Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Task 1.1: Sniffing Packets

Task1.1A – Running the packet sniffing program to prove that packets can be captured, as well as testing how the usage of the root privilege changes the obtained packets.

Exhibit 1: Snippet from initial output of sniffer.py, while running with root privileges.

```
[01/16/21]seed@VM:~/A1$ sudo python sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:a6:95:06
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0xc0
     len       = 209
     id        = 36730
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x1c3c
     src       = 10.0.2.13
     dst       = 192.168.1.1
     \options   \
###[ ICMP ]###
        type      = dest-unreach
        code      = port-unreachable
        chksum    = 0xcb65
        reserved  = 0
        length    = 0
        nexthopmtu= 0
```

Exhibit 2: Initial output from sniffer.py, continued.

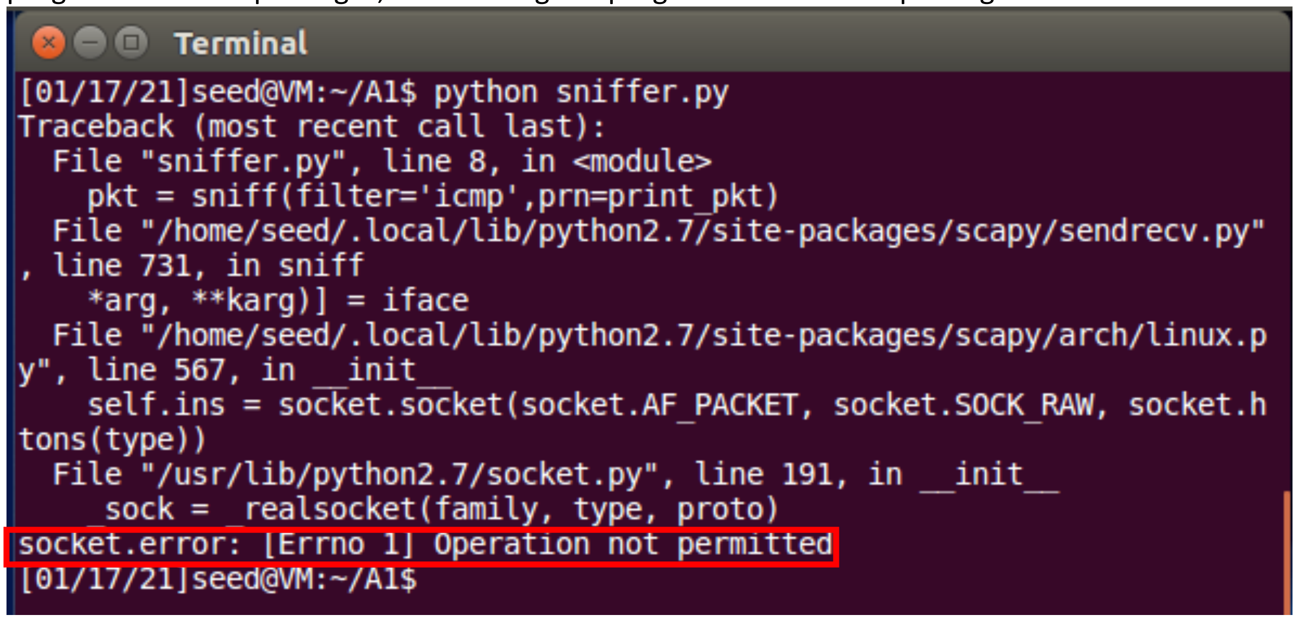
```
###[ IP in ICMP ]###
           version   = 4
           ihl       = 5
           tos       = 0x0
           len       = 181
           id        = 5829
           flags     =
           frag      = 0
           ttl       = 255
           proto     = udp
           chksum    = 0xd6bc
           src       = 192.168.1.1
           dst       = 10.0.2.13
           \options   \
###[ UDP in ICMP ]###
              sport     = domain
              dport     = 48027
              len       = 161
              chksum    = 0xd4e6
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 3: Initial output from sniffer.py, continued.

```
###[ DNS ]###
                 id        = 9894
                 qr        = 1
                 opcode    = QUERY
                 aa        = 0
                 tc        = 0
                 rd        = 1
                 ra        = 1
                 z         = 0
                 ad        = 0
                 cd        = 0
                 rcode     = ok
                 qdcount   = 1
                 ancount   = 3
                 nscount   = 0
                 arcount   = 0
                 \qd        \
                  |###[ DNS Question Record ]###
                  |  qname     = 'detectportal.firefox.com.'
                  |  qtype     = A
                  |  qclass    = IN
                 \an        \
                  |###[ DNS Resource Record ]###
                  |  rrname    = 'detectportal.firefox.com.'
                  |  type      = CNAME
                  |  rclass    = IN
                  |  ttl       = 44
                  |  rdlen     = 30
                  |  rdata     = 'detectportal.prod.mozaws.net.'
                  |###[ DNS Resource Record ]###
                  |  rrname    = 'detectportal.prod.mozaws.net.'
                  |  type      = CNAME
                  |  rclass    = IN
                  |  ttl       = 254
                  |  rdlen     = 44
                  |  rdata     = 'prod.detectportal.prod.cloudops.mozgcp.net.'
                  |###[ DNS Resource Record ]###
                  |  rrname    = 'prod.detectportal.prod.cloudops.mozgcp.net.'
                  |  type      = A
                  |  rclass    = IN
                  |  ttl       = 267
                  |  rdlen     = 4
                  |  rdata     = '34.107.221.82'
                 ns        = None
                 ar        = None
```

Exhibit 4: Attempting to run the script without root privileges, resulting in an error stating that the operation is not permitted. This was the main difference that I noted between running the program with root privileges, and running the program without root privileges.

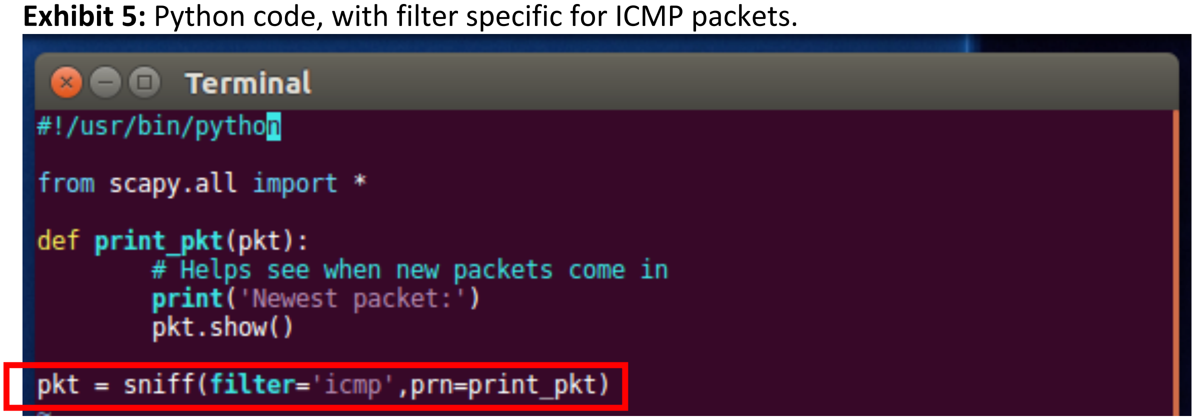


```
[01/17/21]seed@VM:~/A1$ python sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 8, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/sendrecv.py"
, line 731, in sniff
    *arg, **karg)] = iface
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/arch/linux.p
y", line 567, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.h
tons(type))
  File "/usr/lib/python2.7/socket.py", line 191, in __init__
    _sock = _realsocket(family, type, proto)
socket.error: [Errno 1] Operation not permitted
[01/17/21]seed@VM:~/A1$
```

Task 1.1B

- Capturing only the ICMP packet

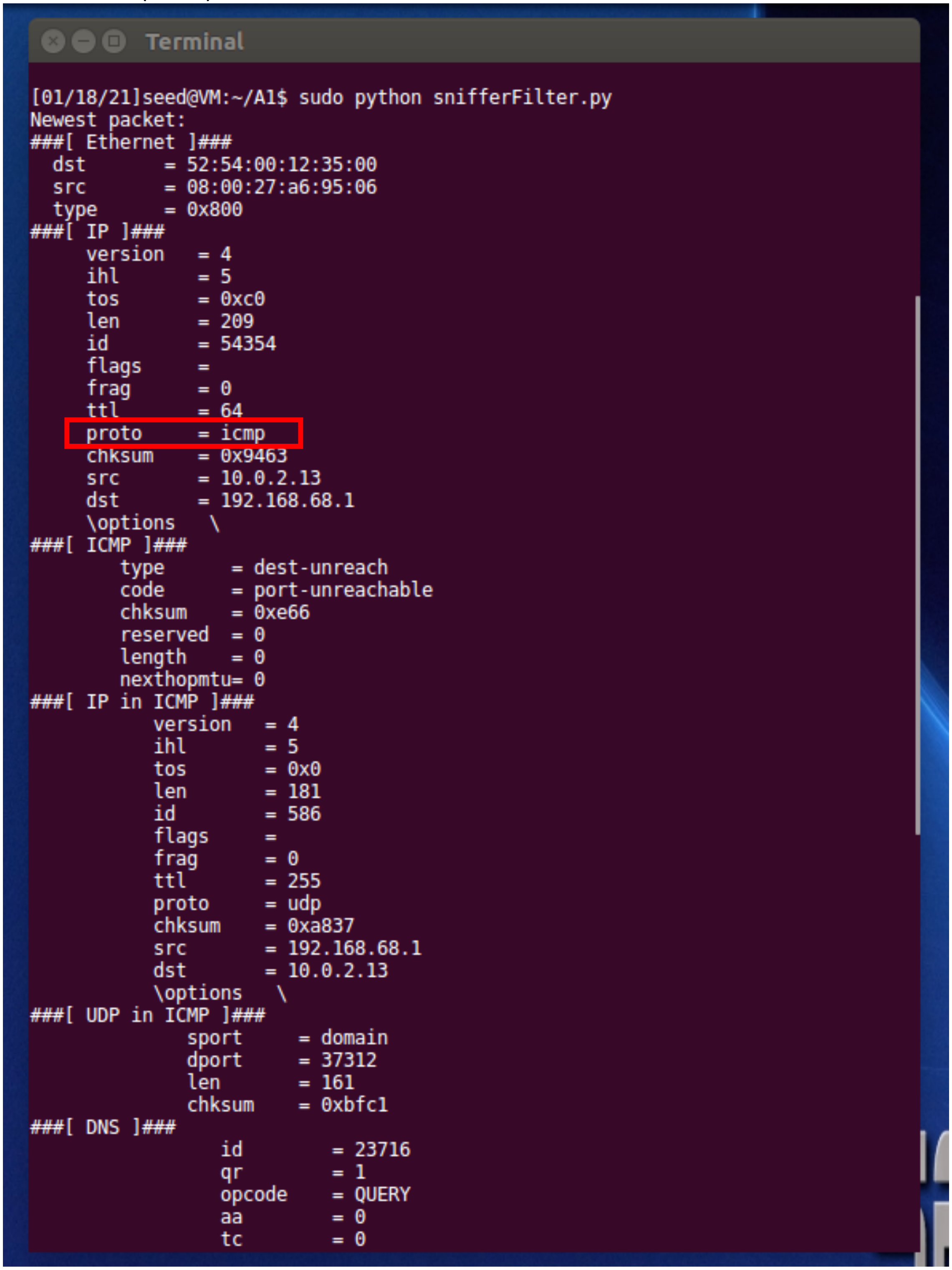
Exhibit 5: Python code, with filter specific for ICMP packets.



```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
        # Helps see when new packets come in
        print('Newest packet:')
        pkt.show()

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

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 6: Snippet of output from python code shown in Exhibit 5, displaying the appropriate protocol of interest (ICMP).



```
Terminal
[01/18/21]seed@VM:~/A1$ sudo python snifferFilter.py
Newest packet:
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:a6:95:06
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0xc0
     len       = 209
     id        = 54354
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x9463
     src       = 10.0.2.13
     dst       = 192.168.68.1
     \options   \
###[ ICMP ]###
        type      = dest-unreach
        code      = port-unreachable
        chksum    = 0xe66
        reserved  = 0
        length    = 0
        nexthopmtu= 0
###[ IP in ICMP ]###
           version   = 4
           ihl       = 5
           tos       = 0x0
           len       = 181
           id        = 586
           flags     =
           frag      = 0
           ttl       = 255
           proto     = udp
           chksum    = 0xa837
           src       = 192.168.68.1
           dst       = 10.0.2.13
           \options   \
###[ UDP in ICMP ]###
              sport     = domain
              dport     = 37312
              len       = 161
              chksum    = 0xbfc1
###[ DNS ]###
                 id        = 23716
                 qr        = 1
                 opcode    = QUERY
                 aa        = 0
                 tc        = 0
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

- Capturing any TCP packet that comes from a particular IP and with a destination port number 23. Note: 10.0.2.15 is the other VM which I have spun up, which I attempted to telnet from in order to create some network traffic to display.

Exhibit 7: Python code to capture tcp traffic from host 10.0.2.15 on port 23.

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
        # Helps see when new packets come in
        print('Newest packet:')
        pkt.show()

pkt = sniff(filter='tcp and src 10.0.2.15 and dst port 23',prn=print_pkt)
```

Exhibit 8: Displaying a portion of the traffic which was captured which I attempted to run the telnet command to the specific host. I noted that the source (src) IP address was indeed 10.0.2.15, and the destination port (dport) is telnet, which is indeed run on port 23.

```
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 32570
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xa35e
     src       = 10.0.2.15
     dst       = 10.0.2.13
     \options   \
###[ TCP ]###
        sport     = 48526
        dport     = telnet
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

- Capture packets that coming from or go to a particular subnet.
    - Chosen subnet: 192.168.1.0/24

Exhibit 9: Python code to capture network traffic to or from the chosen subnet.

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter= "src net 192.168.1.0/24 or dst net 192.168.1.0/24",prn=print_pkt)
```

Exhibit 10: Snippet of a packet which was captured by the script. This displays the appropriate subnet which was configured within the script.

```
        chksum    = 0xcdd0
        urgptr    = 0
        options   = []

###[ Ethernet ]###
  dst       = 08:00:27:a6:95:06
  src       = 52:54:00:12:35:00
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 44
     id        = 4884
     flags     =
     frag      = 0
     ttl       = 255
     proto     = tcp
     chksum    = 0xdb01
     src       = 192.168.1.1
     dst       = 10.0.2.13
     \options   \
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Task 1.2: Spoofing ICMP Packets

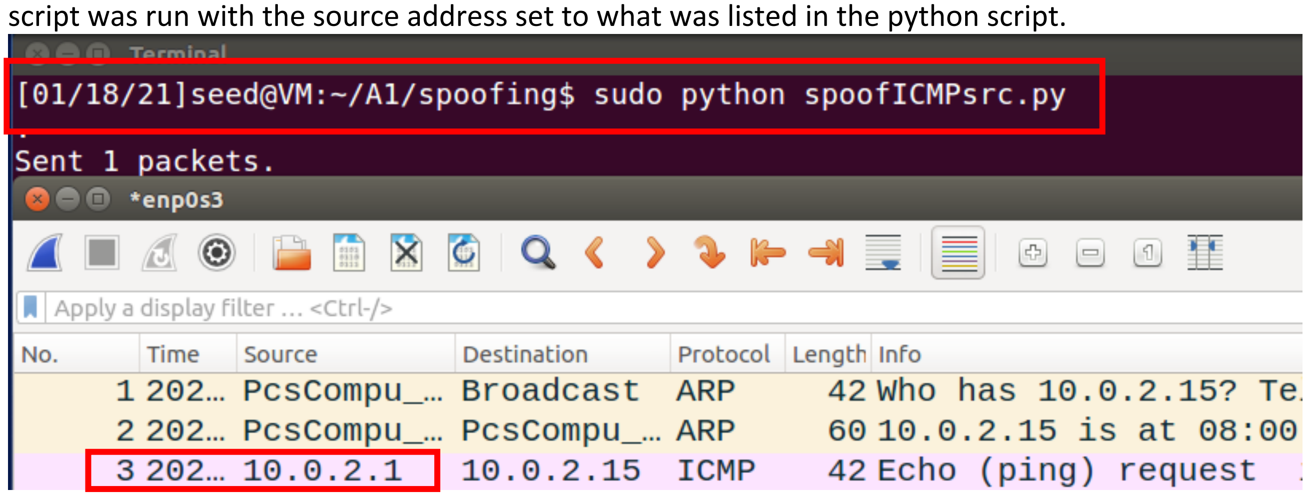
Exhibit 11: Displaying the IP address of the requesting machine. I noted that the address of the sending machine is 10.0.2.13.

```
[01/18/21]seed@VM:~/A1/spoofing$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:a6:95:06
          inet addr:10.0.2.13  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::f623:bc31:47df:bd32/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

Exhibit 12: Python script to send the spoofed ICMP packet to the other host (10.0.2.15) on the network. I ensured to change the source IP address to an IP address other than the original sender's IP address (10.0.2.1 instead of the original 10.0.2.13).

```
Terminal
from scapy.all import *
a = IP()
a.src = '10.0.2.1'
a.dst = '10.0.2.15'
b = ICMP()
spoofed_packet = a/b
send(spoofed_packet)
~
```

Exhibit 13: Wireshark capture, displaying that the spoofed packet was indeed sent after the script was run with the source address set to what was listed in the python script.



Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Task 1.3: Traceroute

Exhibit 14: Python program which emulates traceroute via changing the ttl field. I selected the destination IP address of 1.1.1.1 to trace. My code also includes a range for hops that can be adjusted, as needed (this is used instead of performing the manual changes in the ttl field).

```
from scapy.all import *

for hop in range(1,25):

        a = IP()
        a.dst = '1.1.1.1'
        a.ttl = hop
        b = ICMP()
        spoofed_packet = a/b
        send(spoofed_packet)
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 15: Wireshark output, displaying that ICMP errors were indeed captured until the packet was able to reach the destination IP address of 1.1.1.1. One optimization I could have made to the code is to break out of the loop once a packet was received from 1.1.1.1, but having a loop which ran greater than the number of hops only sent additional ICMP packets. The Wireshark output still allowed me to track the IP addresses for each router along the way to my intended destination.

2	2021-…	RealtekU_12:3…	PcsCompu_a6:9…	ARP	60	10.0.2.1 is at 52:54:00:12:35:00
3	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
4	2021-…	10.0.2.1	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live exceede…
5	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
6	2021-…	192.168.68.1	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live exceede…
7	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
8	2021-…	192.168.1.1	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live exceede…
9	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
10	2021-…	50.46.181.18	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live exceede…
11	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
12	2021-…	64.52.96.4	10.0.2.13	ICMP	110	Time-to-live exceeded (Time to live exceede…
13	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
14	2021-…	137.83.80.22	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live exceede…
15	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
16	2021-…	137.83.80.20	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live exceede…
17	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
18	2021-…	137.83.80.4	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live exceede…
19	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
20	2021-…	137.83.80.6	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live exceede…
21	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
22	2021-…	107.191.236.1…	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live exceede…
23	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
24	2021-…	107.191.236.65	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live exceede…
25	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
26	2021-…	107.191.236.1…	10.0.2.13	ICMP	110	Time-to-live exceeded (Time to live exceede…
27	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
28	2021-…	198.32.195.95	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live exceede…
29	2021-…	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt…
30	2021-…	1.1.1.1	10.0.2.13	ICMP	60	Echo (ping) reply id=0x0000, seq=0/0, tt…

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Task 1.4: Sniffing and-then Spoofing

Below is evidence displaying that the given host is unreachable

Exhibit 16: Upon attempting to ping 10.0.2.16 (another VM I spun up), I do not receive a reply.

```
[01/21/21]seed@VM:~$ ping 10.0.2.16
PING 10.0.2.16 (10.0.2.16) 56(84) bytes of data.
From 10.0.2.15 icmp_seq=10 Destination Host Unreachable
From 10.0.2.15 icmp_seq=11 Destination Host Unreachable
From 10.0.2.15 icmp_seq=12 Destination Host Unreachable
From 10.0.2.15 icmp_seq=13 Destination Host Unreachable
From 10.0.2.15 icmp_seq=14 Destination Host Unreachable
From 10.0.2.15 icmp_seq=15 Destination Host Unreachable
^C
```

Exhibit 17: Original python script to spoof the ICMP echo reply (prior to making needed adjustments).

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

def spoof_ICMP(pkt):
        #Once an icmp packet is seen on the traffic, spoof an ICMP packet
        a = IP()

        # Set the dst IP to originator's IP address
        a.dst=pkt[IP].src

        # Set src to be the original dst of the request
        a.src= pkt[IP].dst

        b = ICMP(type="echo-reply")
        spoofed_packet = a/b
        send(spoofed_packet)

pkt = sniff(filter='icmp[icmptype] == icmp-echo',prn=spoof_ICMP)
```

While Exhibit 17 displays code that spoofs relevant packets, it does not produce the desired result on the target machine which is running the "Ping" command. Looking at examples of legitimate traffic (non-spoofed) via wireshark reveals a few key things: The sequence number, identification number, and payloads seemed to be the same between the ICMP request and ICMP reply. Upon making these adjustments, as seen below, the desired result occurs.

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

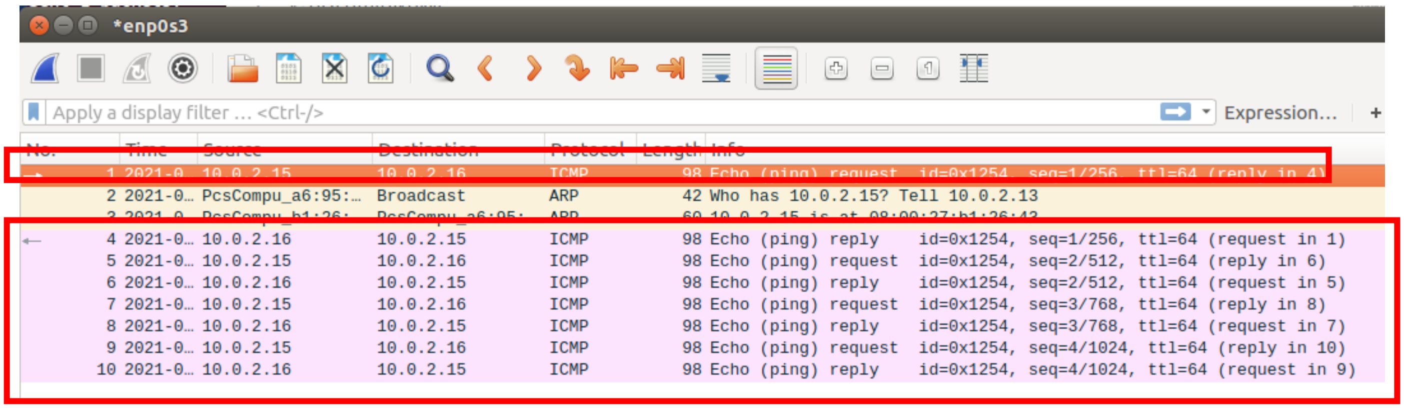
Exhibit 18: Updated python code, accounting for id, sequence number, and payload.

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

def spoof_ICMP(pkt):
    #Once an icmp packet is seen on the traffic, spoof an ICMP packet
    a = IP()

    # Set the dst IP to originator's IP address
    a.dst=pkt[IP].src

    # Set src to be the original dst of the request
    a.src= pkt[IP].dst

    # Set type of ICMP packet, along with sequence and id fields
    b = ICMP(type="echo-reply")
    b.seq = pkt[ICMP].seq
    b.id = pkt[ICMP].id

    # Set the data field based off of what we sniff so that the data fields are the same
    # This data is found in the [Raw] section of the packet.
    payload = pkt[Raw].load

    spoofed_packet = a/b/payload
    send(spoofed_packet)

pkt = sniff(filter='icmp[icmptype] == icmp-echo',prn=spoof_ICMP)
```

Exhibit 19: Upon attempting to ping 10.0.2.16, I was provided with ICMP echo replies, indicating success of the python script.

```
Base Ubuntu Machine 2 (Clone) [Running]
Terminal
[01/22/21]seed@VM:~$ ping 10.0.2.16
PING 10.0.2.16 (10.0.2.16) 56(84) bytes of data.
64 bytes from 10.0.2.16: icmp_seq=1 ttl=64 time=68.4 ms
64 bytes from 10.0.2.16: icmp_seq=2 ttl=64 time=23.4 ms
64 bytes from 10.0.2.16: icmp_seq=3 ttl=64 time=21.4 ms
64 bytes from 10.0.2.16: icmp_seq=4 ttl=64 time=32.4 ms
64 bytes from 10.0.2.16: icmp_seq=5 ttl=64 time=31.6 ms
64 bytes from 10.0.2.16: icmp_seq=6 ttl=64 time=25.8 ms
64 bytes from 10.0.2.16: icmp_seq=7 ttl=64 time=24.1 ms
64 bytes from 10.0.2.16: icmp_seq=8 ttl=64 time=18.2 ms
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021
Exhibit 20: Wireshark output with successful python script returning ICMP replies



Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

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

Task 2.1: Writing Packet Sniffing Program

Question 1: Describe the sequence of the library calls that are essential for sniffer programs.

Based off of the code that has been provided for the sniffer program, at a high level, the sequence of library calls that are essential could be seen as the following:

1. Open a packet capturing session on the specified network interface
2. Create the packet filter that is to be applied to the packet capturing session
3. Set the packet filter onto the session, resulting in the filtering of the packets
4. Begin capturing packets with the sniffer program.

Exhibit 21: High level sequence for library calls that are essential for sniffer programs, as indicated by the provided code for the sniffer program.

```
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 pcp 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 handle
        return 0;
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Question 2: Why do you need root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

Root privilege is needed to run the sniffer program because it requires a connection to the network interface. Such a connection allows any program to see all the network traffic coming from the given host.

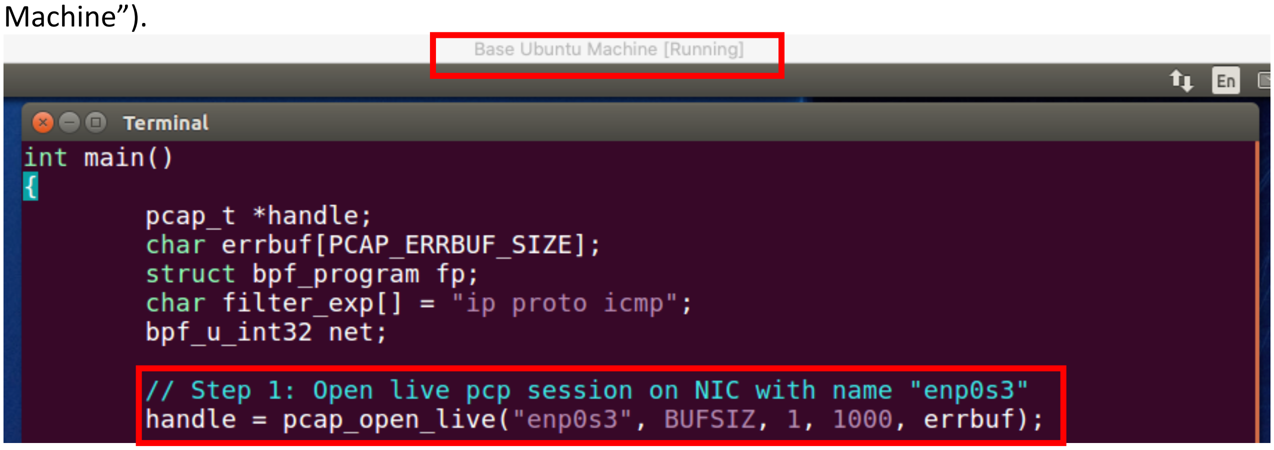
The program will fail at the pcap_open_live command (step 1 in Exhibit 21) if it is executed without the root privilege, because the attempt to open the connection will be denied.

Question 3: Turn on and 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.

The difference between when the promiscuous mode is on or off is the network traffic that is captured. Having promiscuous mode on allows the packet sniffer to capture all network traffic that is on the network. Having promiscuous mode off only captures network traffic that is intended for the given machine.

This can be demonstrated via a second machine creating network traffic.

Exhibit 22: Turning on promiscuous mode on "Machine A" (Mine is called "Base Ubuntu Machine").



Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 23: Noting that "Machine B" (which is called "Base Ubuntu Machine 2") attempting to connect to a website has its network traffic captured by Machine A.

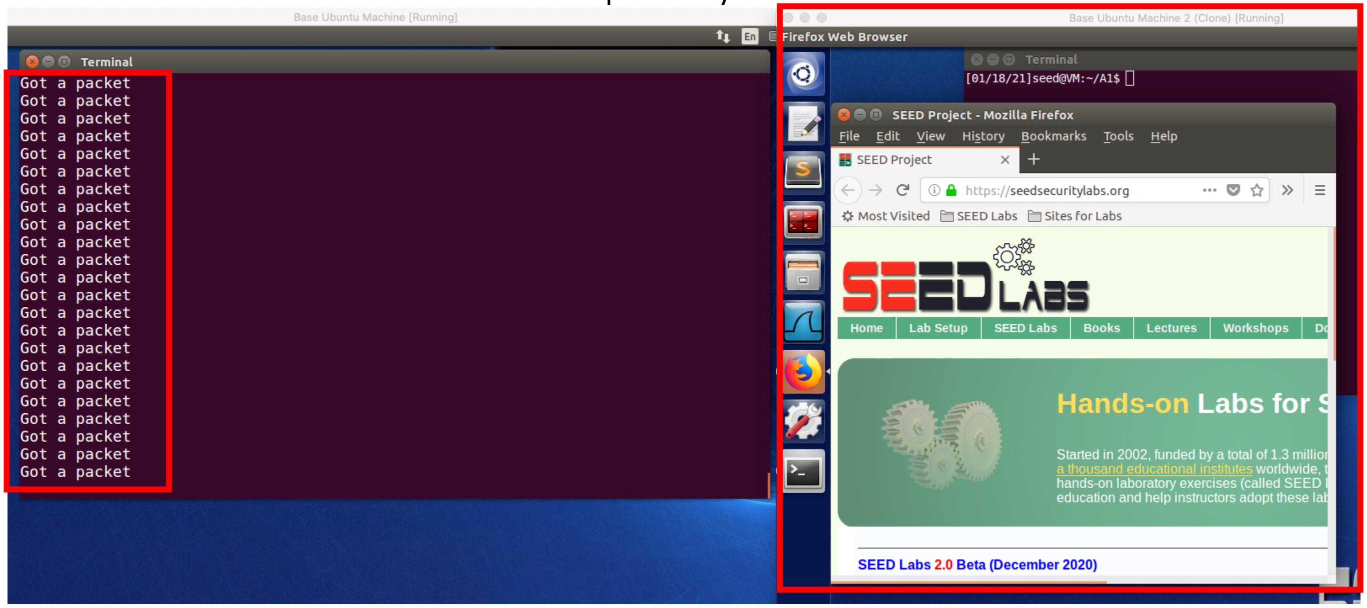
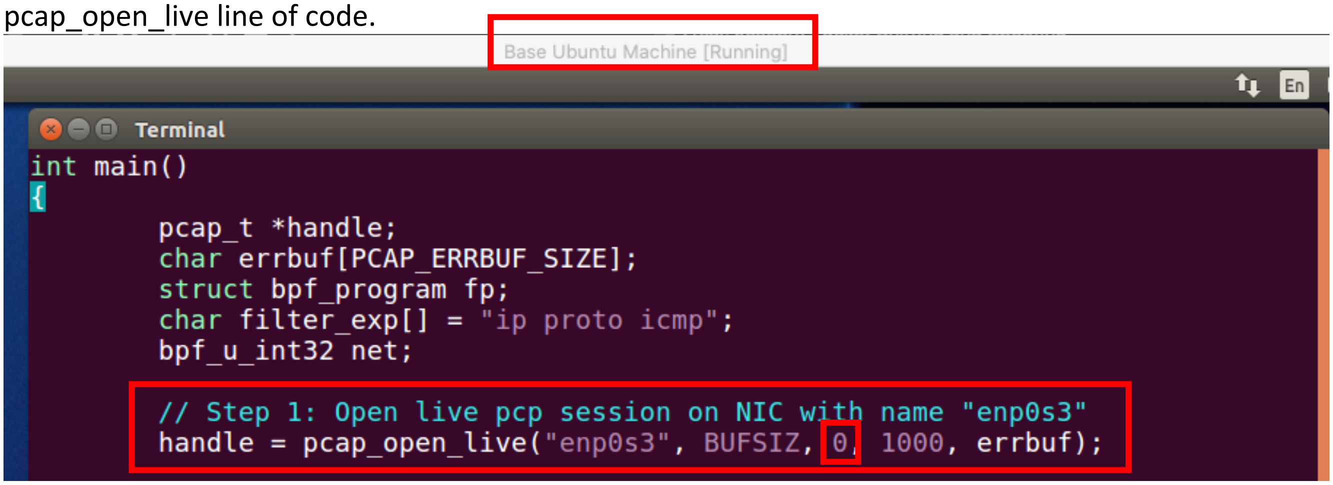


Exhibit 24: Turning off promiscuous mode on Machine A, as seen by the 0 in the pcap_open_live line of code.



Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 25: Connecting to the website on Machine B, which does not display any network traffic captured by Machine A.

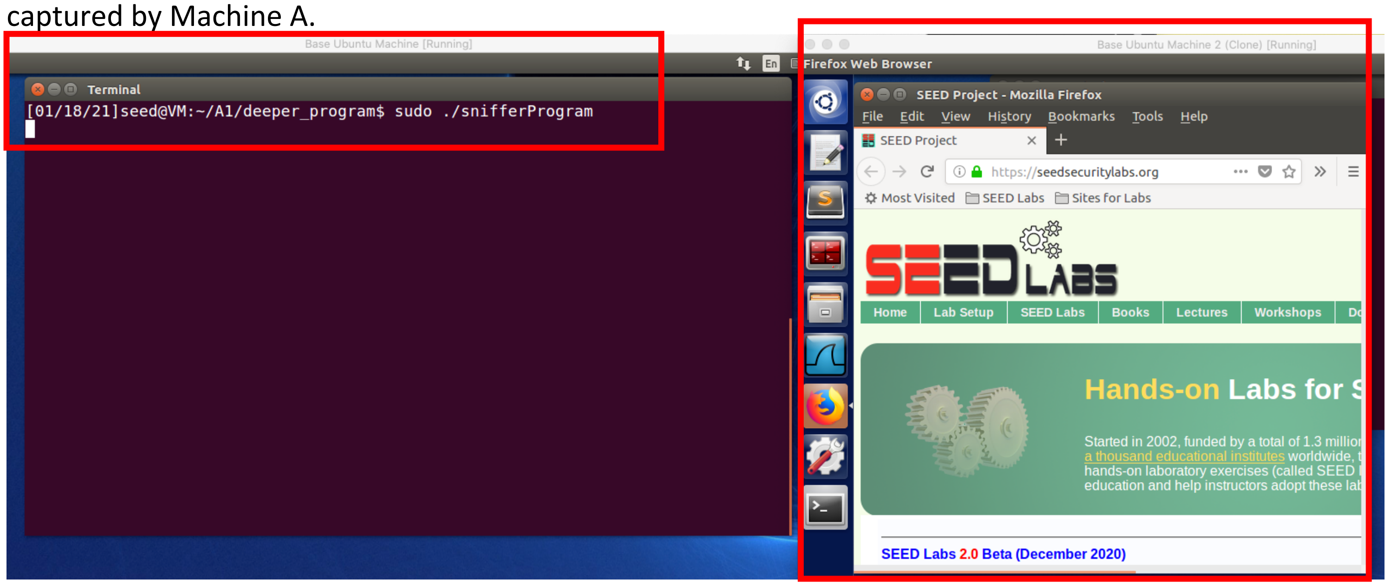
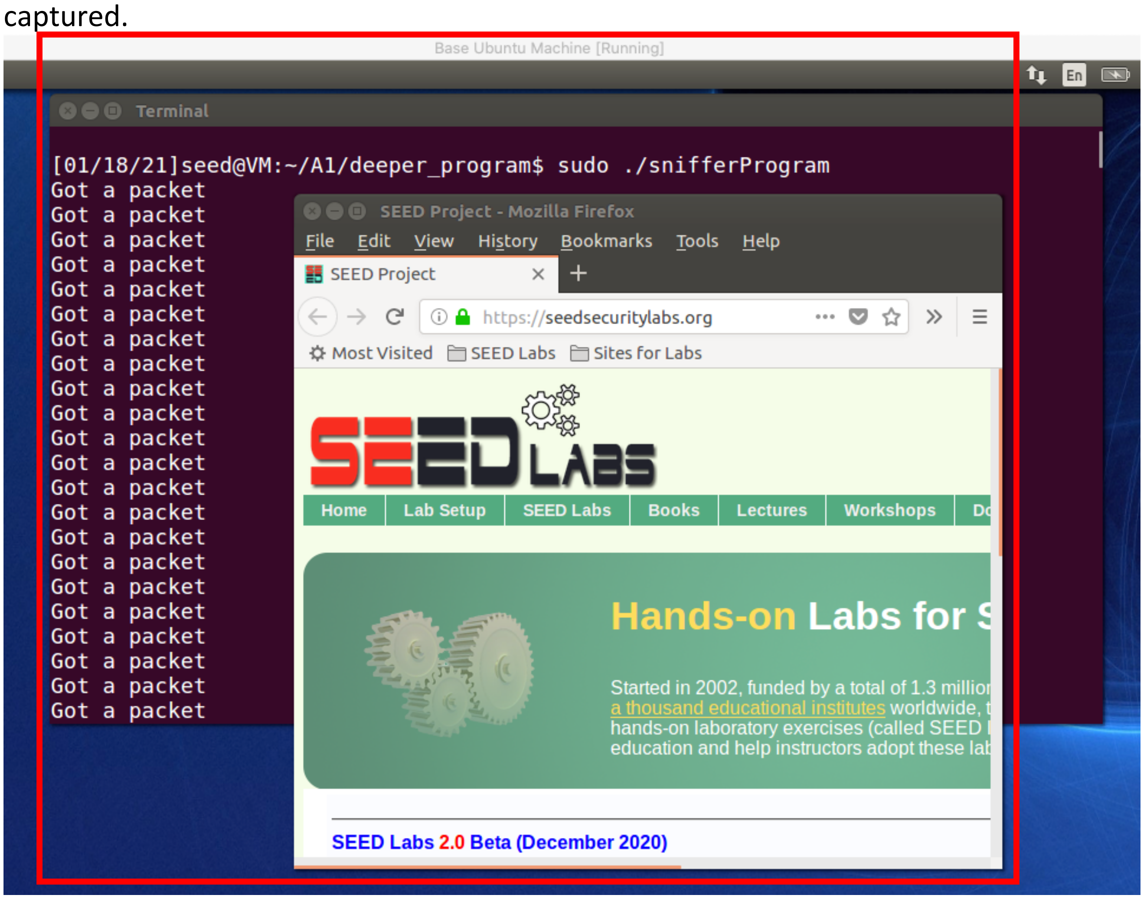


Exhibit 26: Connecting to a website on Machine A, which indeed displays network traffic being captured.



Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Task 2.1B: Writing Filters

- Capturing ICMP packets between two specific hosts.

  I chose 10.0.2.15 and 1.1.1.1 for my specific hosts. The document referenced by the lab greatly assisted me with printing out the relevant fields, as I was not familiar with C programming before: https://www.tcpdump.org/pcap.html

Exhibit 27: Code displaying the filter I applied to capture the relevant traffic.

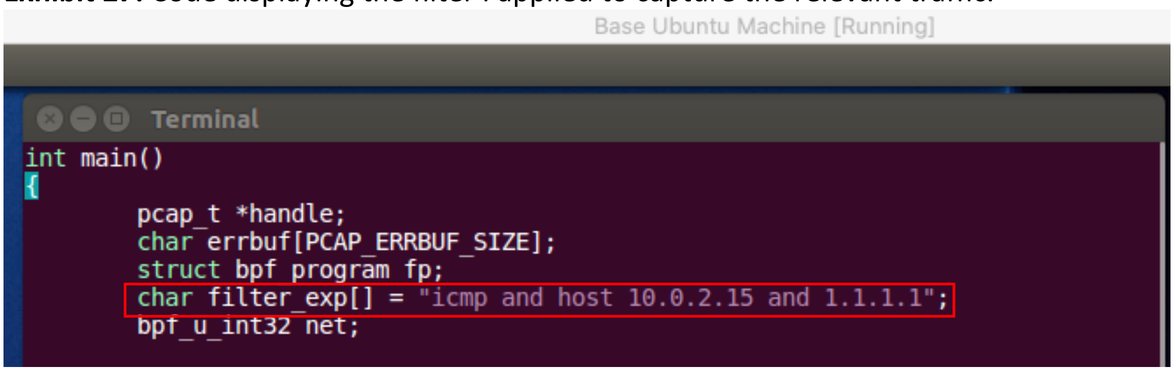


Exhibit 28: Short program I wrote on Machine B to send an icmp packet specifically to 1.1.1.1.

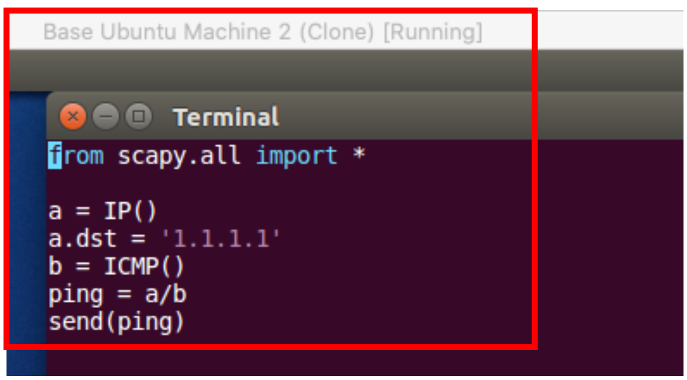
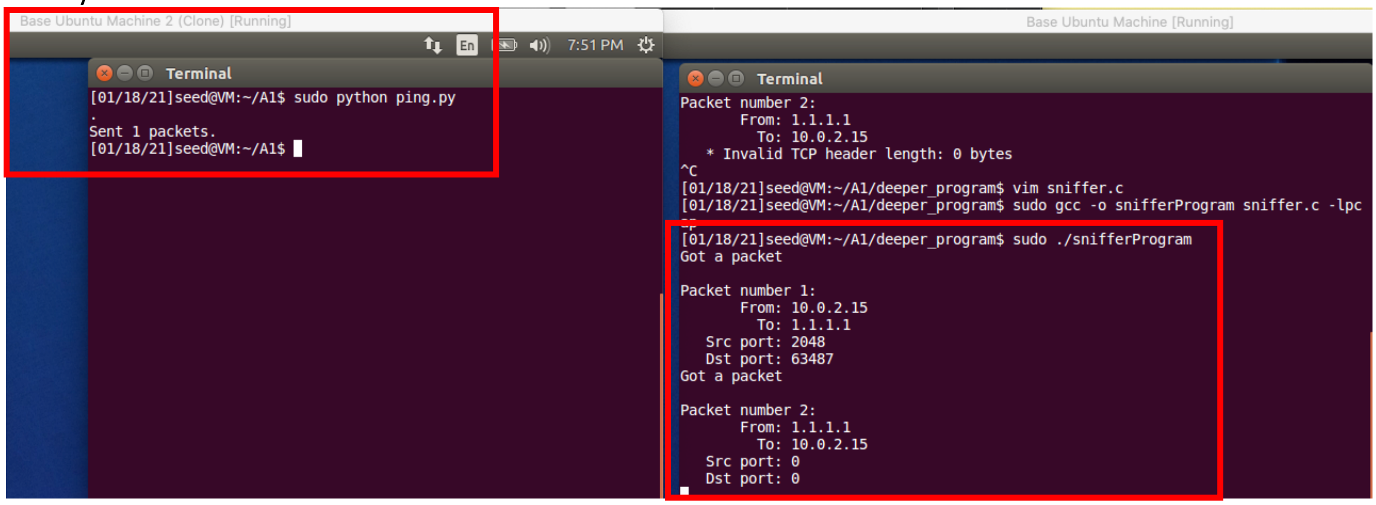
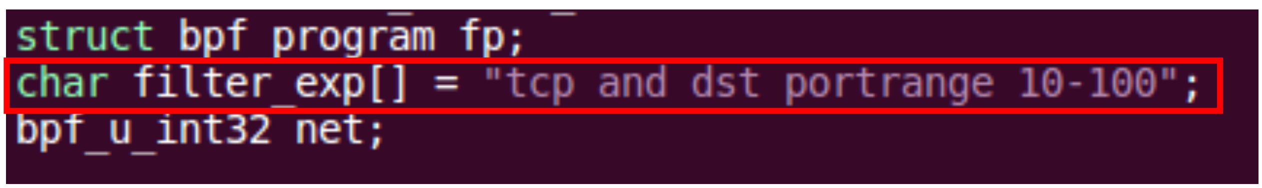


Exhibit 29: Running the ping to 1.1.1.1, which was captured by the snifferProgram. This is clearly seen in the traffic between 10.0.2.15 and 1.1.1.1.



- Capturing TCP packets with a destination port in the range from 10 to 100.

Exhibit 30: Filter I used to capture TCP packets with a destination port in the applicable port range.



Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

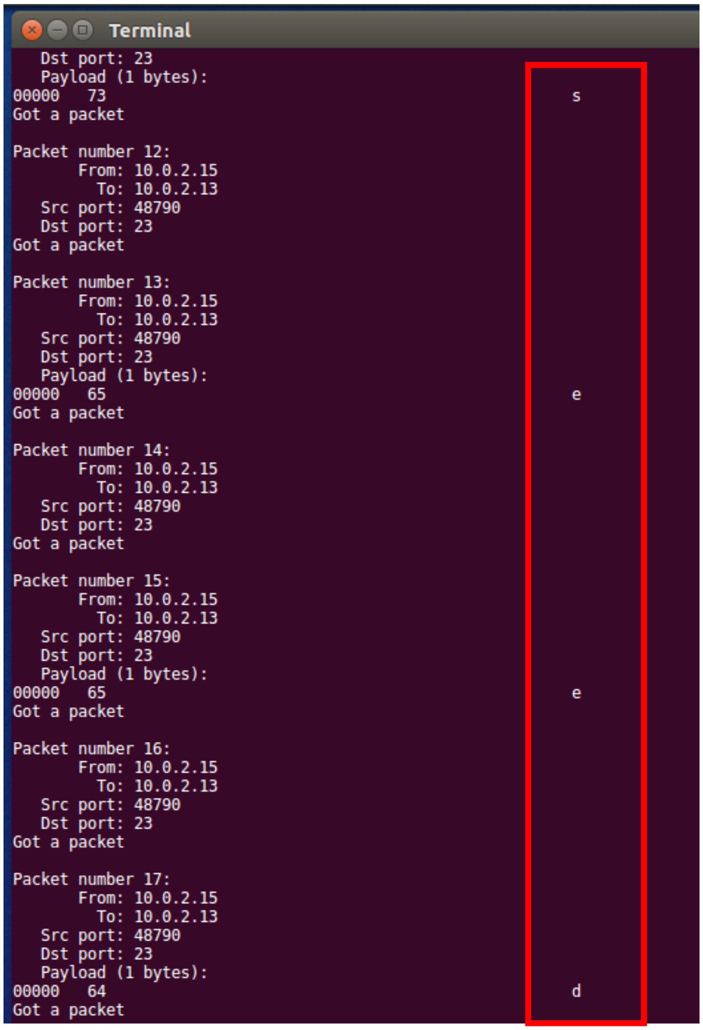
Exhibit 31: Snippet of network traffic, displaying that only TCP traffic has been captured within the specified destination port range.

```
Terminal
Packet number 1:
       From: 10.0.2.13
         To: 34.107.221.82
   Src port: 49562
   Dst port: 80
Got a packet

Packet number 2:
       From: 10.0.2.13
         To: 34.107.221.82
   Src port: 49562
   Dst port: 80
Got a packet

Packet number 3:
       From: 10.0.2.13
         To: 34.107.221.82
   Src port: 49562
   Dst port: 80
Got a packet

Packet number 4:
       From: 10.0.2.13
         To: 34.107.221.82
   Src port: 49562
   Dst port: 80
Got a packet

Packet number 5:
       From: 10.0.2.13
         To: 128.230.247.70
   Src port: 56590
   Dst port: 80
Got a packet

Packet number 6:
       From: 10.0.2.13
         To: 128.230.247.70
   Src port: 56590
   Dst port: 80
Got a packet
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Task 2.1C: Sniffing Passwords

Exhibit 32: Printing the data field of the packets, displaying the username "seed" being sent over the network.



Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 33: Printing the data field of the packets, which displays the password “dees” being sent over the network as well.

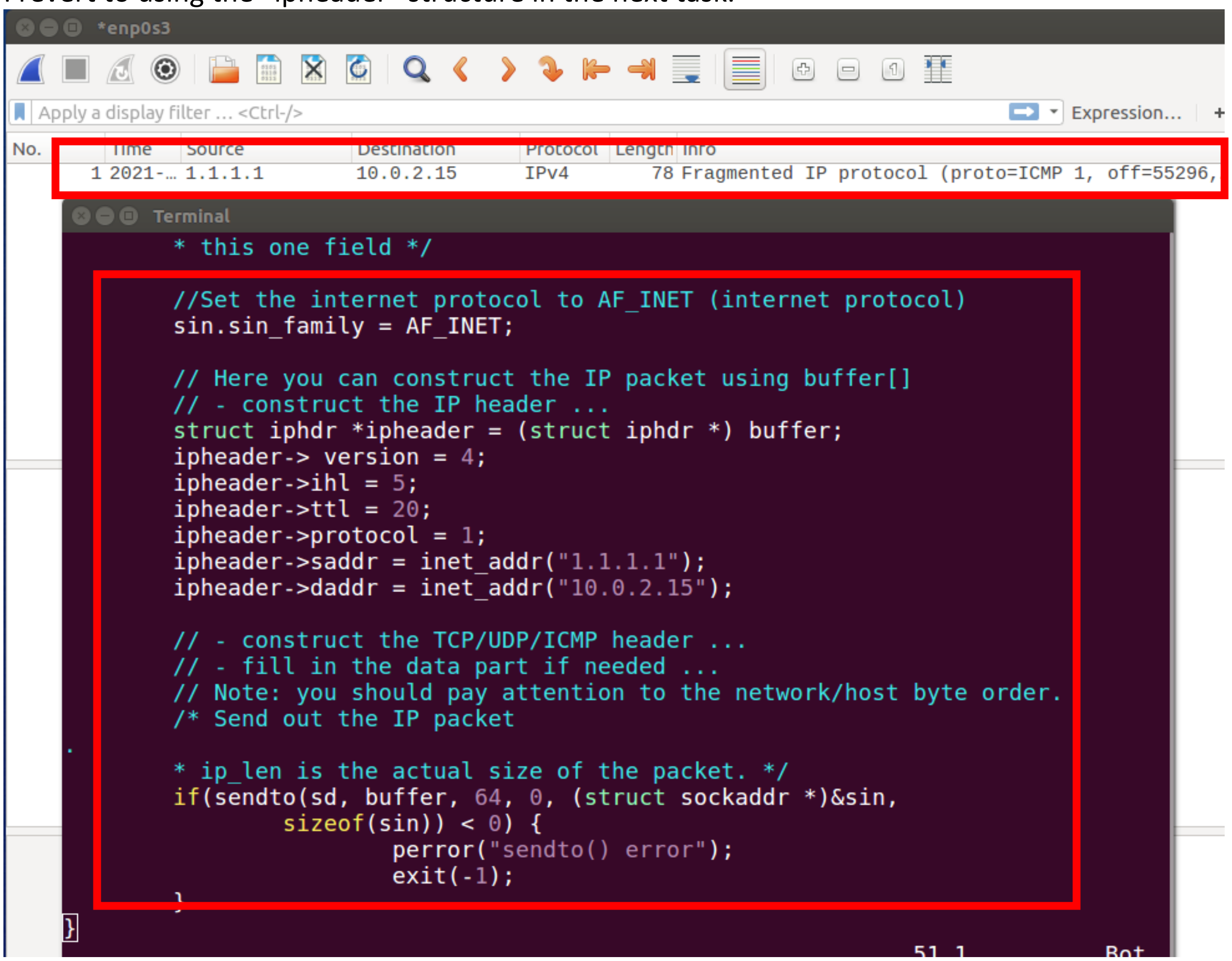
```
Packet number 22:
       From: 10.0.2.15
         To: 10.0.2.13
   Src port: 48790
   Dst port: 23
   Payload (1 bytes):
00000   64                                                    d
Got a packet

Packet number 23:
       From: 10.0.2.15
         To: 10.0.2.13
   Src port: 48790
   Dst port: 23
   Payload (1 bytes):
00000   65                                                    e
Got a packet

Packet number 24:
       From: 10.0.2.15
         To: 10.0.2.13
   Src port: 48790
   Dst port: 23
   Payload (1 bytes):
00000   65                                                    e
Got a packet

Packet number 25:
       From: 10.0.2.15
         To: 10.0.2.13
   Src port: 48790
   Dst port: 23
   Payload (1 bytes):
00000   73                                                    s
Got a packet
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Task 2.2A: Write a spoofing program

Exhibit 34: Snippet of spoofing program based on the provided skeleton code which sets the fields of the ipheader, along with packet capture within Wireshark. I noted that the info stated that the packet was a “Fragmented IP protocol” because I had not yet set the other items needed for a complete packet. However, for purposes of displaying that packets are indeed spoofed, this script does produce an applicable result. I tackle this notification of a fragmented IP protocol within Task 2.3: Spoof an ICMP, below.

Furthermore, I did not realize that the reference code provided contained the “ipheader” structure which is referenced in the sample code. Hence, I used the iphdr structure for this task. I revert to using the “ipheader” structure in the next task.



Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Task 2.2B: Spoof an ICMP Echo Request (screenshot displaying the successful result is at the end of this task)

Exhibit 35: Code displaying what was configured within the program to spoof the request. These are the import statements which I used, along with the structures that were provided in the reference code.

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

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

/* IP Header */
struct ipheader {
  unsigned char      iph_ihl:4, //IP header length
                     iph_ver:4; //IP version
  unsigned char      iph_tos; //Type of service
  unsigned short int iph_len; //IP Packet length (data + header)
  unsigned short int iph_ident; //Identification
  unsigned short int iph_flag:3, //Fragmentation flags
                     iph_offset:13; //Flags offset
  unsigned char      iph_ttl; //Time to Live
  unsigned char      iph_protocol; //Protocol type
  unsigned short int iph_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 36: ICMP spoofing code, continued. This is the checksum function which is provided in the reference code.

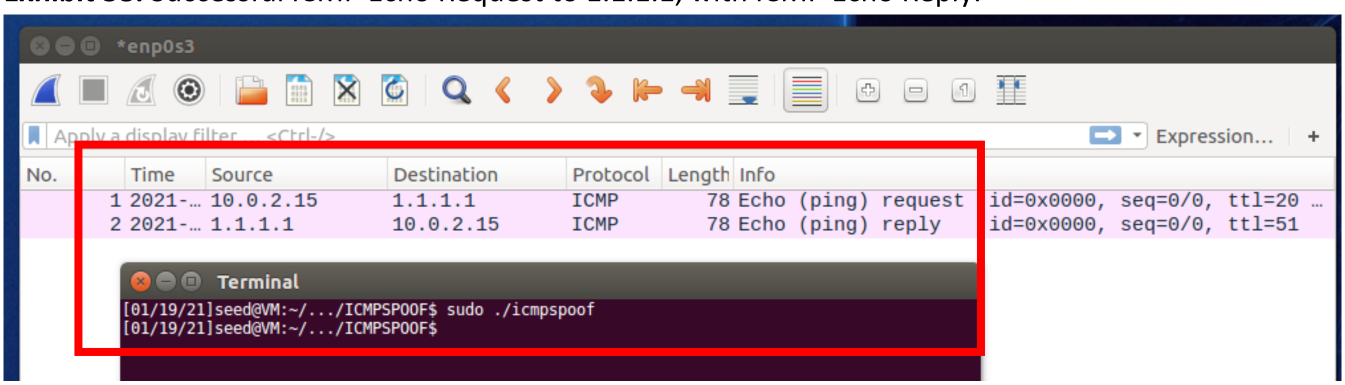
```
unsigned short in_cksum (unsigned short *buf, int length)
{
   unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;

   /*
    * 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);
}
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 37: Code snippet which displays the creation of the ICMP and IP headers. These configurations are reflected in the Wireshark capture in Exhibit 38. The majority of this code was also provided as reference material on the Canvas page.

```
    // Create icmp header
    struct icmpheader *icmp = (struct icmpheader *)
                             (buffer + sizeof(struct ipheader));
    icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

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

    // create ipheader

    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("10.0.2.15");
    ip->iph_destip.s_addr = inet_addr("1.1.1.1");
    ip->iph_protocol = IPPROTO_ICMP;
    ip->iph_len = htons(sizeof(struct ipheader) +
                  sizeof(struct icmpheader));
```

Exhibit 38: Successful ICMP Echo Request to 1.1.1.1, with ICMP Echo Reply.



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

Yes, it appears that we are able to set the IP packet length to an arbitrary value, as the length of the packet continually returns to its original size. I attempted the following values with no changes in result:

- 0
- 10
- 100
- 1000

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

I noted that attempting numbers such as 1000000 that I was produced with a warning, but I believe that this is due to the usage of a unsigned short int field type that is used for the ipheader structure.

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

No, I do not believe we need to calculate the checksum for the IP headers when using raw socket programming. It seems that the checksum for the IP header is calculated by the machine when sending the packets out, regardless if I set it myself within the packet.

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?

You need root privileges to run the programs that use raw sockets because using raw sockets presents the chance to interfere with the standard network configurations of the machine. Hence, this could alter the traffic which is entering into the machine and can be seen as privileged access.

The program fails at the line where the socket is first declared if it is not run with root privileges.

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Task 2.3: Sniff and Spoof

The following screenshots display the code that I used to sniff and spoof ICMP replies.

Exhibit 39: Function which opens the sniffing session and calls the function, "got_packet" whenever a packet is received that is of type "icmp-echo".

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

    // Step 1: Open live pcp 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 handle
    return 0;

    //Compilation command: gcc -o sniffer sniff.c -lcap
}
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 40: Initial portion of my "got_packet" function. The structures which are defined can be found in the provided reference code for the assignment. At a high level, this code defines the initial structures that I later use to spoof the ICMP packet through analyzing the packet that has been sniffed.

```
// Function invoked by pcap for each captured packet.
void got_packet(u_char *args, const struct pcap_pkthdr *header,
    const u_char *packet)

{
    printf("Got a packet\n");

    static int count = 1;                   /* packet counter */

    /* declare pointers to packet headers */
    const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
    const struct sniff_ip *ip;              /* The IP header */
    const struct icmpheader *captured_icmphdr; /*ICMP header*/
    int size_ip;

    printf("\nPacket number %d:\n", count);
    count++;

    /* define ethernet header */
    ethernet = (struct sniff_ethernet*)(packet);

    /* define/compute ip header offset */
    ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
    size_ip = IP_HL(ip)*4;

    // Check if the ip header length is an appropriate size (also given by reference code)
    if (size_ip < 20) {
        printf("   * Invalid IP header length: %u bytes\n", size_ip);
        return;
    }

    /* print source and destination IP addresses */
    printf("       From: %s\n", inet_ntoa(ip->ip_src));
    printf("         To: %s\n", inet_ntoa(ip->ip_dst));


    // Define icmpheader to obtain information regarding captured ICMP packet
    captured_icmphdr = (struct icmpheader*)(packet + SIZE_ETHERNET + size_ip);
```

Exhibit 41: Creation of the raw socket, buffer to which structures will be cast, and creation of the ICMP packet based on the sniffed ICMP packet.

```
/****** Begin construction of the spoofed ICMP packet *********/
int sd;
    struct sockaddr_in sin;
    char buffer[1550]; // You can change the buffer size
    /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
    * tells the sytem that the IP header is already included;
    * this prevents the OS from adding another IP header. */
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if(sd < 0) {
            perror("socket() error");
            exit(-1);
    }

    //Set the internet protocol to AF_INET (internet protocol)
    sin.sin_family = AF_INET;

    /** Change this line here to be the right MAC address **/
    // Set the address to be that of the source address of which was captured
    sin.sin_addr.s_addr = ip->ip_dst.s_addr;

    memset(buffer, 0, 1550);

    // Create spoofed icmp header
    struct icmpheader *icmp = (struct icmpheader *)
                            (buffer + sizeof(struct ipheader));

    /*Set relevant fields to spoof the icmp header*/
    icmp->icmp_type = 0;
    icmp->icmp_id = captured_icmphdr->icmp_id;

    // Use the same sequence number as the captured packet
    icmp->icmp_seq = captured_icmphdr -> icmp_seq;

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

Exhibit 42: Creation of the IP header, with relevant fields set based on what was sniffed by the packet sniffer.

```
/********** Create IPheader ********/

// create ipheader
struct ipheader *ipspoof = (struct ipheader *) buffer;
ipspoof->iph_ver = 4;
ipspoof->iph_ihl = 5;
ipspoof->iph_ttl = 64;
ipspoof->iph_sourceip = ip->ip_dst;


// Send the packet back to the requestor
//ipspoof->iph_destip.s_addr = inet_addr("1.1.1.1");
ipspoof->iph_destip = ip->ip_src;
ipspoof->iph_protocol = IPPROTO_ICMP;
ipspoof->iph_len = htons(sizeof(struct ipheader) +
                sizeof(struct icmpheader)
                + sizeof(struct sniff_ethernet));
```

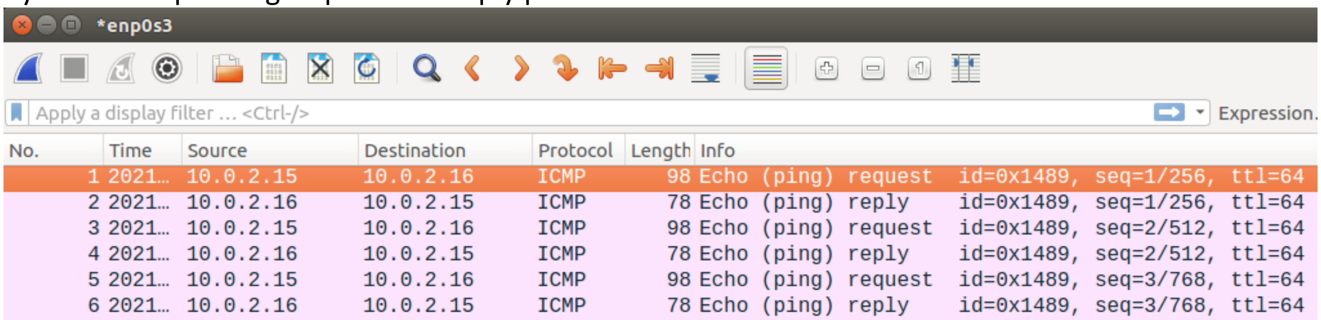
Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 43: Creation of the data field within the spoofed packet. I took the data from the sniffed packet and reassigned it to the buffer which I created in Exhibit 41. This allows the program to have the same data field as that of the sniffed packet. The offset for the buffer is defined via the size of the structures which create the overall spoofed packet. Once the packet is created, it is then sent within the if statement at the bottom of the function. The if statement attempts to send the packet. If it cannot, it will exit the spoofing program and print out the relevant error.

```
        // Calculate the data to add to the packet
        char *data = (u_char *)packet +
                            sizeof(struct sniff_ethernet) +
                            sizeof(struct ipheader) +
                            sizeof(struct sniff_tcp);

        // Create a corresponding pointer that's within the buffer, at the end of the buffer
        char *data_pointer = (u_char *)buffer +
                                sizeof(struct sniff_ethernet) +
                                sizeof(struct ipheader) +
                                sizeof(struct sniff_tcp);


        /****Construct data for spoofed packet***/
        // Understand how large the data is so that we can understand how long to loop for
        int size_data = ntohs(ip->ip_len) - (sizeof(struct ipheader));
        if (size_data > 0) {

            // Iterate through the data and make it the same as what is within the request packet
            for (int i = 0; i < size_data; i++) {

                *data_pointer = *data;
                data++;
                data_pointer++;
            }
        }

        // Documentation regarding https://pubs.opengroup.org/onlinepubs/009695399/functions/sendto.html
        // how to use the sendto() function.
        // First parameter = "socket"
        // Second parameter = "message", which is defined as "A buffer containing the message to be sent"
        if(sendto(sd, buffer, 64, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
            perror("sendto() error");
            exit(-1);
        }
        printf("SPOOFED PACKET HAS BEEN SENT");
}
```

Joseph Tsai
CSS 537 – Assignment 1: Packet Sniffing and Spoofing
January 16th, 2021

Exhibit 44: Wireshark capture displaying the spoofed packets created by the program, as seen by the corresponding request and reply packets.



No.	Time	Source	Destination	Protocol	Length	Info
1	2021…	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1489, seq=1/256, ttl=64
2	2021…	10.0.2.16	10.0.2.15	ICMP	78	Echo (ping) reply id=0x1489, seq=1/256, ttl=64
3	2021…	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1489, seq=2/512, ttl=64
4	2021…	10.0.2.16	10.0.2.15	ICMP	78	Echo (ping) reply id=0x1489, seq=2/512, ttl=64
5	2021…	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1489, seq=3/768, ttl=64
6	2021…	10.0.2.16	10.0.2.15	ICMP	78	Echo (ping) reply id=0x1489, seq=3/768, ttl=64