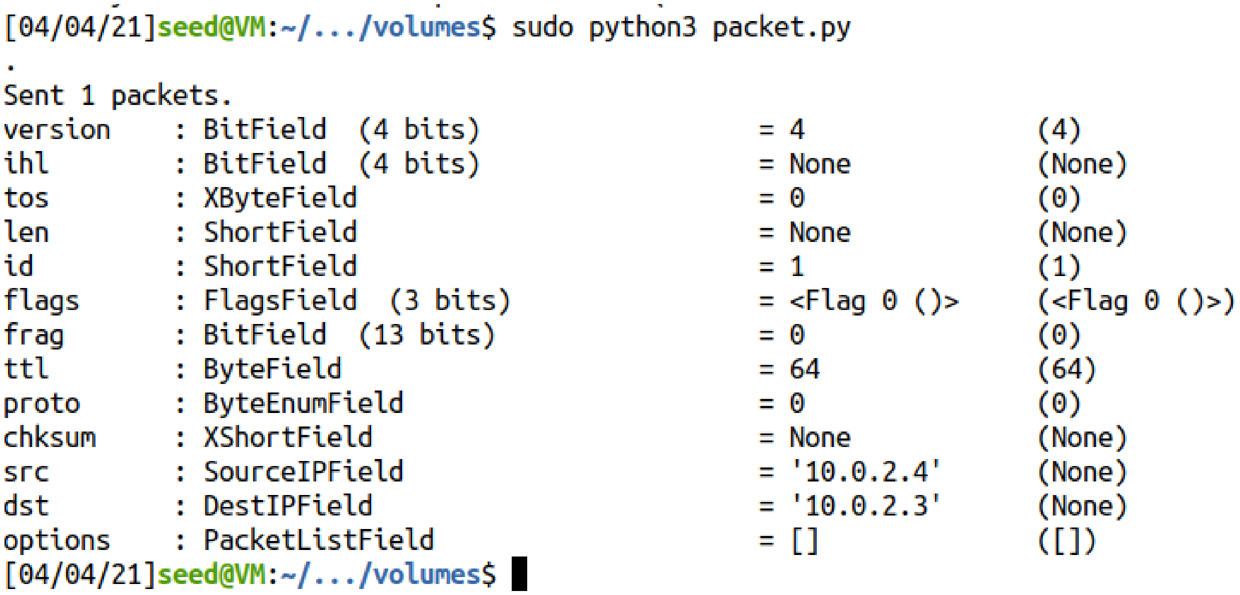
```
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# python3 sniffer.py
^Croot@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# python3 sniffer.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:09:8e:4f:1f
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 9741
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x85
     src       = 10.9.0.1
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x8e19
        id        = 0x27
        seq       = 0x6
###[ Raw ]###
           load      = '9\xa3k`\x00\x00\x00\x00\x00\xe3\x05\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15
\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 02:42:09:8e:4f:1f
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 58930
     flags     =
```

In this part of the task I filtered captured packets by the ICMP protocol, TCP packets from a particular IP (10.9.0.1) with port 23, and packets from a particular subnet (10.9.0.0/24), with the code and output of such included in order.

Task 1.2:



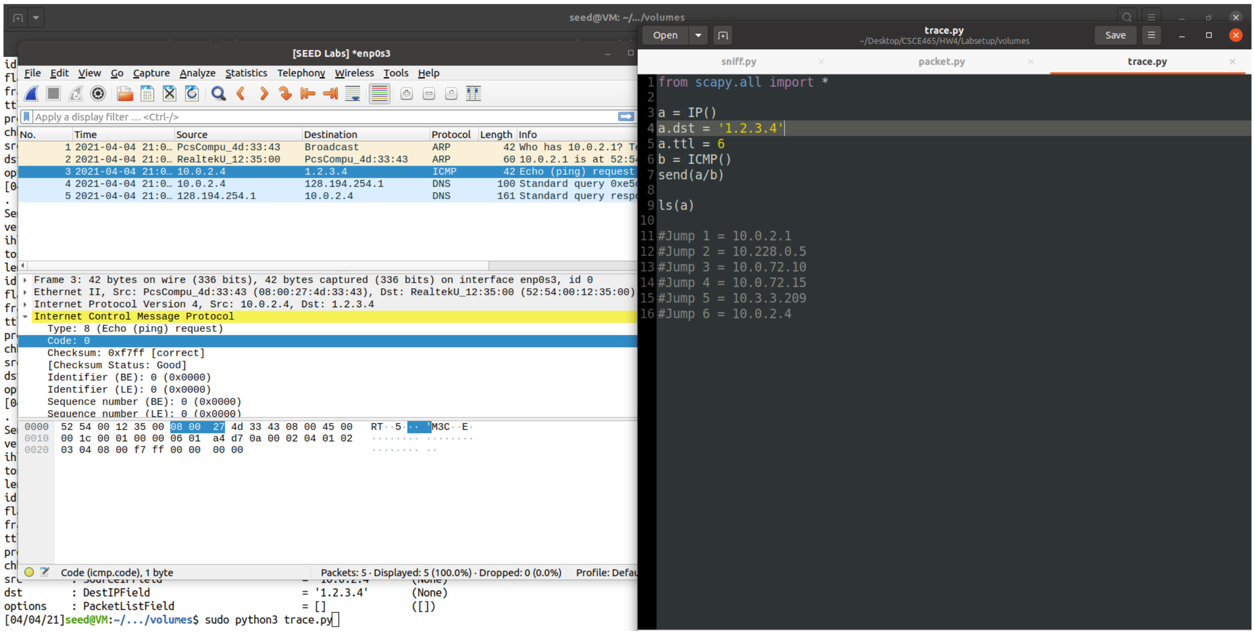
```
from scapy.all import *
a = IP()
a.dst = '10.0.2.3'
b = ICMP()
p = a/b
send(p)

ls(a)
```



```
[04/04/21]seed@VM:~/.../volumes$ sudo python3 packet.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                       = '10.0.2.4'     (None)
dst        : DestIPField                         = '10.0.2.3'     (None)
options    : PacketListField                     = []             ([])
[04/04/21]seed@VM:~/.../volumes$
```

This task required me to use scapy to send spoofed packets to an arbitrary address using provided code in the manual, with code and results included as images above showing the requirements met.

Task 1.3:


seed@VM: ~/.../volumes
[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Open
trace.py
~/Desktop/CSCE465/HW4/Labsetup/volumes
Save
sniff.py
packet.py
trace.py

```
No. Time            Source              Destination         Protocol Length Info
  1 2021-04-04 21:0… PcsCompu_4d:33:43  Broadcast           ARP          42 Who has 10.0.2.1? Te
  2 2021-04-04 21:0… RealtekU_12:35:00  PcsCompu_4d:33:43   ARP          60 10.0.2.1 is at 52:54
  3 2021-04-04 21:0… 10.0.2.4           1.2.3.4             ICMP         42 Echo (ping) request
  4 2021-04-04 21:0… 10.0.2.4           128.194.254.1       DNS         100 Standard query 0xe5c
  5 2021-04-04 21:0… 128.194.254.1      10.0.2.4            DNS         161 Standard query respo
```

```
▸ Frame 3: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PcsCompu_4d:33:43 (08:00:27:4d:33:43), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
▸ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 1.2.3.4
▾ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0xf7ff [correct]
    [Checksum Status: Good]
    Identifier (BE): 0 (0x0000)
    Identifier (LE): 0 (0x0000)
    Sequence number (BE): 0 (0x0000)
    Sequence number (LE): 0 (0x0000)
```

```
0000  52 54 00 12 35 00 08 00  27 4d 33 43 08 00 45 00   RT··5···'M3C··E·
0010  00 1c 00 01 00 00 06 01  a4 d7 0a 00 02 04 01 02   ········ ········
0020  03 04 08 00 f7 ff 00 00  00 00                     ········ ··
```

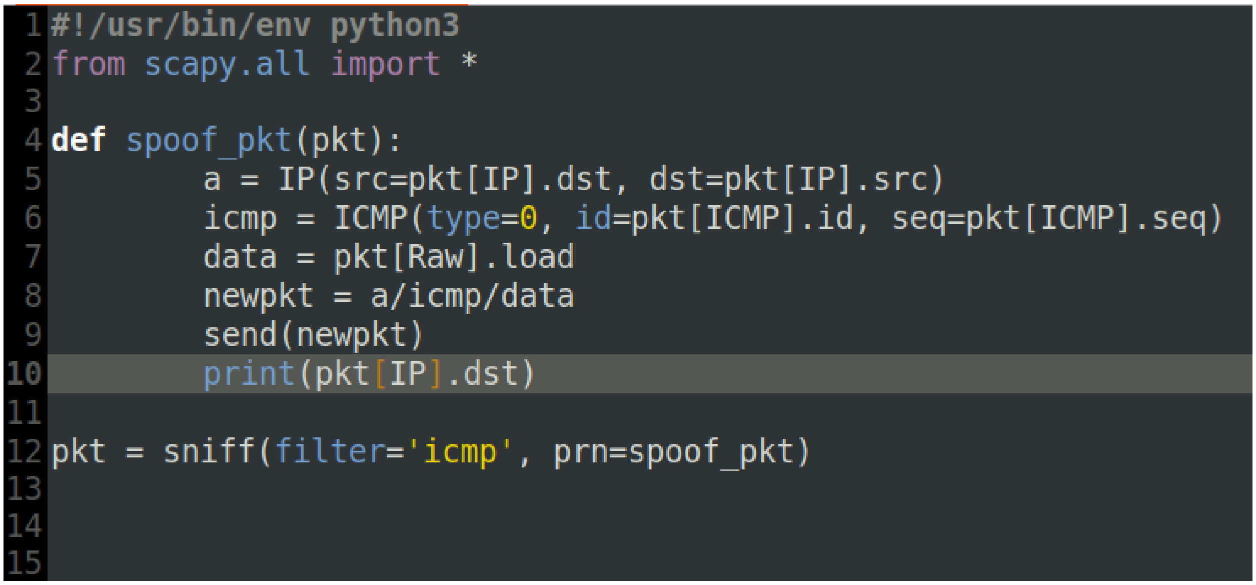
Code (icmp.code), 1 byte   Packets: 5 · Displayed: 5 (100.0%) · Dropped: 0 (0.0%)   Profile: Defau

```python
from scapy.all import *

a = IP()
a.dst = '1.2.3.4'
a.ttl = 6
b = ICMP()
send(a/b)

ls(a)

#Jump 1 = 10.0.2.1
#Jump 2 = 10.228.0.5
#Jump 3 = 10.0.72.10
#Jump 4 = 10.0.72.15
#Jump 5 = 10.3.3.209
#Jump 6 = 10.0.2.4
```

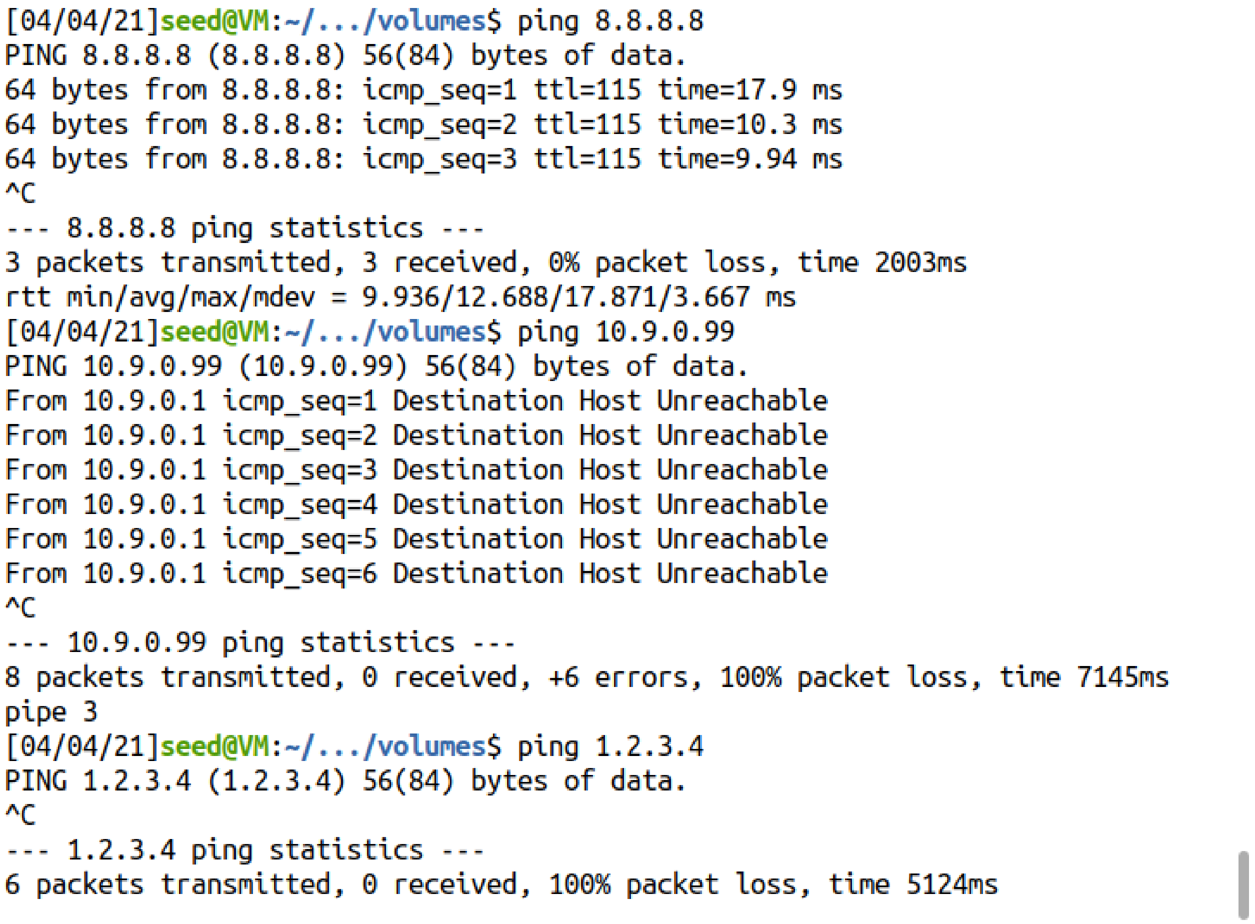
```
dst        : DestIPField                         = '1.2.3.4'      (None)
options    : PacketListField                     = []             ([])
[04/04/21]seed@VM:~/.../volumes$ sudo python3 trace.py
```

In this task I was required to estimate the distance, in terms of routers, between my VM and a particular destination IP (1.2.3.4). To do so, I modified the ttl value of the IP until I got a successful ICMP transmission, rather than error messages, inside of WireShark. To this extent, I incremented my ttl value and recorded the IP addresses that it reached in each iteration (seen as comments in the code above) until the ICMP packet eventually was delivered, which occurred on the 6th iteration for me.

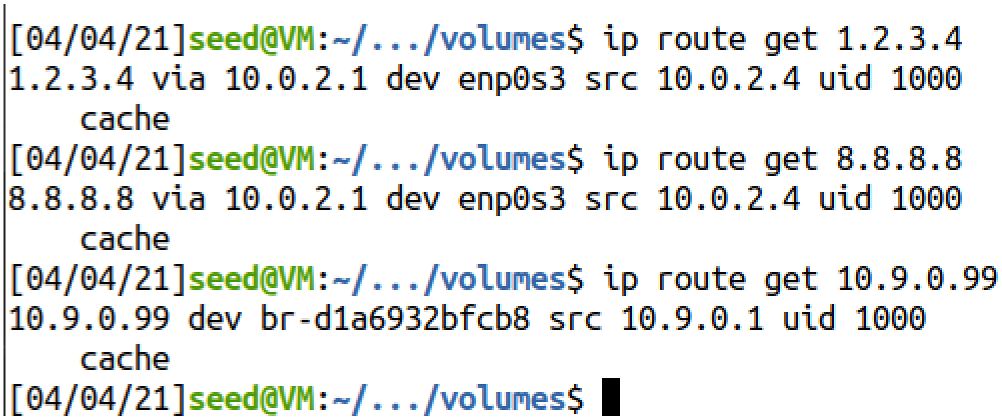
Task 1.4:



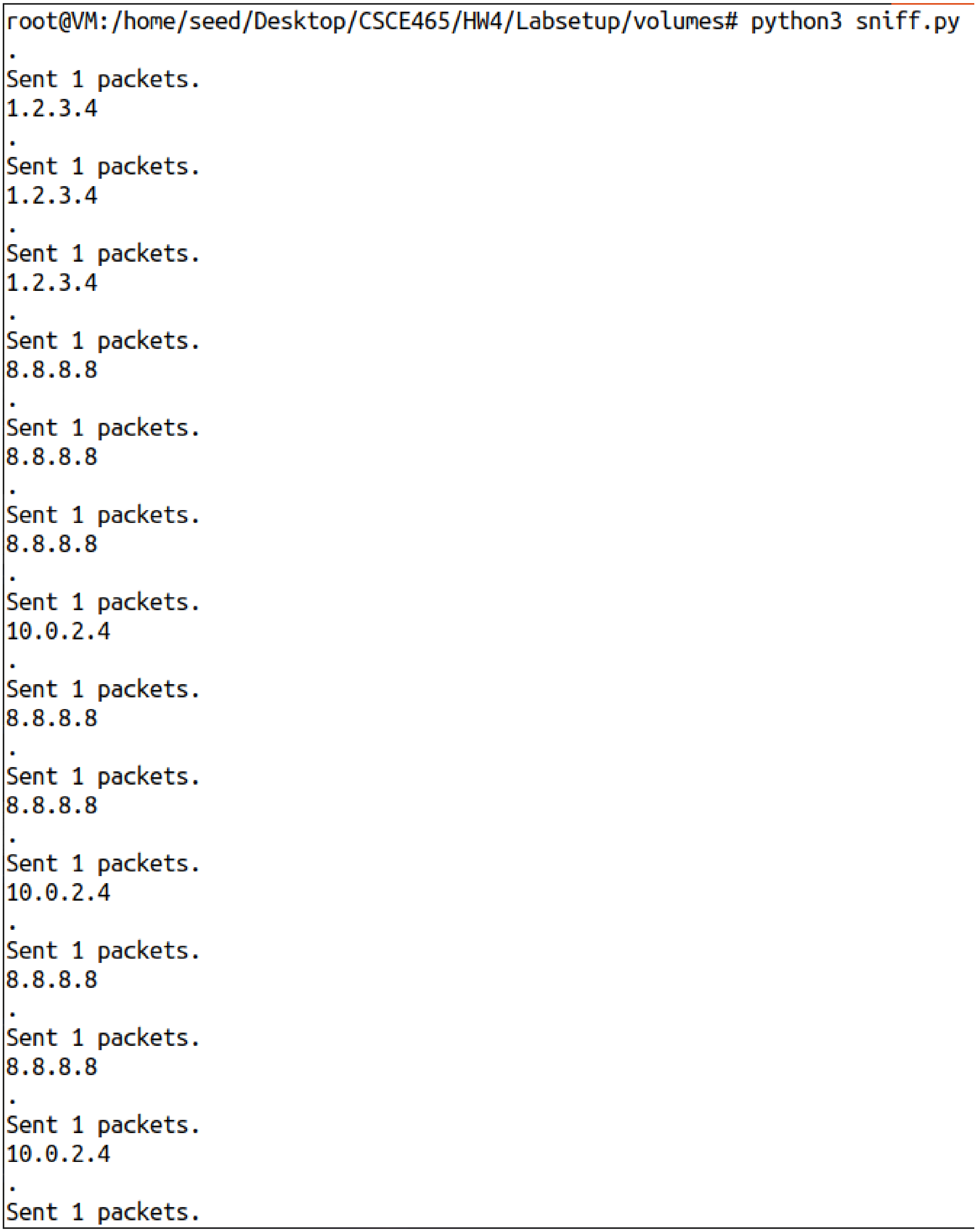
```python
#!/usr/bin/env python3
from scapy.all import *

def spoof_pkt(pkt):
        a = IP(src=pkt[IP].dst, dst=pkt[IP].src)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = a/icmp/data
        send(newpkt)
        print(pkt[IP].dst)

pkt = sniff(filter='icmp', prn=spoof_pkt)



```



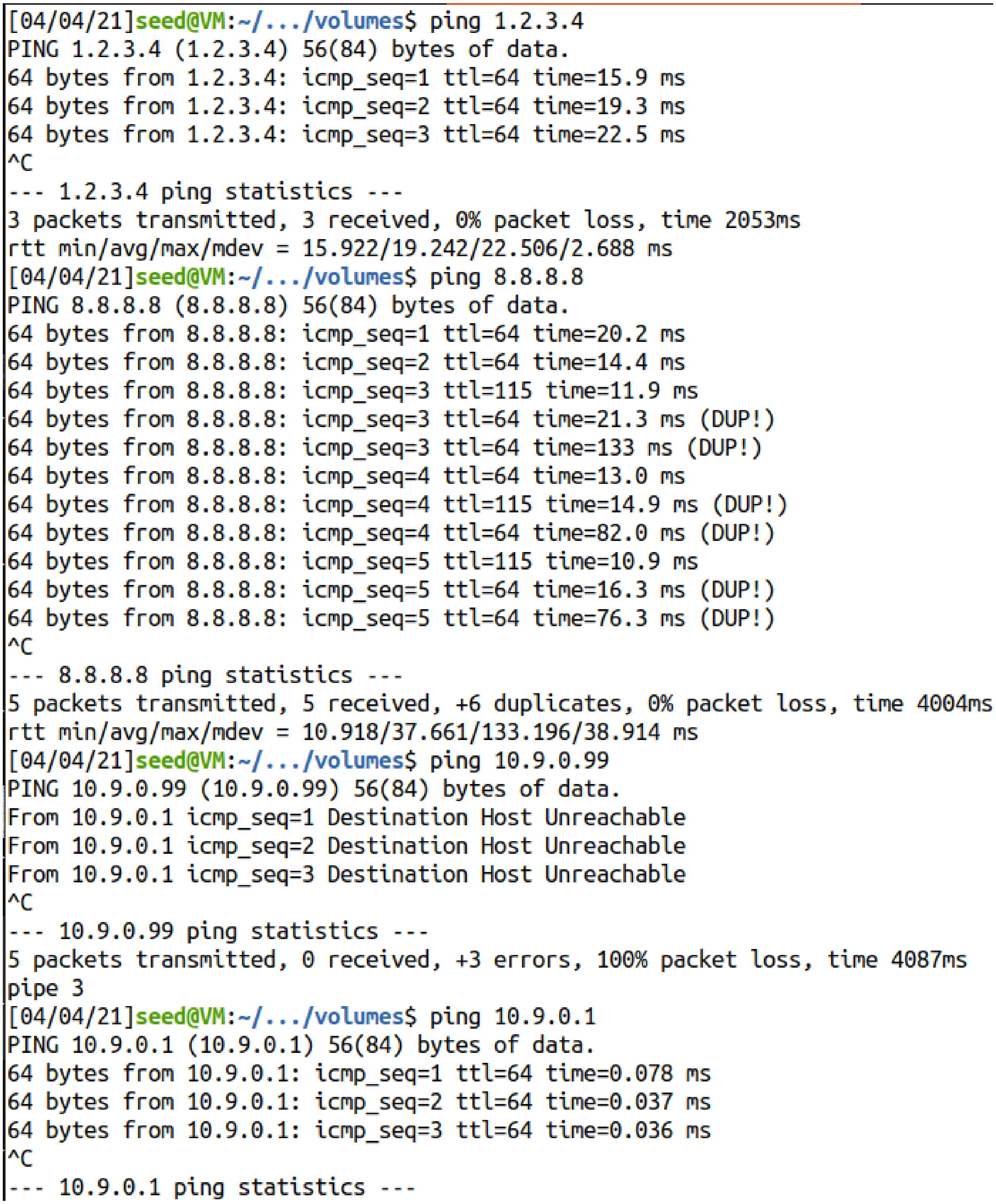
```
[04/04/21]seed@VM:~/.../volumes$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=115 time=17.9 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=115 time=10.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=115 time=9.94 ms
^C
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 9.936/12.688/17.871/3.667 ms
[04/04/21]seed@VM:~/.../volumes$ ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.1 icmp_seq=1 Destination Host Unreachable
From 10.9.0.1 icmp_seq=2 Destination Host Unreachable
From 10.9.0.1 icmp_seq=3 Destination Host Unreachable
From 10.9.0.1 icmp_seq=4 Destination Host Unreachable
From 10.9.0.1 icmp_seq=5 Destination Host Unreachable
From 10.9.0.1 icmp_seq=6 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
8 packets transmitted, 0 received, +6 errors, 100% packet loss, time 7145ms
pipe 3
[04/04/21]seed@VM:~/.../volumes$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
^C
--- 1.2.3.4 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5124ms
```



```
[04/04/21]seed@VM:~/.../volumes$ ip route get 1.2.3.4
1.2.3.4 via 10.0.2.1 dev enp0s3 src 10.0.2.4 uid 1000
    cache
[04/04/21]seed@VM:~/.../volumes$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3 src 10.0.2.4 uid 1000
    cache
[04/04/21]seed@VM:~/.../volumes$ ip route get 10.9.0.99
10.9.0.99 dev br-d1a6932bfcb8 src 10.9.0.1 uid 1000
    cache
[04/04/21]seed@VM:~/.../volumes$ 
```



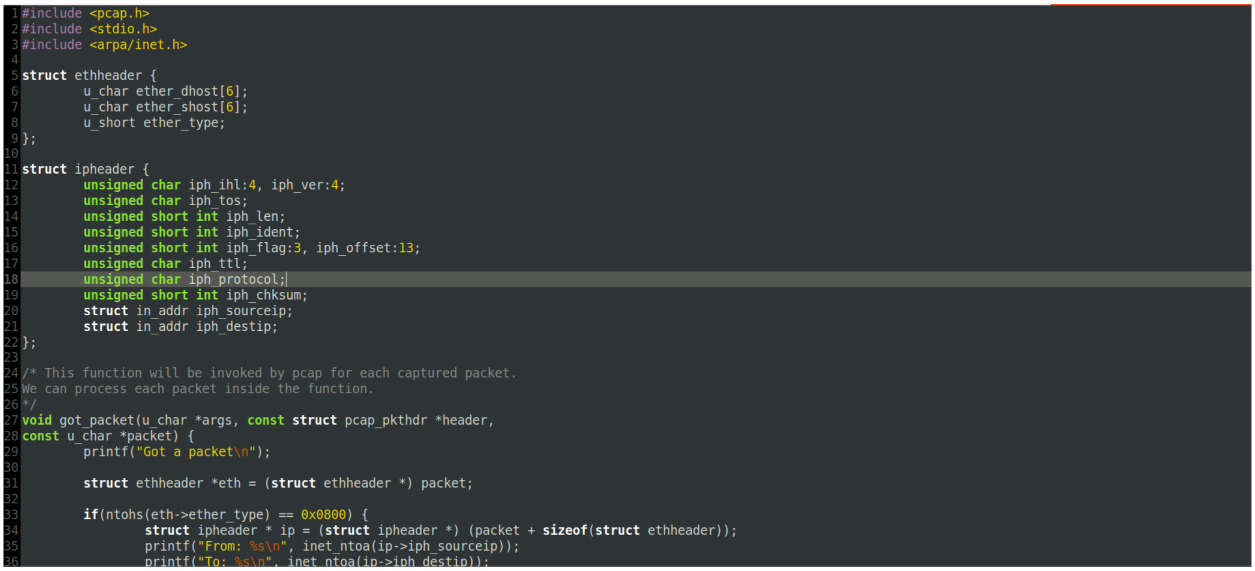
```
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# python3 sniff.py
.
Sent 1 packets.
1.2.3.4
.
Sent 1 packets.
1.2.3.4
.
Sent 1 packets.
1.2.3.4
.
Sent 1 packets.
8.8.8.8
.
Sent 1 packets.
8.8.8.8
.
Sent 1 packets.
8.8.8.8
.
Sent 1 packets.
10.0.2.4
.
Sent 1 packets.
8.8.8.8
.
Sent 1 packets.
8.8.8.8
.
Sent 1 packets.
10.0.2.4
.
Sent 1 packets.
8.8.8.8
.
Sent 1 packets.
8.8.8.8
.
Sent 1 packets.
10.0.2.4
.
Sent 1 packets.
```



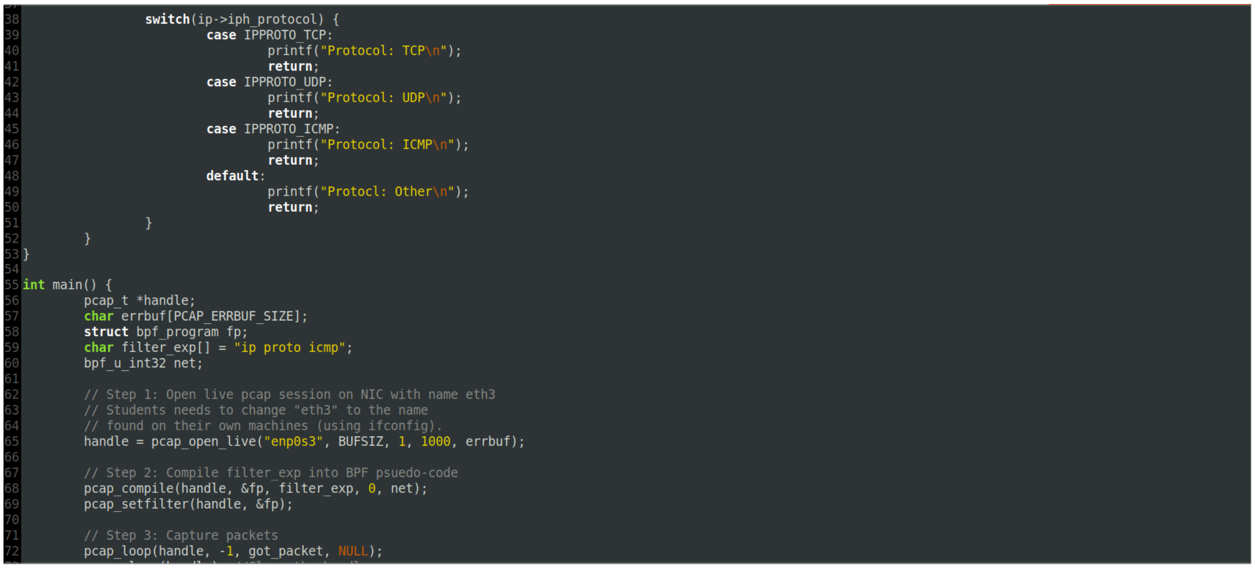
```
[04/04/21]seed@VM:~/.../volumes$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=15.9 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=19.3 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=22.5 ms
^C
--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 15.922/19.242/22.506/2.688 ms
[04/04/21]seed@VM:~/.../volumes$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=20.2 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=14.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=115 time=11.9 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=21.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=133 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=64 time=13.0 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=115 time=14.9 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=64 time=82.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=5 ttl=115 time=10.9 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=64 time=16.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=5 ttl=64 time=76.3 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, +6 duplicates, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 10.918/37.661/133.196/38.914 ms
[04/04/21]seed@VM:~/.../volumes$ ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.1 icmp_seq=1 Destination Host Unreachable
From 10.9.0.1 icmp_seq=2 Destination Host Unreachable
From 10.9.0.1 icmp_seq=3 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
5 packets transmitted, 0 received, +3 errors, 100% packet loss, time 4087ms
pipe 3
[04/04/21]seed@VM:~/.../volumes$ ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.078 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.037 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.036 ms
^C
--- 10.9.0.1 ping statistics ---
```

This task sought to combine the code and capabilities from the previous tasks in order to design a program that would first sniff out packets, then spoof responses from each destination whether a response would occur naturally or not. To this extent, I first show the code I designed for this task, followed by the results of the routes I tested. These results indicate that 8.8.8.8 could receive and respond to ICMP packets being sent, 1.2.3.4 would result in 100% packet loss due to the packets not being able to be delivered to a non-existent internet host, and 10.9.0.99 responding with “Destination Host Unreachable” due to it being both a non-existent address but also on the LAN, as valid LAN hosts are capable of pinging and responding accordingly. Similarly, the routes of these respectively show that 1.2.3.4 and 8.8.8.8 are on the “enp0s3” interface while 10.9.0.99 is on the “br-d1a…” interface associated with the LAN created by the container. Knowing these details, I modified the code such that the spoof_pkt function would be called by the sniffer and subsequently send out the spoofed packet after the sniff was called on the according packet. The results of running this are indicated in the following images, showing the packets being sent by the sniff/spoof program and then the received responses from the pings that previously hadn’t worked (as well as that which did). However, 10.9.0.99 still was unable to receive packets in response due to its nature being on the LAN, while 10.9.0.1 was able to receive responses due to it being a live host on the LAN.

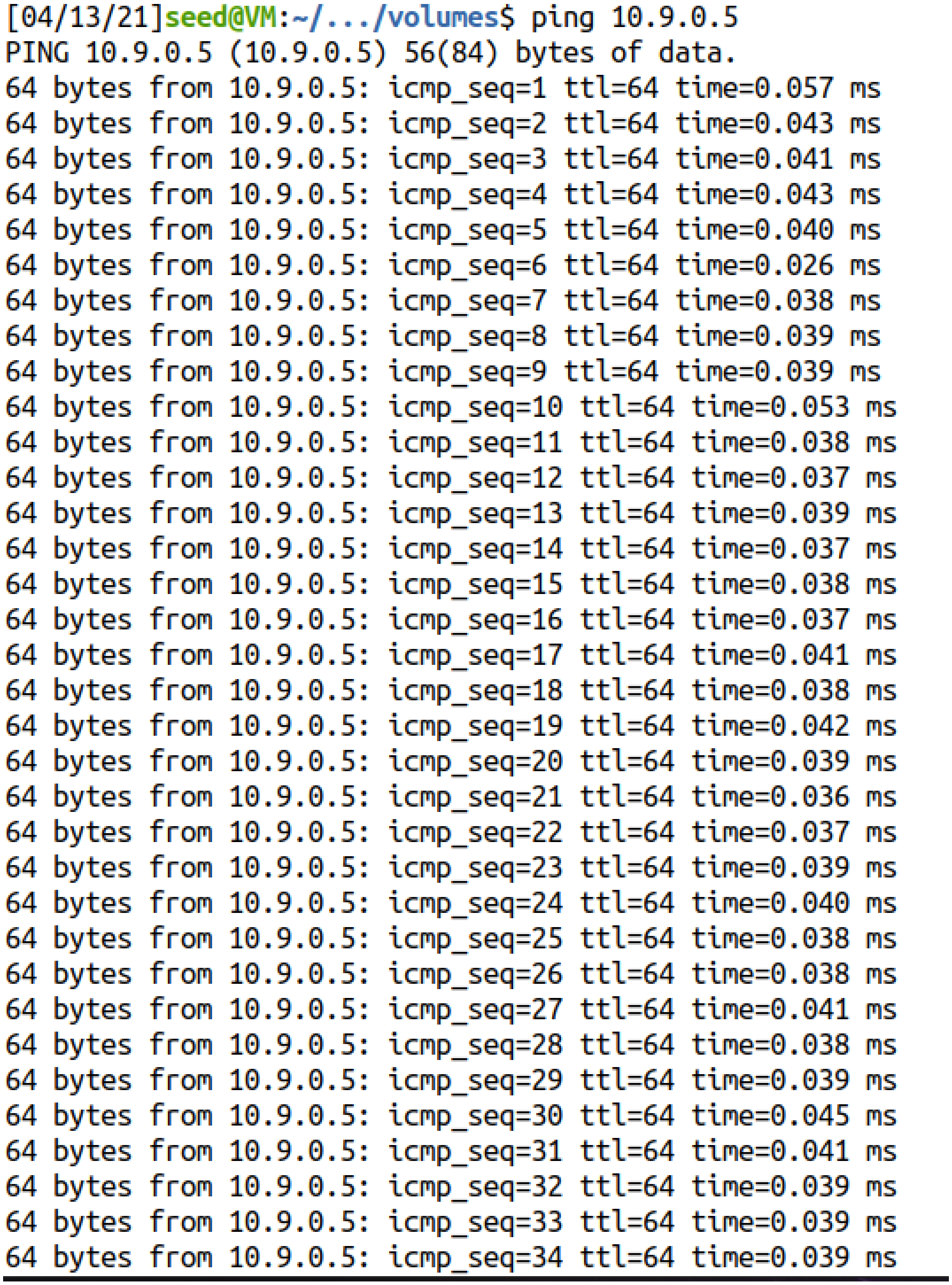
Task 2.1A:



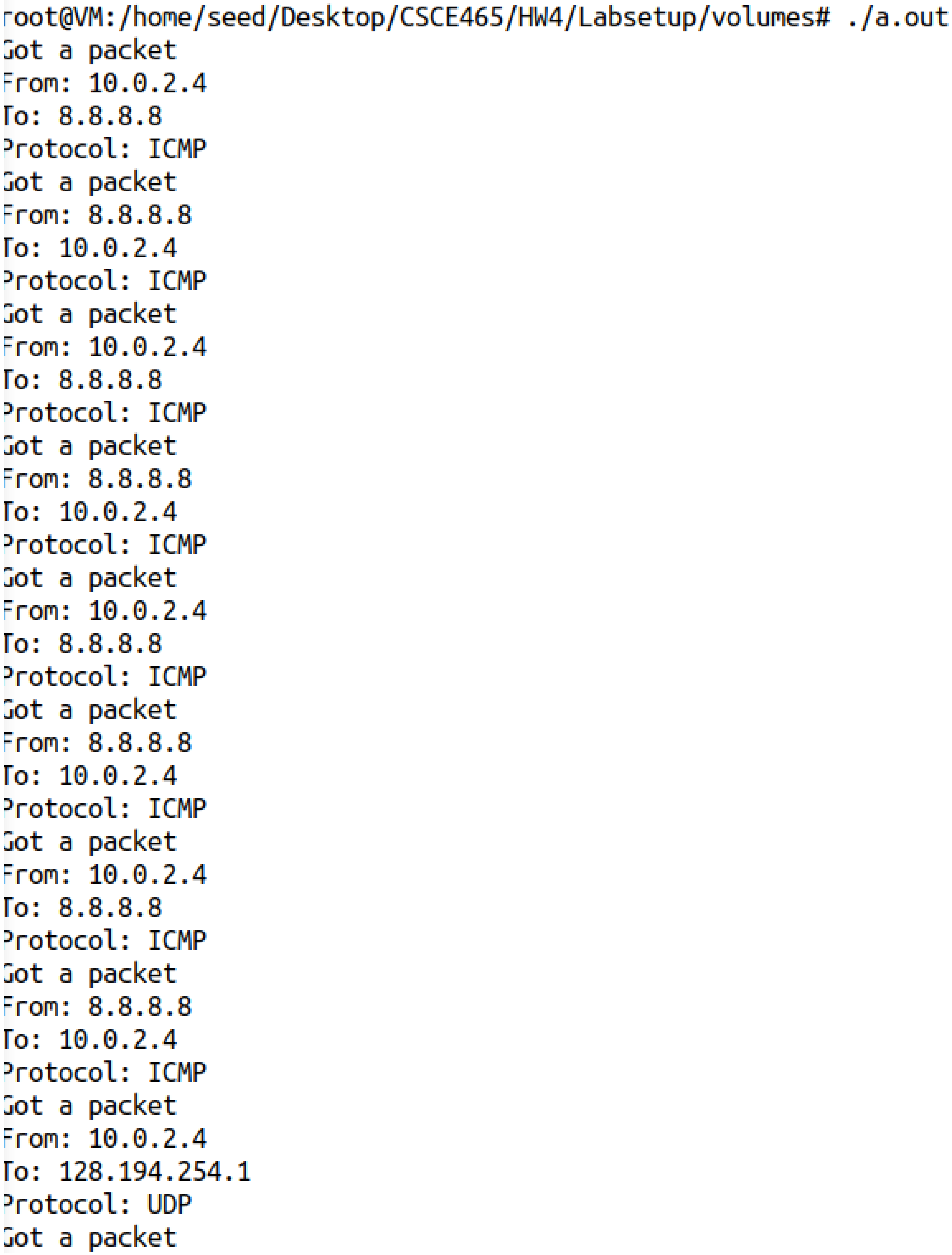
```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>

struct ethheader {
        u_char ether_dhost[6];
        u_char ether_shost[6];
        u_short ether_type;
};

struct ipheader {
        unsigned char iph_ihl:4, iph_ver:4;
        unsigned char iph_tos;
        unsigned short int iph_len;
        unsigned short int iph_ident;
        unsigned short int iph_flag:3, iph_offset:13;
        unsigned char iph_ttl;
        unsigned char iph_protocol;
        unsigned short int iph_chksum;
        struct in_addr iph_sourceip;
        struct in_addr iph_destip;
};

/* This function will be invoked by pcap for each captured packet.
We can process each packet inside the function.
*/
void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet) {
        printf("Got a packet\n");

        struct ethheader *eth = (struct ethheader *) packet;

        if(ntohs(eth->ether_type) == 0x0800) {
                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));
```



```
                switch(ip->iph_protocol) {
                        case IPPROTO_TCP:
                                printf("Protocol: TCP\n");
                                return;
                        case IPPROTO_UDP:
                                printf("Protocol: UDP\n");
                                return;
                        case IPPROTO_ICMP:
                                printf("Protocol: ICMP\n");
                                return;
                        default:
                                printf("Protocl: Other\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 eth3
        // Students needs to change "eth3" to the name
        // found on their own machines (using ifconfig).
        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);
```



```
[04/13/21]seed@VM:~/.../volumes$ ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.057 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.043 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.041 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.043 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=64 time=0.040 ms
64 bytes from 10.9.0.5: icmp_seq=6 ttl=64 time=0.026 ms
64 bytes from 10.9.0.5: icmp_seq=7 ttl=64 time=0.038 ms
64 bytes from 10.9.0.5: icmp_seq=8 ttl=64 time=0.039 ms
64 bytes from 10.9.0.5: icmp_seq=9 ttl=64 time=0.039 ms
64 bytes from 10.9.0.5: icmp_seq=10 ttl=64 time=0.053 ms
64 bytes from 10.9.0.5: icmp_seq=11 ttl=64 time=0.038 ms
64 bytes from 10.9.0.5: icmp_seq=12 ttl=64 time=0.037 ms
64 bytes from 10.9.0.5: icmp_seq=13 ttl=64 time=0.039 ms
64 bytes from 10.9.0.5: icmp_seq=14 ttl=64 time=0.037 ms
64 bytes from 10.9.0.5: icmp_seq=15 ttl=64 time=0.038 ms
64 bytes from 10.9.0.5: icmp_seq=16 ttl=64 time=0.037 ms
64 bytes from 10.9.0.5: icmp_seq=17 ttl=64 time=0.041 ms
64 bytes from 10.9.0.5: icmp_seq=18 ttl=64 time=0.038 ms
64 bytes from 10.9.0.5: icmp_seq=19 ttl=64 time=0.042 ms
64 bytes from 10.9.0.5: icmp_seq=20 ttl=64 time=0.039 ms
64 bytes from 10.9.0.5: icmp_seq=21 ttl=64 time=0.036 ms
64 bytes from 10.9.0.5: icmp_seq=22 ttl=64 time=0.037 ms
64 bytes from 10.9.0.5: icmp_seq=23 ttl=64 time=0.039 ms
64 bytes from 10.9.0.5: icmp_seq=24 ttl=64 time=0.040 ms
64 bytes from 10.9.0.5: icmp_seq=25 ttl=64 time=0.038 ms
64 bytes from 10.9.0.5: icmp_seq=26 ttl=64 time=0.038 ms
64 bytes from 10.9.0.5: icmp_seq=27 ttl=64 time=0.041 ms
64 bytes from 10.9.0.5: icmp_seq=28 ttl=64 time=0.038 ms
64 bytes from 10.9.0.5: icmp_seq=29 ttl=64 time=0.039 ms
64 bytes from 10.9.0.5: icmp_seq=30 ttl=64 time=0.045 ms
64 bytes from 10.9.0.5: icmp_seq=31 ttl=64 time=0.041 ms
64 bytes from 10.9.0.5: icmp_seq=32 ttl=64 time=0.039 ms
64 bytes from 10.9.0.5: icmp_seq=33 ttl=64 time=0.039 ms
64 bytes from 10.9.0.5: icmp_seq=34 ttl=64 time=0.039 ms
```



```
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# ./a.out
Got a packet
From: 10.0.2.4
To: 8.8.8.8
Protocol: ICMP
Got a packet
From: 8.8.8.8
To: 10.0.2.4
Protocol: ICMP
Got a packet
From: 10.0.2.4
To: 8.8.8.8
Protocol: ICMP
Got a packet
From: 8.8.8.8
To: 10.0.2.4
Protocol: ICMP
Got a packet
From: 10.0.2.4
To: 8.8.8.8
Protocol: ICMP
Got a packet
From: 8.8.8.8
To: 10.0.2.4
Protocol: ICMP
Got a packet
From: 10.0.2.4
To: 8.8.8.8
Protocol: ICMP
Got a packet
From: 8.8.8.8
To: 10.0.2.4
Protocol: ICMP
Got a packet
From: 10.0.2.4
To: 128.194.254.1
Protocol: UDP
Got a packet
```

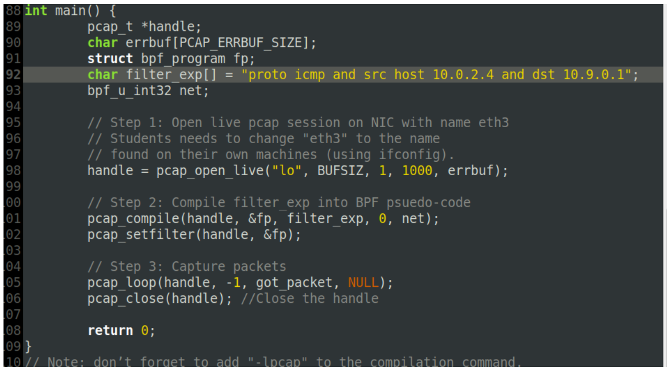
In this task I was required to design a new sniffing program in C, rather than relying on scapy through Python, instead using the pcap library. To this extent, I referenced code from the Hands-On book to develop the sniffing program to sniff out packages from pinging 8.8.8.8 from my host machine, which the results of such being presented in the third picture above.

Question 1: In order to design a sniffing program, first the program must create a socket with the configurations expected for the packets to be intercepted. After this, the program must then construct the information necessary for it to listen to, particularly the source address and port, and bind this information to the socket. Once this is done, the sniffer can begin waiting to receive packets that match these specifications across the network. PCAP does this through its library calls that open a session using raw sockets, then updates the socket's information through the compilation and filter functions, then captures packets through the loop function.

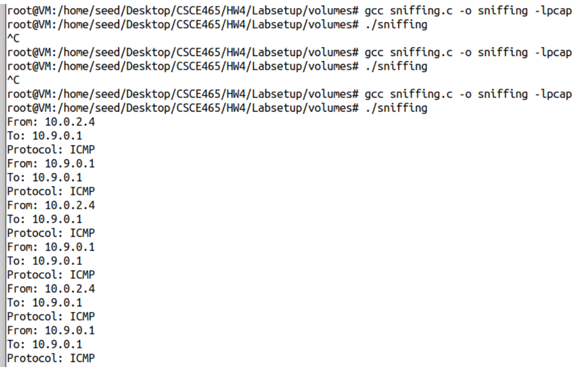
Question 2: Root privilege is necessary to run a sniffer program, as previously explained, due to the use of raw sockets inside of the library calls utilized. To this extent, the program will fail during each pcap library call due to their usage and manipulation of raw sockets within them.

Question 3: When disabling promiscuous mode, I received much less packets through my sniffing program than I had originally so, although the sniffer was able to capture some packets still. While on, the sniffer is able to capture any and all packets passed through the network due to promiscuous mode forcing all packets through the kernel, thus allowing the raw sockets used in the sniffer to detail the traffic accordingly.

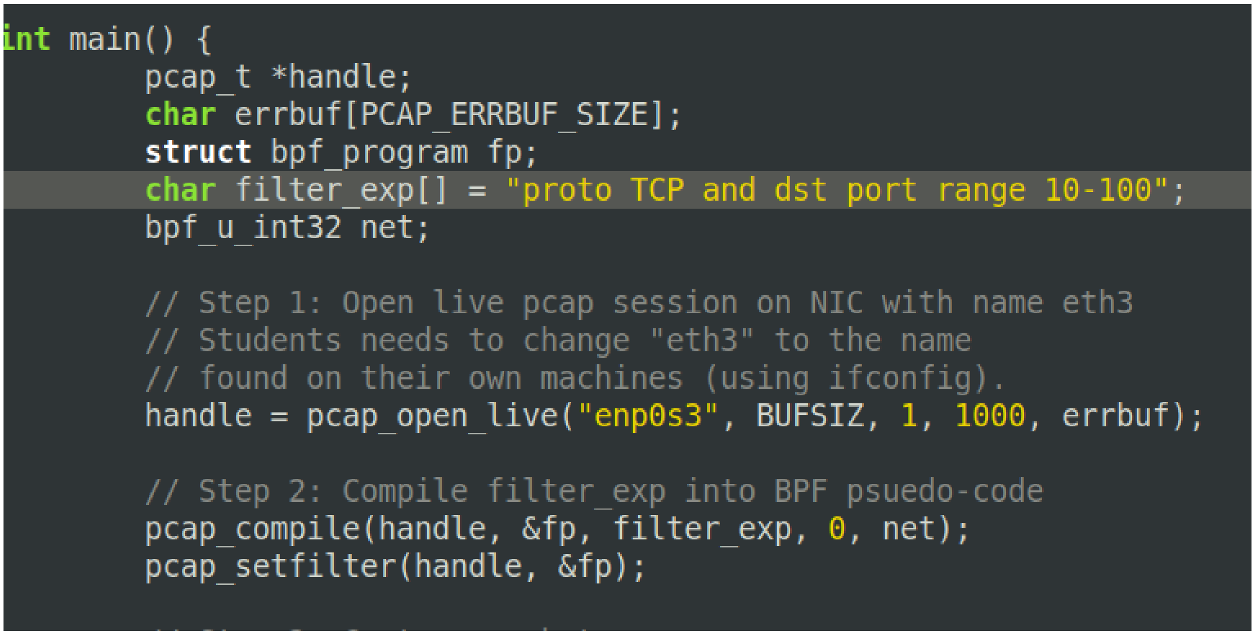
Task 2.1B:



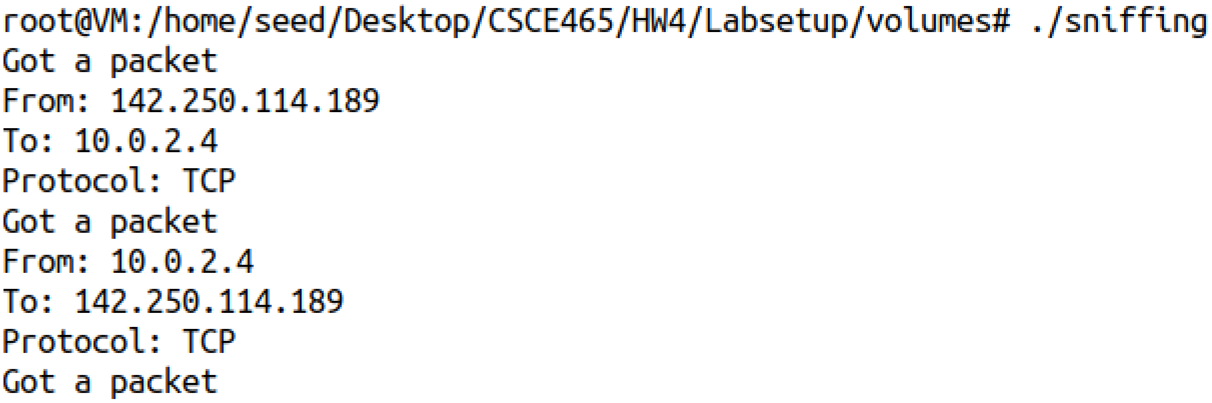
```
int main() {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "proto icmp and src host 10.0.2.4 and dst 10.9.0.1";
        bpf_u_int32 net;

        // Step 1: Open live pcap session on NIC with name eth3
        // Students needs to change "eth3" to the name
        // found on their own machines (using ifconfig).
        handle = pcap_open_live("lo", 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;
}
// Note: don't forget to add "-lpcap" to the compilation command.
```



```
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# gcc sniffing.c -o sniffing -lpcap
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# ./sniffing
^C
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# gcc sniffing.c -o sniffing -lpcap
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# ./sniffing
^C
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# gcc sniffing.c -o sniffing -lpcap
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# ./sniffing
From: 10.0.2.4
To: 10.9.0.1
Protocol: ICMP
From: 10.9.0.1
To: 10.9.0.1
Protocol: ICMP
From: 10.0.2.4
To: 10.9.0.1
Protocol: ICMP
From: 10.9.0.1
To: 10.9.0.1
Protocol: ICMP
From: 10.0.2.4
To: 10.9.0.1
Protocol: ICMP
From: 10.9.0.1
To: 10.9.0.1
Protocol: ICMP
```



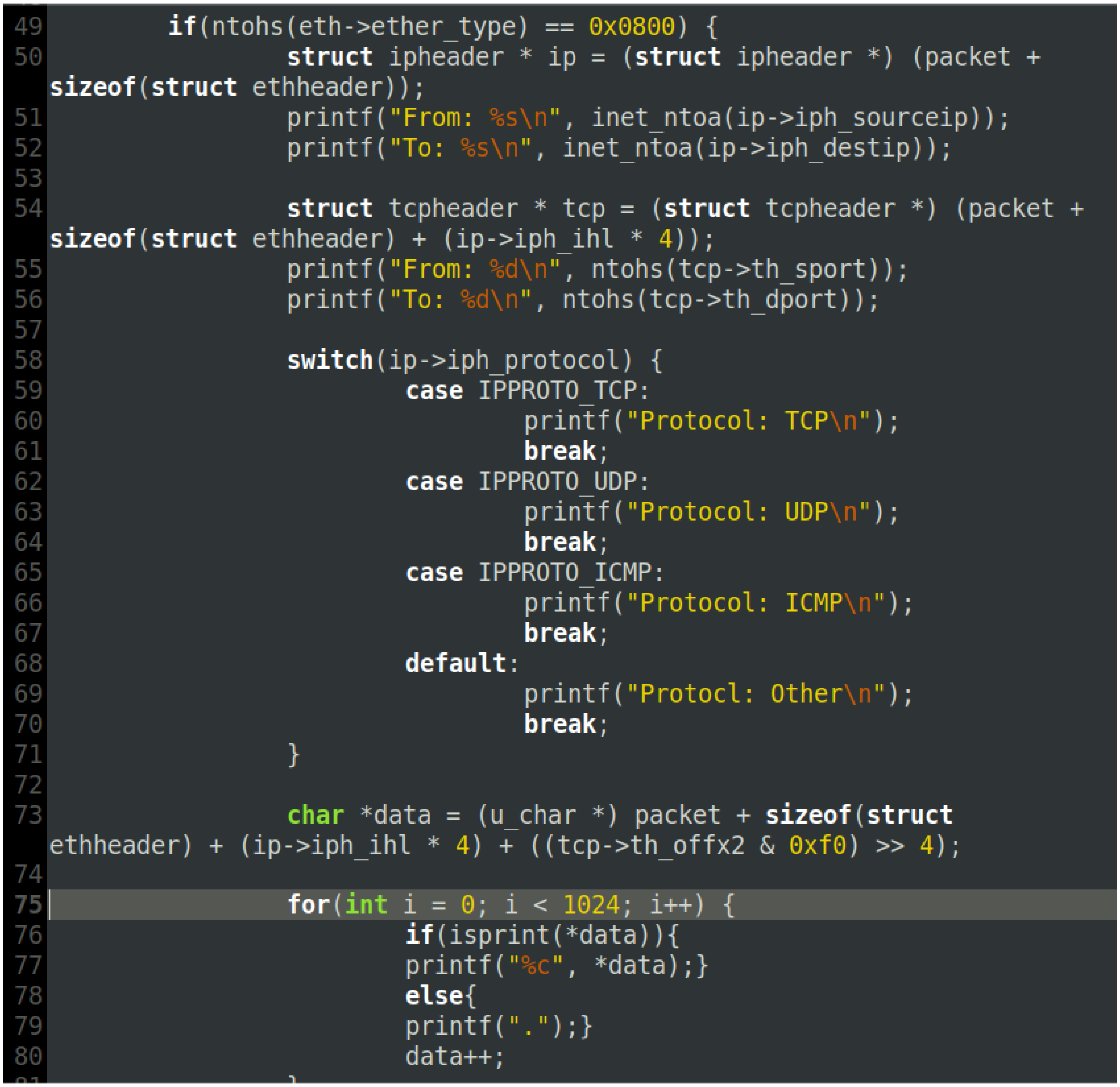
```
int main() {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "proto TCP and dst port range 10-100";
        bpf_u_int32 net;

        // Step 1: Open live pcap session on NIC with name eth3
        // Students needs to change "eth3" to the name
        // found on their own machines (using ifconfig).
        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);
```



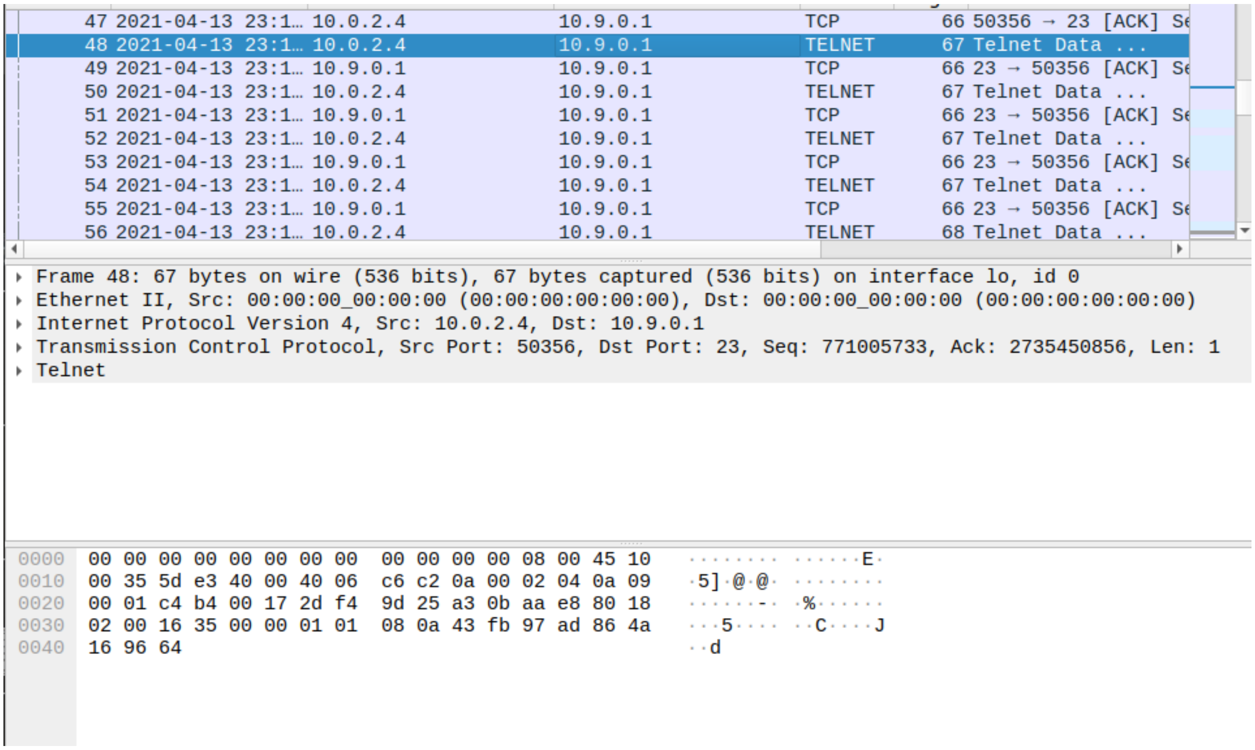
```
root@VM:/home/seed/Desktop/CSCE465/HW4/Labsetup/volumes# ./sniffing
Got a packet
From: 142.250.114.189
To: 10.0.2.4
Protocol: TCP
Got a packet
From: 10.0.2.4
To: 142.250.114.189
Protocol: TCP
Got a packet
```

In this task, I had two separate filters I needed to apply and test with the sniffing program I designed in Task 2.1A. For the first, I applied the filter “proto icmp and src host 10.0.2.4 and dst 10.9.0.1” to monitor the ICMP traffic between the source IP 10.0.2.4 and the destination IP 10.9.0.1, with the results of such found and the edited code found in the first image above. Likewise, for the TCP filter with destination ports from 10 to 100, I applied the filter “proto TCP and dst port range 10-100”, then applied a telnet connection to 10.9.0.1 in order to transmit TCP packets, which were then picked up (results in the third image above and the edited code for TCP in the second).

Task 2.1C:



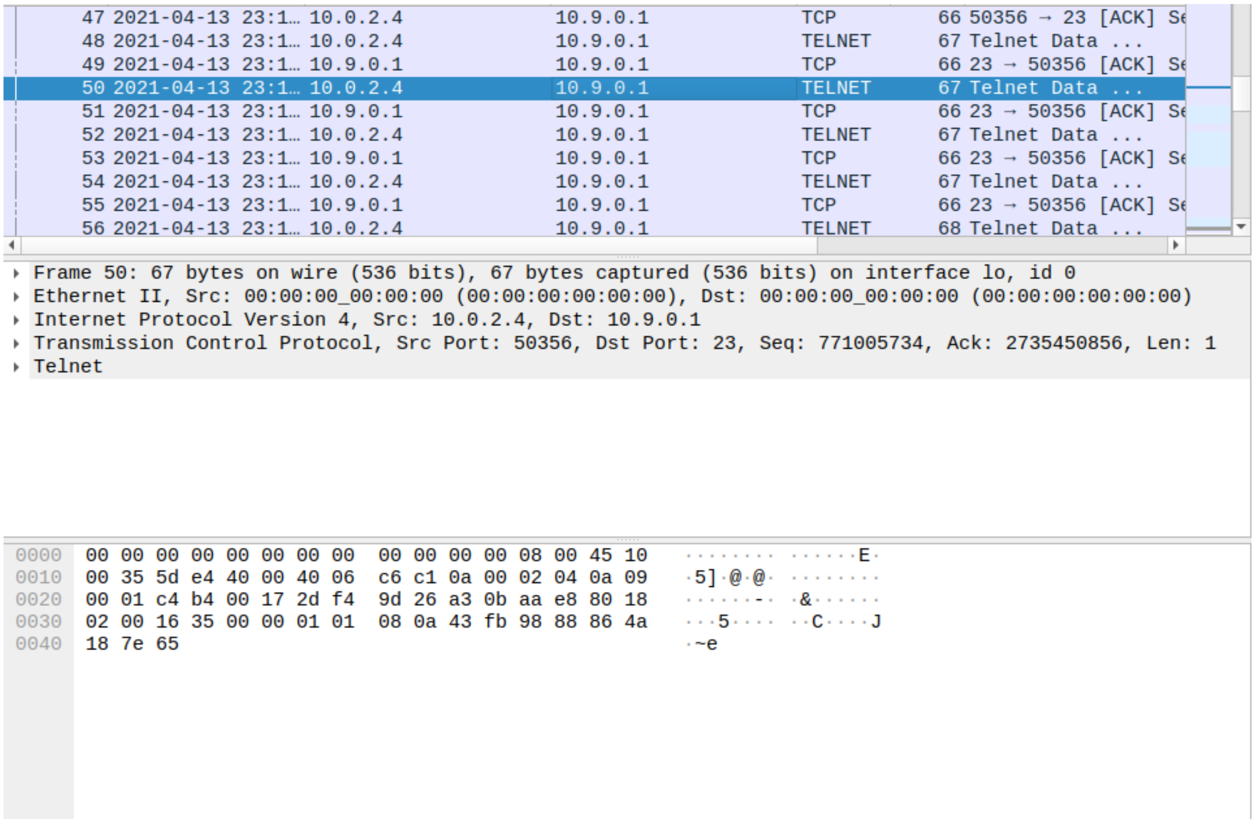
```
        if(ntohs(eth->ether_type) == 0x0800) {
                struct ipheader * ip = (struct ipheader *) (packet + sizeof(struct ethheader));
                printf("From: %s\n", inet_ntoa(ip->iph_sourceip));
                printf("To: %s\n", inet_ntoa(ip->iph_destip));

                struct tcpheader * tcp = (struct tcpheader *) (packet + sizeof(struct ethheader) + (ip->iph_ihl * 4));
                printf("From: %d\n", ntohs(tcp->th_sport));
                printf("To: %d\n", ntohs(tcp->th_dport));

                switch(ip->iph_protocol) {
                        case IPPROTO_TCP:
                                printf("Protocol: TCP\n");
                                break;
                        case IPPROTO_UDP:
                                printf("Protocol: UDP\n");
                                break;
                        case IPPROTO_ICMP:
                                printf("Protocol: ICMP\n");
                                break;
                        default:
                                printf("Protocl: Other\n");
                                break;
                }

                char *data = (u_char *) packet + sizeof(struct ethheader) + (ip->iph_ihl * 4) + ((tcp->th_offx2 & 0xf0) >> 4);

                for(int i = 0; i < 1024; i++) {
                        if(isprint(*data)){
                        printf("%c", *data);}
                        else{
                        printf(".");}
                        data++;
                }
```



```
47 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TCP      66 50356 → 23 [ACK] Se
48 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
49 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
50 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
51 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
52 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
53 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
54 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
55 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
56 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   68 Telnet Data ...
```

```
▸ Frame 48: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.9.0.1
▸ Transmission Control Protocol, Src Port: 50356, Dst Port: 23, Seq: 771005733, Ack: 2735450856, Len: 1
▸ Telnet
```

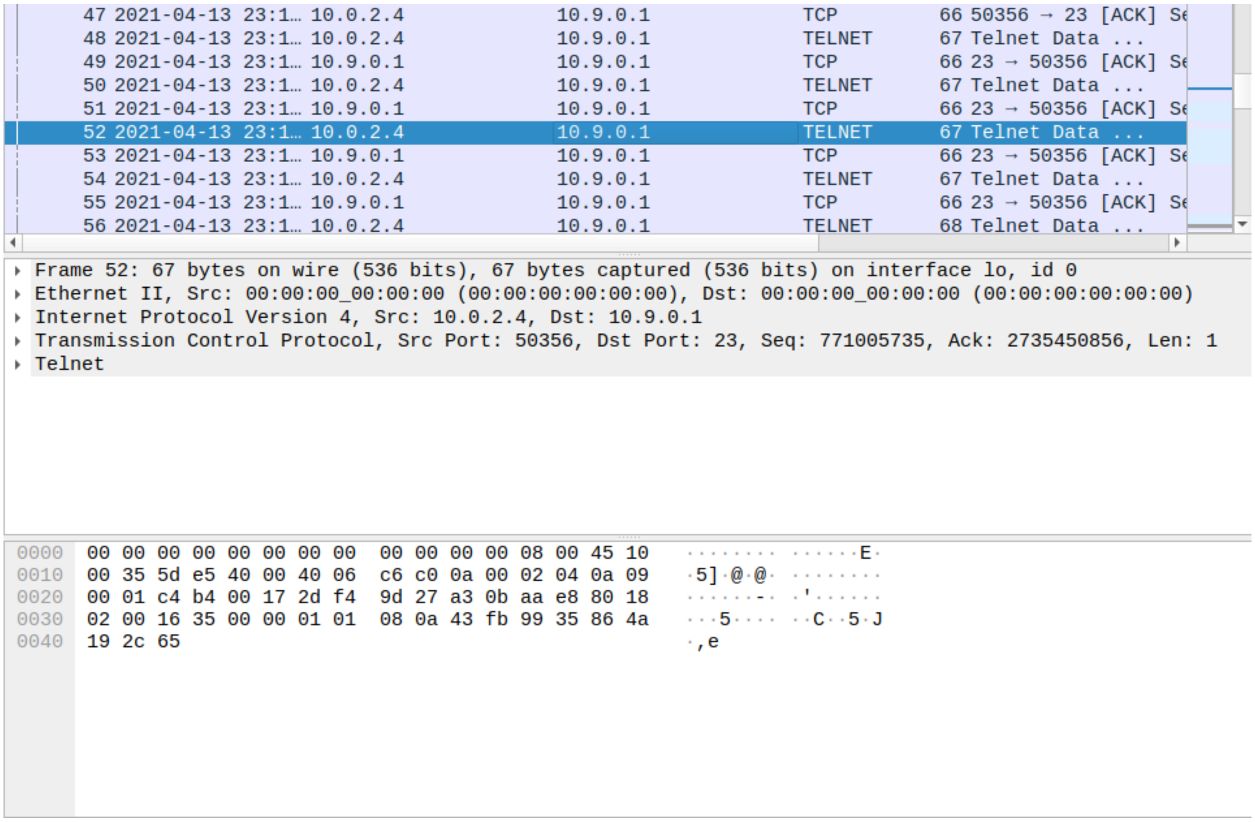
```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 10   ········ ······E·
0010  00 35 5d e3 40 00 40 06  c6 c2 0a 00 02 04 0a 09   ·5]·@·@· ········
0020  00 01 c4 b4 00 17 2d f4  9d 25 a3 0b aa e8 80 18   ······-· ·%······
0030  02 00 16 35 00 00 01 01  08 0a 43 fb 97 ad 86 4a   ···5···· ··C····J
0040  16 96 64                                           ··d
```



```
47 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TCP      66 50356 → 23 [ACK] Se
48 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
49 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
50 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
51 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
52 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
53 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
54 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
55 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
56 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   68 Telnet Data ...
```

```
▸ Frame 50: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.9.0.1
▸ Transmission Control Protocol, Src Port: 50356, Dst Port: 23, Seq: 771005734, Ack: 2735450856, Len: 1
▸ Telnet
```

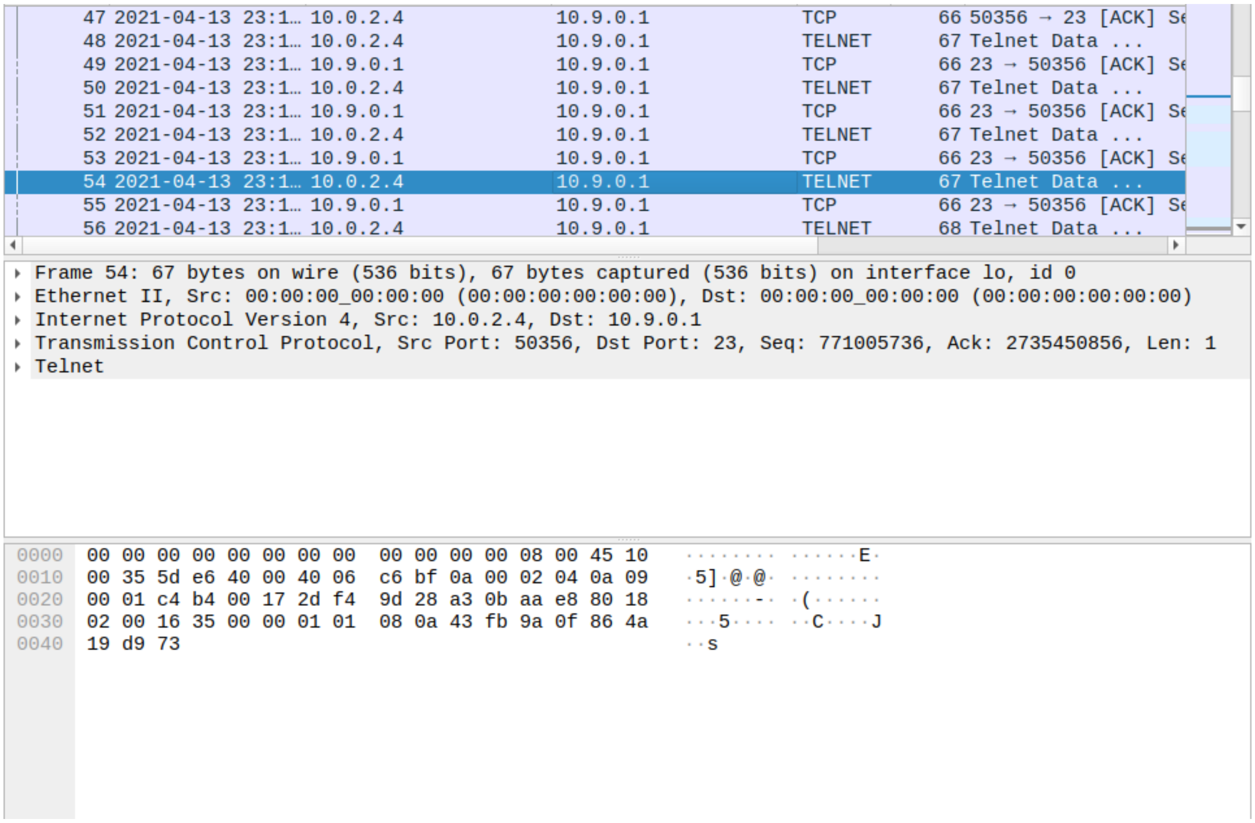
```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 10   ········ ······E·
0010  00 35 5d e4 40 00 40 06  c6 c1 0a 00 02 04 0a 09   ·5]·@·@· ········
0020  00 01 c4 b4 00 17 2d f4  9d 26 a3 0b aa e8 80 18   ······-· ·&······
0030  02 00 16 35 00 00 01 01  08 0a 43 fb 98 88 86 4a   ···5···· ··C····J
0040  18 7e 65                                           ·~e
```



```
47 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TCP      66 50356 → 23 [ACK] Se
48 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
49 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
50 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
51 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
52 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
53 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
54 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
55 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
56 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   68 Telnet Data ...
```

```
▸ Frame 52: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.9.0.1
▸ Transmission Control Protocol, Src Port: 50356, Dst Port: 23, Seq: 771005735, Ack: 2735450856, Len: 1
▸ Telnet
```

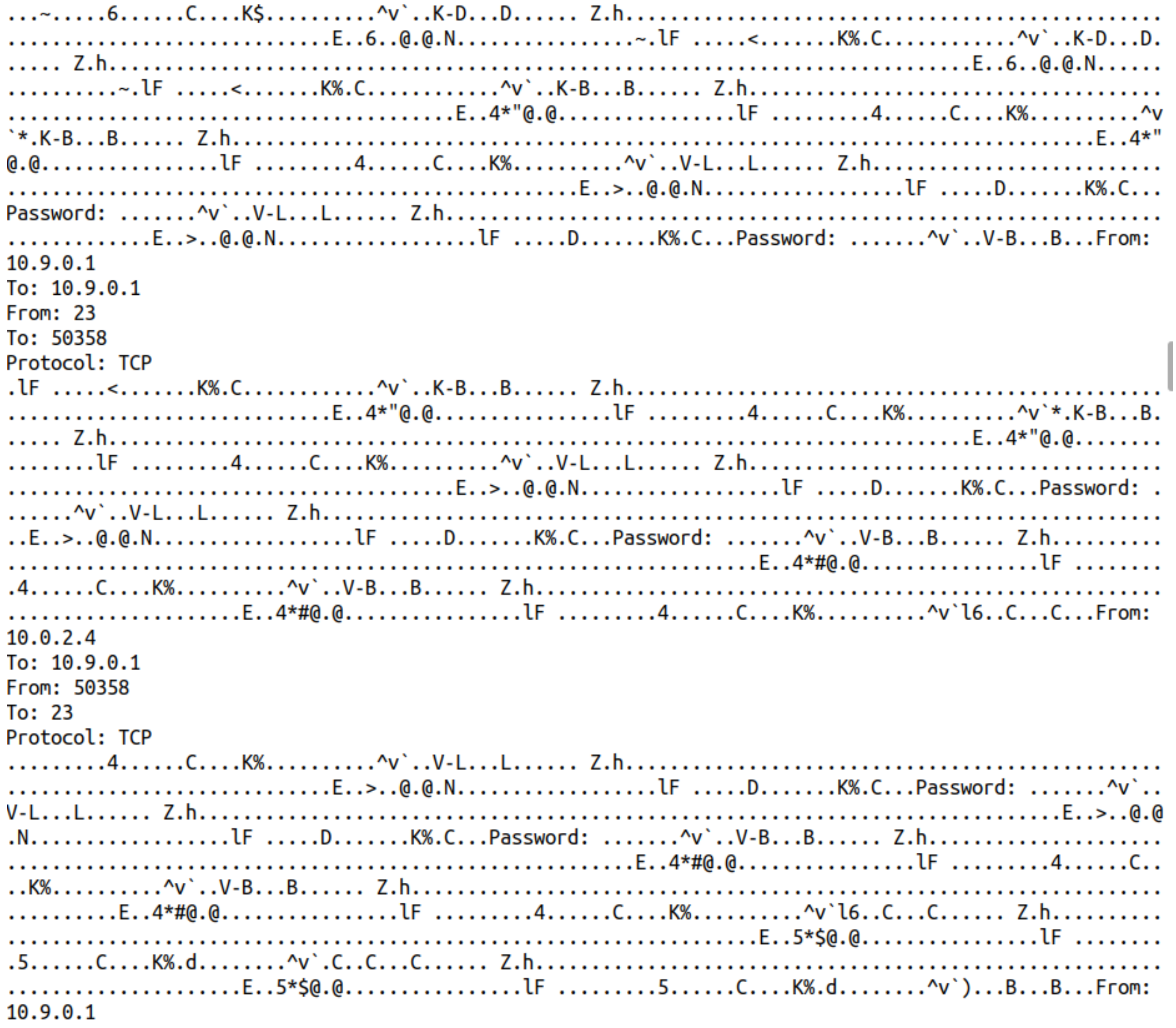
```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 10   ········ ······E·
0010  00 35 5d e5 40 00 40 06  c6 c0 0a 00 02 04 0a 09   ·5]·@·@· ········
0020  00 01 c4 b4 00 17 2d f4  9d 27 a3 0b aa e8 80 18   ······-· ·'······
0030  02 00 16 35 00 00 01 01  08 0a 43 fb 99 35 86 4a   ···5···· ··C··5·J
0040  19 2c 65                                           ·,e
```



```
47 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TCP      66 50356 → 23 [ACK] Se
48 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
49 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
50 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
51 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
52 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
53 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
54 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   67 Telnet Data ...
55 2021-04-13 23:1… 10.9.0.1    10.9.0.1    TCP      66 23 → 50356 [ACK] Se
56 2021-04-13 23:1… 10.0.2.4    10.9.0.1    TELNET   68 Telnet Data ...
```

```
▸ Frame 54: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.9.0.1
▸ Transmission Control Protocol, Src Port: 50356, Dst Port: 23, Seq: 771005736, Ack: 2735450856, Len: 1
▸ Telnet
```

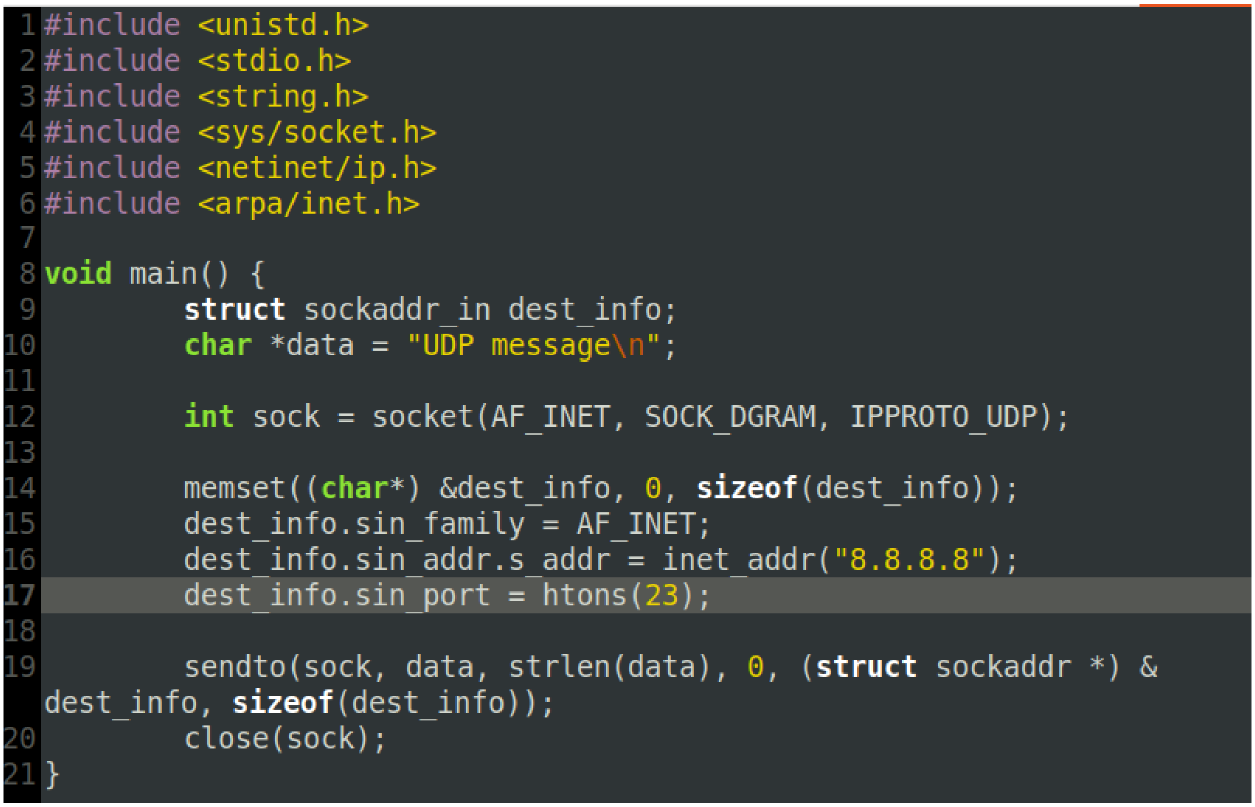
```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 10   ········ ······E·
0010  00 35 5d e6 40 00 40 06  c6 bf 0a 00 02 04 0a 09   ·5]·@·@· ········
0020  00 01 c4 b4 00 17 2d f4  9d 28 a3 0b aa e8 80 18   ······-· ·(······
0030  02 00 16 35 00 00 01 01  08 0a 43 fb 9a 0f 86 4a   ···5···· ··C····J
0040  19 d9 73                                           ··s
```



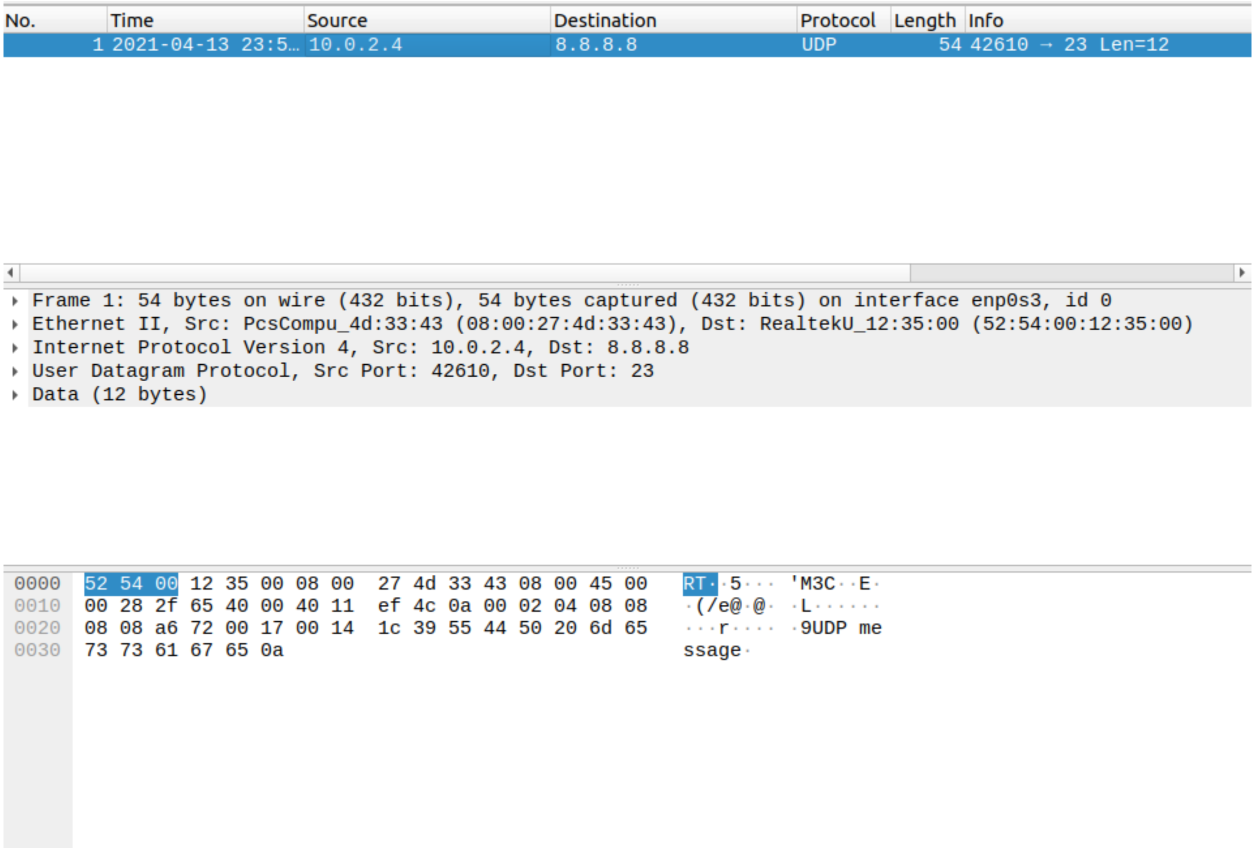
```
...~.....6......C....K$.........^v`..K-D...D...... Z.h...............................................
...........................E..6..@.@.N...............~.lF .....<.......K%.C...........^v`..K-D...D.
..... Z.h.......................................................................E..6..@.@.N......
..........~.lF .....<.......K%.C...........^v`..K-B...B...... Z.h..................................
.....................................E..4*"@.@...............lF .........4......C....K%..........^v
`*.K-B...B...... Z.h.......................................................................E..4*"
@.@...............lF .........4......C....K%..........^v`..V-L...L...... Z.h........................
...............................................E..>..@.@.N.................lF .....D.......K%.C...
Password: .......^v`..V-L...L...... Z.h.............................................................
............E..>..@.@.N.................lF .....D.......K%.C...Password: .......^v`..V-B...B...From:
10.9.0.1
To: 10.9.0.1
From: 23
To: 50358
Protocol: TCP
.lF .....<.......K%.C...........^v`..K-B...B...... Z.h..............................................
...........................E..4*"@.@...............lF .........4......C....K%..........^v`*.K-B...B.
..... Z.h.......................................................................E..4*"@.@.......
........lF .........4......C....K%..........^v`..V-L...L...... Z.h..................................
.....................................E..>..@.@.N.................lF .....D.......K%.C...Password: .
......^v`..V-L...L...... Z.h........................................................................
..E..>..@.@.N.................lF .....D.......K%.C...Password: .......^v`..V-B...B...... Z.h.........
...............................................................E..4*#@.@...............lF ........
.4......C....K%..........^v`..V-B...B...... Z.h.....................................................
....................E..4*#@.@...............lF .........4......C....K%..........^v`l6..C...C...From:
10.0.2.4
To: 10.9.0.1
From: 50358
To: 23
Protocol: TCP
.........4......C....K%..........^v`..V-L...L...... Z.h..............................................
...........................E..>..@.@.N.................lF .....D.......K%.C...Password: .......^v`..
V-L...L...... Z.h.......................................................................E..>..@.@
.N.................lF .....D.......K%.C...Password: .......^v`..V-B...B...... Z.h....................
....................................................E..4*#@.@...............lF .........4......C..
..K%..........^v`..V-B...B...... Z.h.................................................................
.........E..4*#@.@...............lF .........4......C....K%..........^v`l6..C...C...... Z.h.........
...............................................................E..5*$@.@...............lF ........
.5......C....K%.d........^v`.C..C...C...... Z.h.....................................................
....................E..5*$@.@...............lF .........5......C....K%.d........^v`)...B...B...From:
10.9.0.1
```

In this task I was required to monitor network traffic (i.e. TCP packets transmitted through a telnet) to sniff a password from the transaction between the two hosts. In the first image above I document the code I modified in order to output the contents of the packets, and the last image shows the printed out contents of the packets received. Unfortunately, I was unable to read out the password from these packet contents output from my sniffer, but was able to find the password in the last line of each packet received in the previous four images (starting after my code images). The interesting thing about these packets, and the password “dees” spelled out at the end of each of them, is that the password is contained within “telnet” packets, rather than TCP packets (which are processed alongside them). The contents of these TCP packets transmitted between them don’t have the password within them (even when monitored within WireShark).

Task 2.2A:



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

void main() {
        struct sockaddr_in dest_info;
        char *data = "UDP message\n";

        int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

        memset((char*) &dest_info, 0, sizeof(dest_info));
        dest_info.sin_family = AF_INET;
        dest_info.sin_addr.s_addr = inet_addr("8.8.8.8");
        dest_info.sin_port = htons(23);

        sendto(sock, data, strlen(data), 0, (struct sockaddr *) &
dest_info, sizeof(dest_info));
        close(sock);
}
```



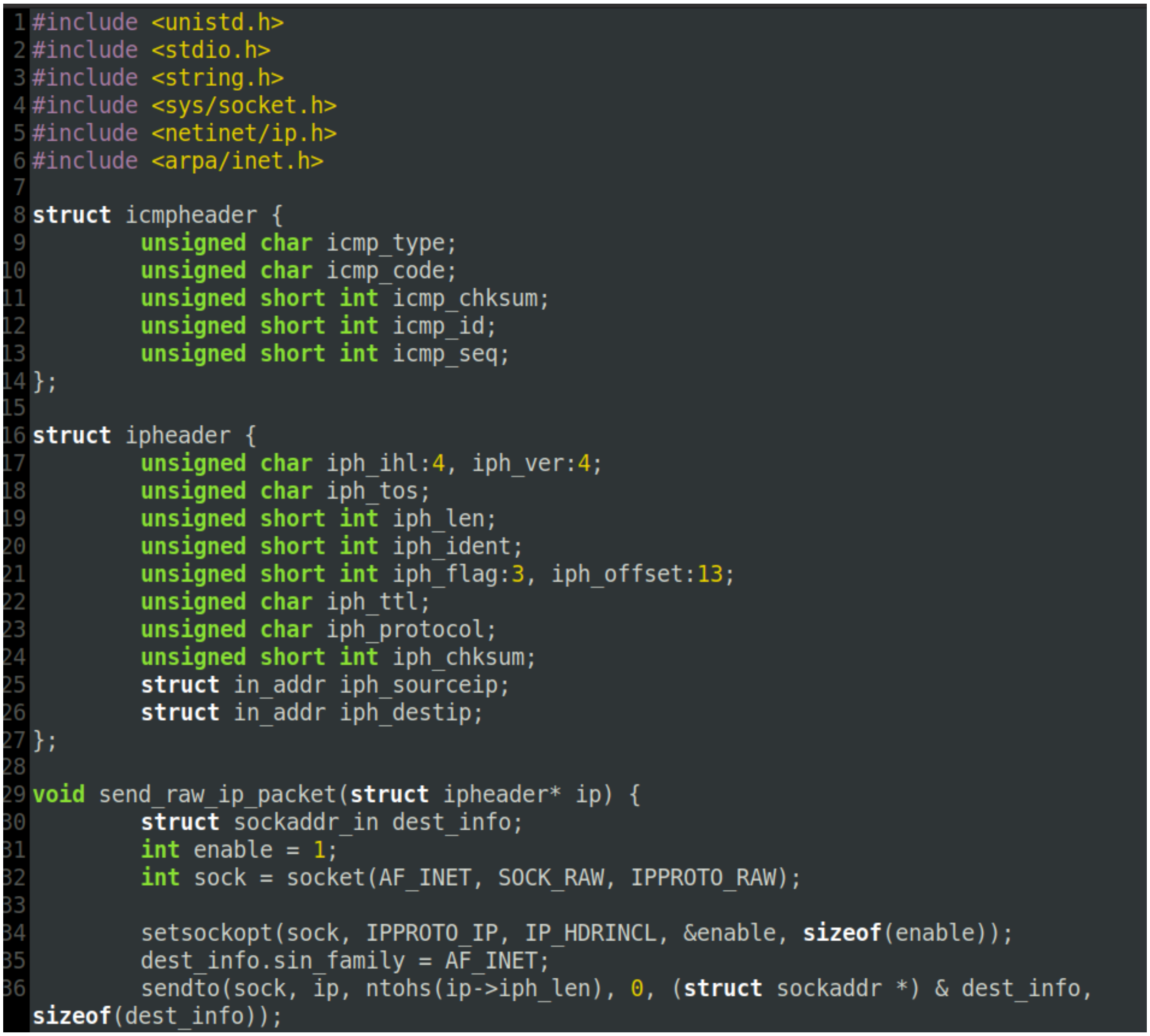
No.	Time	Source	Destination	Protocol	Length	Info
1	2021-04-13 23:5…	10.0.2.4	8.8.8.8	UDP	54	42610 → 23 Len=12

```
▸ Frame 1: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PcsCompu_4d:33:43 (08:00:27:4d:33:43), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
▸ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 8.8.8.8
▸ User Datagram Protocol, Src Port: 42610, Dst Port: 23
▸ Data (12 bytes)
```

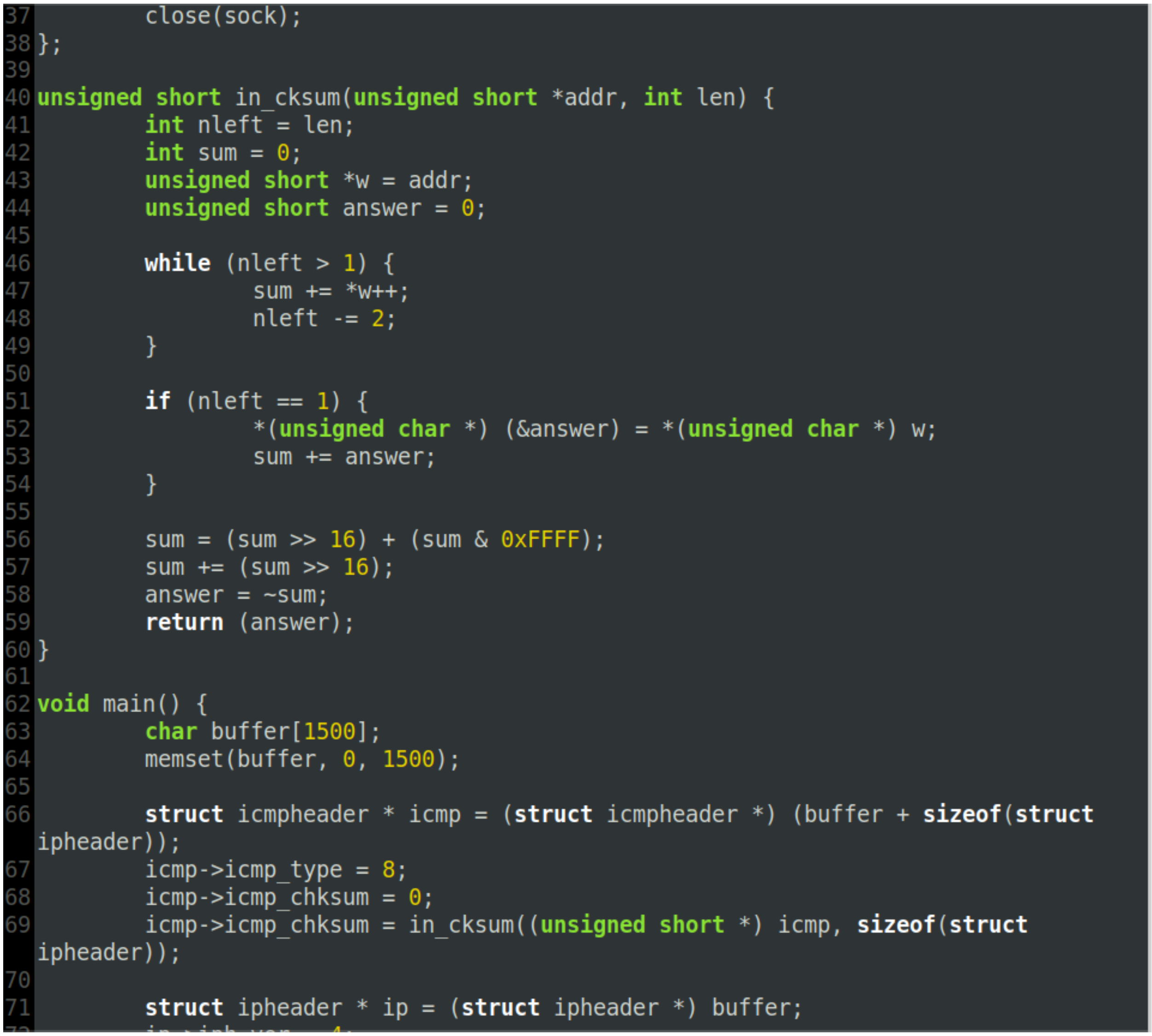
```
0000  52 54 00 12 35 00 08 00  27 4d 33 43 08 00 45 00   RT··5···  'M3C··E·
0010  00 28 2f 65 40 00 40 11  ef 4c 0a 00 02 04 08 08   ·(/e@·@·  ·L······
0020  08 08 a6 72 00 17 00 14  1c 39 55 44 50 20 6d 65   ···r····  ·9UDP me
0030  73 73 61 67 65 0a                                  ssage·
```

In this task I simply designed a program that would use C and raw sockets in order to send spoofed packets to an arbitrary destination host. This designed code is located in the first image above, with evidence of the spoofed packet located in the second image.

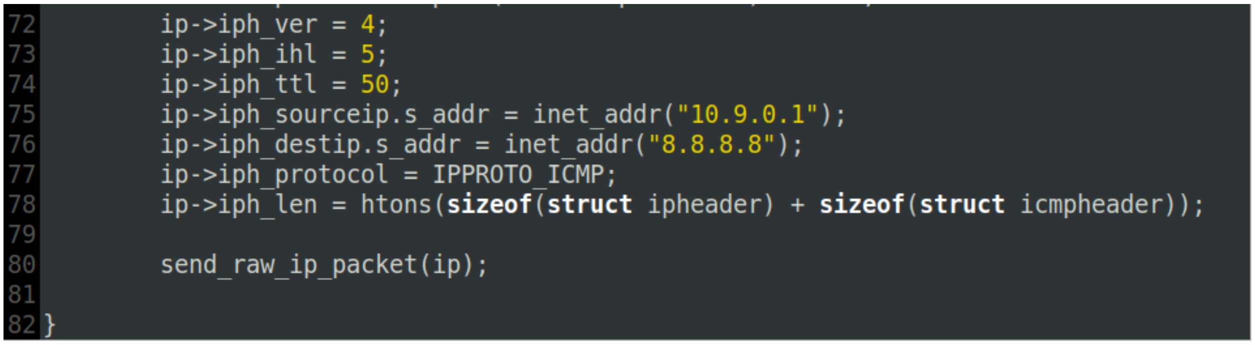
Task 2.2B:



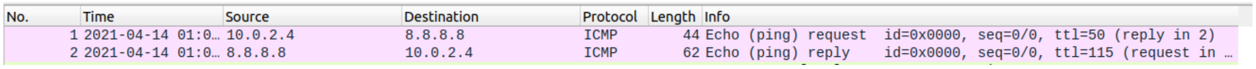
```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

struct icmpheader {
        unsigned char icmp_type;
        unsigned char icmp_code;
        unsigned short int icmp_chksum;
        unsigned short int icmp_id;
        unsigned short int icmp_seq;
};

struct ipheader {
        unsigned char iph_ihl:4, iph_ver:4;
        unsigned char iph_tos;
        unsigned short int iph_len;
        unsigned short int iph_ident;
        unsigned short int iph_flag:3, iph_offset:13;
        unsigned char iph_ttl;
        unsigned char iph_protocol;
        unsigned short int iph_chksum;
        struct in_addr iph_sourceip;
        struct in_addr iph_destip;
};

void send_raw_ip_packet(struct ipheader* ip) {
        struct sockaddr_in dest_info;
        int enable = 1;
        int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

        setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
        dest_info.sin_family = AF_INET;
        sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *) & dest_info,
sizeof(dest_info));
```



```c
        close(sock);
};

unsigned short in_cksum(unsigned short *addr, int len) {
        int nleft = len;
        int sum = 0;
        unsigned short *w = addr;
        unsigned short answer = 0;

        while (nleft > 1) {
                sum += *w++;
                nleft -= 2;
        }

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

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

void 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
ipheader));

        struct ipheader * ip = (struct ipheader *) buffer;
```



```c
        ip->iph_ver = 4;
        ip->iph_ihl = 5;
        ip->iph_ttl = 50;
        ip->iph_sourceip.s_addr = inet_addr("10.9.0.1");
        ip->iph_destip.s_addr = inet_addr("8.8.8.8");
        ip->iph_protocol = IPPROTO_ICMP;
        ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));

        send_raw_ip_packet(ip);

}
```



No.	Time	Source	Destination	Protocol	Length	Info
1	2021-04-14 01:0…	10.0.2.4	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=50 (reply in 2)
2	2021-04-14 01:0…	8.8.8.8	10.0.2.4	ICMP	62	Echo (ping) reply id=0x0000, seq=0/0, ttl=115 (request in …

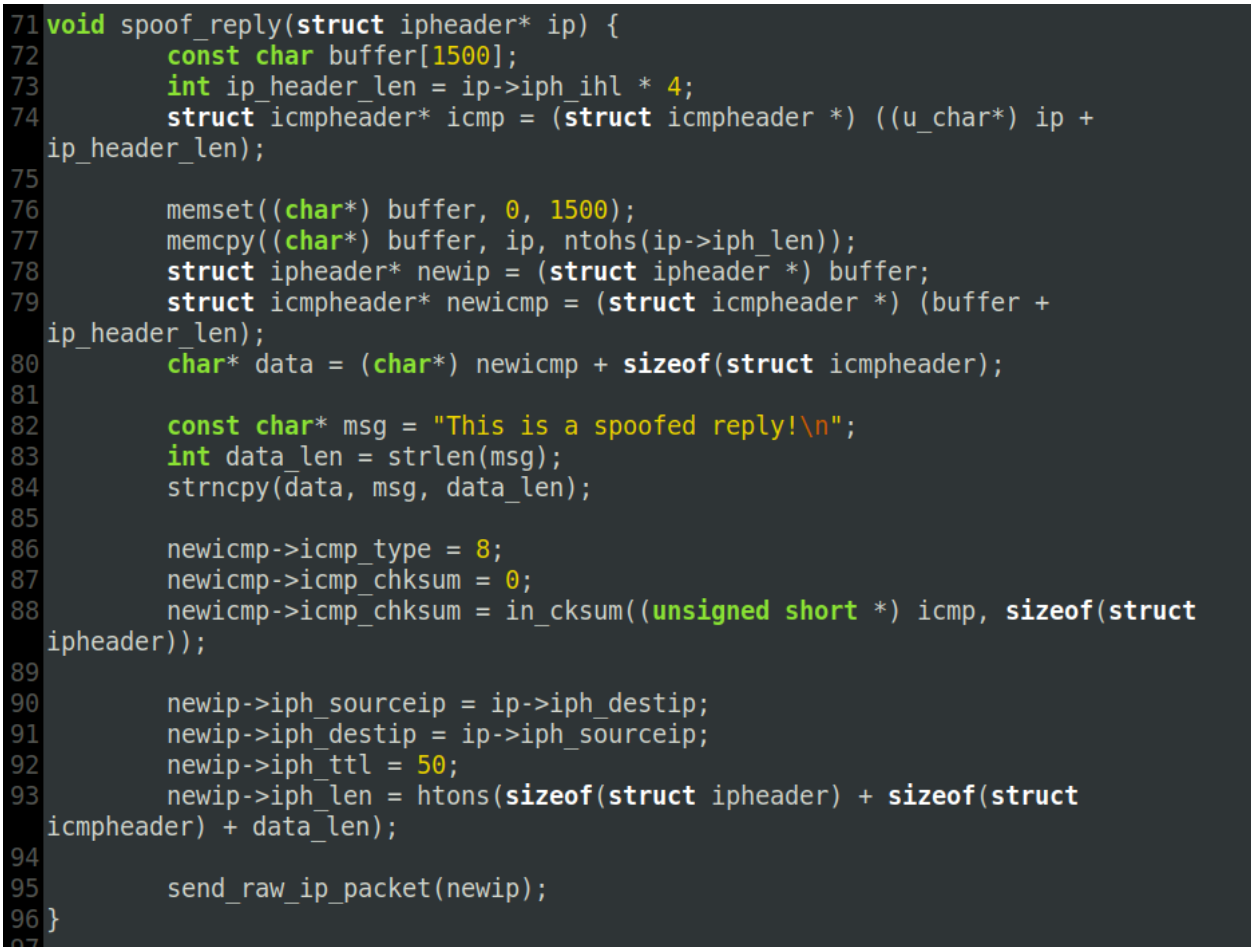
Similarly, in this task, I modified the code (as per requirements) to spoof echo request packets. This process required me to adopt the Hands-On book's checksum function and update the ICMP and IP details accordingly to then use the send_raw_ip_packet function from the previous code to send the spoofed packet echo on behalf of 8.8.8.8. The modified code is found in the first three images, with the evidence of the spoofed echo being located in the fourth image, containing the spoofed contents within such to verify it was the spoofed result.

Question 4: The length of the IP header can be set arbitrarily due to the fact that the IP length is modified to its original size via input without manually being set.

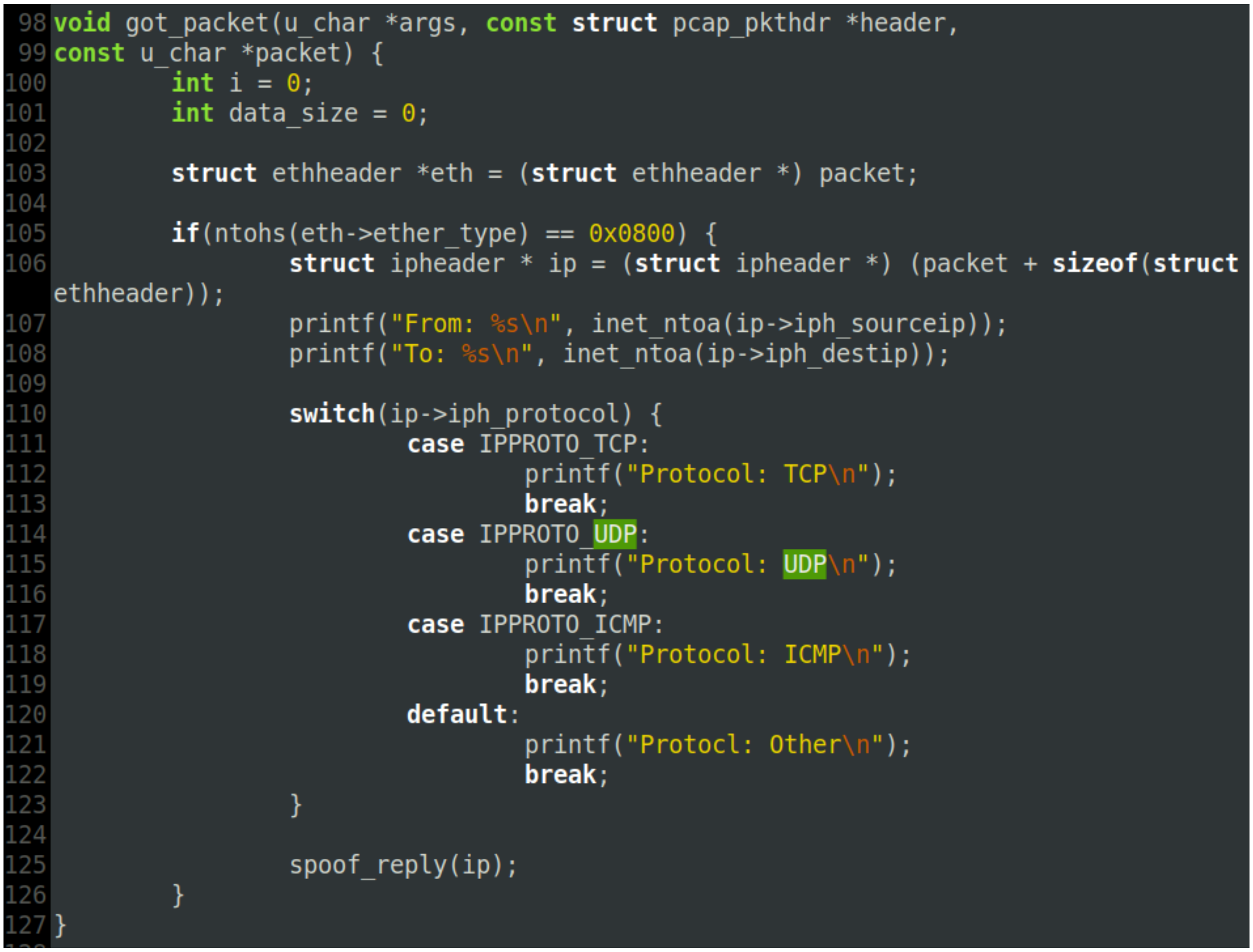
Question 5: Due to the nature of raw sockets the checksum has to be calculated manually for an arbitrary packet being sent by a programmer.

Question 6: Again, as mentioned previously, root privileges are necessary to use raw sockets due to the fact that they allow a user to simulate a server on any port, which would break many customary rules in place for networking. As such, trying to execute this program as a non-root user the program would crash due to permissions issues rising when creating/manipulating raw sockets.

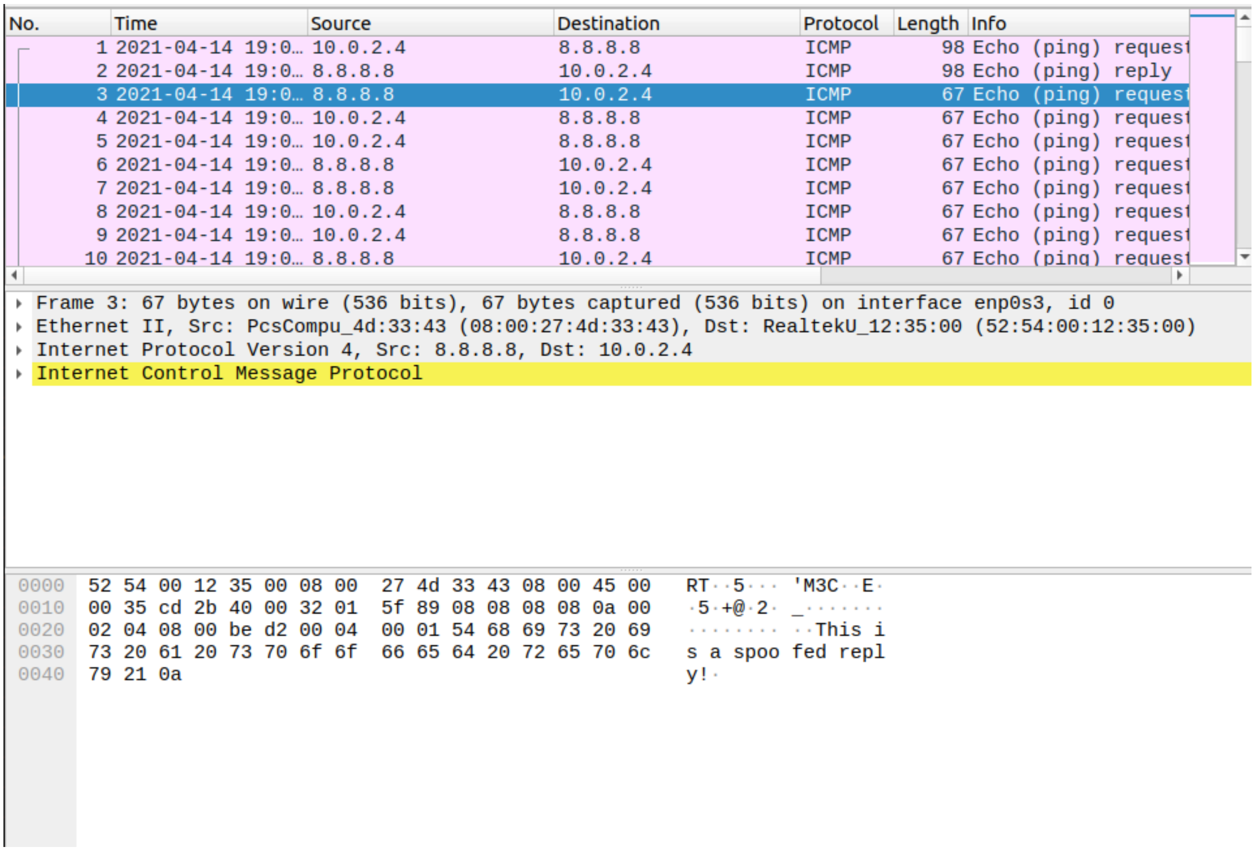
Task 2.3:



```c
void spoof_reply(struct ipheader* ip) {
        const char buffer[1500];
        int ip_header_len = ip->iph_ihl * 4;
        struct icmpheader* icmp = (struct icmpheader *) ((u_char*) ip + ip_header_len);

        memset((char*) buffer, 0, 1500);
        memcpy((char*) buffer, ip, ntohs(ip->iph_len));
        struct ipheader* newip = (struct ipheader *) buffer;
        struct icmpheader* newicmp = (struct icmpheader *) (buffer + ip_header_len);
        char* data = (char*) newicmp + sizeof(struct icmpheader);

        const char* msg = "This is a spoofed reply!\n";
        int data_len = strlen(msg);
        strncpy(data, msg, data_len);

        newicmp->icmp_type = 8;
        newicmp->icmp_chksum = 0;
        newicmp->icmp_chksum = in_cksum((unsigned short *) icmp, sizeof(struct ipheader));

        newip->iph_sourceip = ip->iph_destip;
        newip->iph_destip = ip->iph_sourceip;
        newip->iph_ttl = 50;
        newip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader) + data_len);

        send_raw_ip_packet(newip);
}
```



```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet) {
        int i = 0;
        int data_size = 0;

        struct ethheader *eth = (struct ethheader *) packet;

        if(ntohs(eth->ether_type) == 0x0800) {
                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));

                switch(ip->iph_protocol) {
                        case IPPROTO_TCP:
                                printf("Protocol: TCP\n");
                                break;
                        case IPPROTO_UDP:
                                printf("Protocol: UDP\n");
                                break;
                        case IPPROTO_ICMP:
                                printf("Protocol: ICMP\n");
                                break;
                        default:
                                printf("Protocl: Other\n");
                                break;
                }

                spoof_reply(ip);
        }
}
```



No.	Time	Source	Destination	Protocol	Length	Info
1	2021-04-14 19:0…	10.0.2.4	8.8.8.8	ICMP	98	Echo (ping) request
2	2021-04-14 19:0…	8.8.8.8	10.0.2.4	ICMP	98	Echo (ping) reply
3	2021-04-14 19:0…	8.8.8.8	10.0.2.4	ICMP	67	Echo (ping) request
4	2021-04-14 19:0…	10.0.2.4	8.8.8.8	ICMP	67	Echo (ping) request
5	2021-04-14 19:0…	10.0.2.4	8.8.8.8	ICMP	67	Echo (ping) request
6	2021-04-14 19:0…	8.8.8.8	10.0.2.4	ICMP	67	Echo (ping) request
7	2021-04-14 19:0…	8.8.8.8	10.0.2.4	ICMP	67	Echo (ping) request
8	2021-04-14 19:0…	10.0.2.4	8.8.8.8	ICMP	67	Echo (ping) request
9	2021-04-14 19:0…	10.0.2.4	8.8.8.8	ICMP	67	Echo (ping) request
10	2021-04-14 19:0…	8.8.8.8	10.0.2.4	ICMP	67	Echo (ping) request

```
▸ Frame 3: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PcsCompu_4d:33:43 (08:00:27:4d:33:43), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
▸ Internet Protocol Version 4, Src: 8.8.8.8, Dst: 10.0.2.4
▸ Internet Control Message Protocol
```

```
0000  52 54 00 12 35 00 08 00  27 4d 33 43 08 00 45 00   RT··5··· 'M3C··E·
0010  00 35 cd 2b 40 00 32 01  5f 89 08 08 08 08 0a 00   ·5·+@·2· _·······
0020  02 04 08 00 be d2 00 04  00 01 54 68 69 73 20 69   ········ ··This i
0030  73 20 61 20 73 70 6f 6f  66 65 64 20 72 65 70 6c   s a spoo fed repl
0040  79 21 0a                                           y!·
```

Finally, in this task I combined the code from the previous subtasks of Task 2 in order to create a program that first sniffs for packets and then sends spoofed packets in reply to such. To do this, I adopted the spoof_reply function from the Hands-On book and used such at the end of my sniffing function (specifically called got_packet) that got IP details from the pcap calls used previously (code detailed in the first two images above). The evidence of the ping request and the spoofed reply is found in the third and final image above, with the spoofed text detailed by WireShark from there.