**1. Using Tools to Sniff and Spoof Packets**

**Task 1.1 Sniffing Packets**

**Task 1.1a**

The python code for sniffing ICMP packets:

```
Open ▾

#!/usr/bin/python3
# -*- coding: utf-8 -*-
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='icmp',prn=print_pkt)
```

**Setup:** I have 3 VM's running. One be attacker machine and other 2 legit users with ip 10.0.2.6 and 10.0.2.5. from my attacking machine we used the above code and sniffed packet that shows information about the packets. In below image there we can see 2 icmp packets which shows ethernet layer, ip layer, icmp layer and raw layer.

in ethernet layer we can see the source and destination mac address. In ip layer we can see the see source and destination ip addresses then comes the icmp layer and raw layer.

Results from the packet sniffer after running with root privilege and pinging a VM from another VM.

```
/bin/bash 173x55
[07/26/21]seed@VM:~/Desktop$ sudo python mycod.py
###[ Ethernet ]###
  dst       = 08:00:27:59:41:98
  src       = 08:00:27:8a:13:0c
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 42496
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x7c9e
     src       = 10.0.2.6
     dst       = 10.0.2.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x825f
        id        = 0xafc
        seq       = 0x1
###[ Raw ]###
           load      = '\xba\x9c\xfe`\xbf\xa2\x07\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 08:00:27:8a:13:0c
  src       = 08:00:27:59:41:98
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 49728
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xa05e
     src       = 10.0.2.5
     dst       = 10.0.2.6
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x8a5f
        id        = 0xafc
        seq       = 0x1
###[ Raw ]###
           load      = '\xba\x9c\xfe`\xbf\xa2\x07\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Running the program without root privileges:

```
/bin/bash 173x55
[07/26/21]seed@VM:~/Desktop$ python sniff.py
Traceback (most recent call last):
  File "sniff.py", line 6, 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.py", line 567, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))
  File "/usr/lib/python2.7/socket.py", line 191, in __init__
    _sock = _realsocket(family, type, proto)
socket.error: [Errno 1] Operation not permitted
[07/26/21]seed@VM:~/Desktop$
```

**Observation**: As we can see from the above screenshots we demonstrate how to use scapy to sniff network traffic. If we try to run without using the root privileges the program will not run.

**Explanation:** Scapy is one of the tools makes it easy to sniff network traffic. It requires root privileges to run since you need to put the NIC into promiscuous mode.

**Task 1.1b**

Code for capturing only ICMP traffic:

```
Open
#!/usr/bin/python3
# -*- coding: utf-8 -*-
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='icmp',prn=print_pkt)
```

Output of icmp traffic code used to capture packet from a VM sending icmp packet to other VM.

```
/bin/bash 173x55
[07/26/21]seed@VM:~/Desktop$ sudo python mycod.py
###[ Ethernet ]###
  dst       = 08:00:27:59:41:98
  src       = 08:00:27:8a:13:0c
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 42496
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x7c9e
     src       = 10.0.2.6
     dst       = 10.0.2.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x825f
        id        = 0xafc
        seq       = 0x1
###[ Raw ]###
           load      = '\xba\x9c\xfe`\xbf\xa2\x07\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 08:00:27:8a:13:0c
  src       = 08:00:27:59:41:98
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 49728
     flags     = 
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xa05e
     src       = 10.0.2.5
     dst       = 10.0.2.6
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x8a5f
        id        = 0xafc
        seq       = 0x1
###[ Raw ]###
           load      = '\xba\x9c\xfe`\xbf\xa2\x07\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Code for capturing only TCP traffic of particular host at destination port 23:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='tcp and (src host 10.0.2.6 and dst port 23)', prn=print_pkt)
```

Output from sniffing only TCP packet from particular host at destination port 23 that is telnet:

```
[07/26/21]seed@VM:~/CODE$ sudo python sniffer2\ tcp.py
###[ Ethernet ]###
  dst       = 08:00:27:59:41:98
  src       = 08:00:27:8a:13:0c
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 40
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x62c5
     src       = 10.0.2.6
     dst       = 10.0.2.5
     \options   \
###[ TCP ]###
        sport     = ftp_data
        dport     = telnet
        seq       = 0
        ack       = 0
        dataofs   = 5
        reserved  = 0
        flags     = S
        window    = 8192
        chksum    = 0x77ad
        urgptr    = 0
        options   = []
###[ Padding ]###
           load      = '\x00\x00\x00\x00\x00\x00'
```

Code for capturing packets only in subnet 192.168.0.0/16:

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

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='src net 192.168.5.0/8', prn=print_pkt)
```

Capture of packets only to/from 192.168.8.0/8 while pinging 192.168.5.2:

```
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:3a:a0:7e
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 49671
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xa6f3
     src       = 10.0.2.4
     dst       = 192.168.5.2
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x342a
        id        = 0x3352
        seq       = 0x7
###[ Raw ]###
           load      = '\xfc\xba\xfe`\xa4]\x06\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x
```

```
/bin/bash 110x4
[07/26/21]seed@VM:~/CODE$ ping 192.168.5.2
PING 192.168.5.2 (192.168.5.2) 56(84) bytes of data.
```

**Observation**: The screenshots above show some of the filters that can be used to capture only specific packets. We first only process ICMP packts, then TCP with a destination port of 23 and then packets only to/from subnet 192.168.8.0/8.

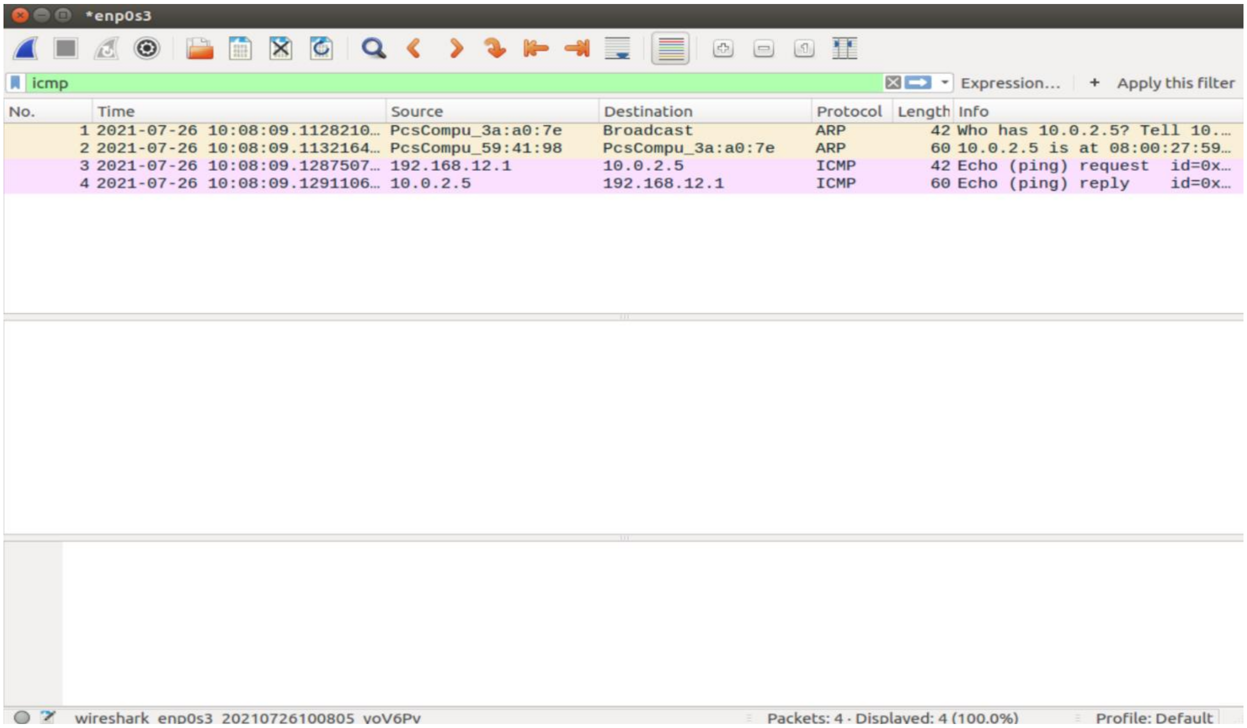
**Explanation:** Scapy allows for various traffic filters to capture only the packets that we are concerned with.

### Task 1.2 Spoofing ICMP Packets

Code for spoofing ICMP packet:

```
/bin/bash 80x24
[07/26/21]seed@VM:~$ sudo python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> a=IP()
>>> b=ICMP()
>>> a.src='192.168.12.1'
>>> a.dst='10.0.2.5'
>>> p=a/b
>>> send(p)
.
Sent 1 packets.
>>>
```

Wireshark output from spoofed packet:




**Observation**: Above we use Scapy to spoof ICMP packets, making it look like the packet came from IP 192.168.12.1 when that was not the case.

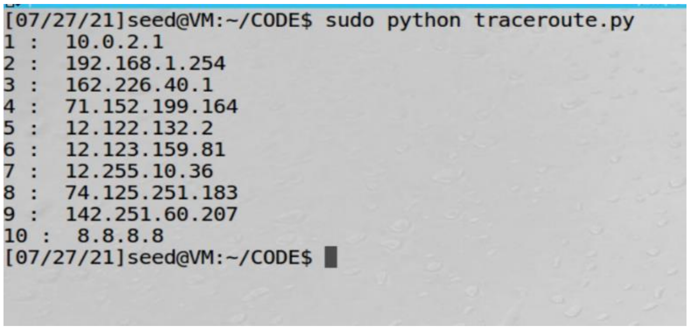
**Explanation:** Scapy make operations like spoofing an ICMP packet trivial. Packets can be spoofed and made to look like a different origin than it actually came from.

**Task 1.3 Traceroute**

Code for traceroute to 8.8.8.8:

```
#!/usr/bin/python
from scapy.all import *
i=1
while i <=15:
    a=IP()
    a.dst='8.8.8.8'
    a.ttl=i
    b=ICMP()
    pkt=a/b
    replypkt = sr1(pkt, verbose=0)
    if replypkt[IP].src == a.dst:
        print "%d : " %i, replypkt[IP].src
        break
    elif replypkt[ICMP].type == 0:
        print "%d : " %i, replypkt[IP].src

    else :
        print "%d : " %i, replypkt[IP].src

    i += 1
```

Output from executing traceroute program to 8.8.8.8:



```
[07/27/21]seed@VM:~/CODE$ sudo python traceroute.py
1 :  10.0.2.1
2 :  192.168.1.254
3 :  162.226.40.1
4 :  71.152.199.164
5 :  12.122.132.2
6 :  12.123.159.81
7 :  12.255.10.36
8 :  74.125.251.183
9 :  142.251.60.207
10 :  8.8.8.8
[07/27/21]seed@VM:~/CODE$
```

**Observation**: In the screenshot above we can see the route packet used to travel from host to the Google DNS server (8.8.8.8)

**Explanation:** Scapy can also be used for traceroute like functionality to show the route that was taken between the host and the server.

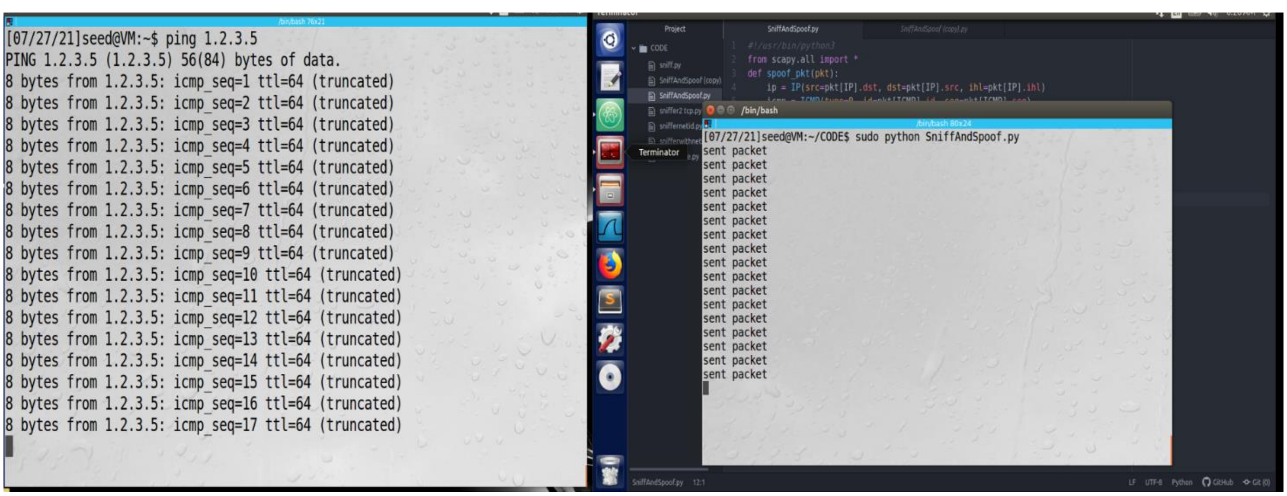
### Task 1.4 Sniffing and-then Snooping

Code for Sniffing and Spoofing program:

```
SniffAndSpoof.py        SniffAndSpoof (copy).py

#!/usr/bin/python3
from scapy.all import *
def spoof_pkt(pkt):
    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)

    newpkt=ip/icmp
    send(newpkt, verbose=0)
    print("sent packet")

pkt = sniff(filter='icmp[icmptype] = 8', prn=spoof_pkt)
```

VM A requesting ping from 1.2.3.5, VM B running sniff and spoof program

VM A is receiving the spoofed reply from VM B:



```
[07/27/21]seed@VM:~$ ping 1.2.3.5
PING 1.2.3.5 (1.2.3.5) 56(84) bytes of data.
8 bytes from 1.2.3.5: icmp_seq=1 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=2 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=3 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=4 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=5 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=6 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=7 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=8 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=9 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=10 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=11 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=12 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=13 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=14 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=15 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=16 ttl=64 (truncated)
8 bytes from 1.2.3.5: icmp_seq=17 ttl=64 (truncated)
```

```
[07/27/21]seed@VM:~/CODE$ sudo python SniffAndSpoof.py
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
sent packet
```

**Observation**: in above screenshot we can see how we sniffed the icmp request packet and spoofed icmp reply packet

**Explanation:** VM A I used ping command to ip 1.2.3.5 and on VM B which is my attacking machine, I run a python code for sniffing icmp echo request packets and spoofing icmp echo reply. So VM A thinks that there is a machine with ip 1.2.3.5 but no such machine exists.

## 2. Writing Programs to Sniff and Spoof Packets

## 2.1 Writing a Packet Sniffing Program

## 2.1A Understanding How a Sniffer Works

This task demonstrations using the PCAP library to sniff packets.

Code for Sniffer:

sniffer.c    sniff.c

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

/* Ethernet header */
struct ethheader {
  u_char  ether_dhost[6]; /* destination host address */
  u_char  ether_shost[6]; /* source host address */
  u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
};

/* 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
};

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));

```

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

    /* 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");
            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);   //Close the handle
    return 0;
}
```

Output for running sniffing program, and pinging google/opening Firefox browser:

```
/bin/bash
/bin/bash 80x24
[07/28/21]seed@VM:~/CODE$ gcc -o sniffer sniffer.c -lpcap
[07/28/21]seed@VM:~/CODE$ sudo ./sniffer
       From: 10.0.2.5
         To: 8.8.8.8
   Protocol: ICMP
       From: 8.8.8.8
         To: 10.0.2.5
   Protocol: ICMP
       From: 10.0.2.5
         To: 8.8.8.8
   Protocol: ICMP
       From: 8.8.8.8
         To: 10.0.2.5
   Protocol: ICMP
       From: 10.0.2.4
         To: 224.0.0.251
   Protocol: UDP
       From: 10.0.2.4
         To: 224.0.0.251
   Protocol: UDP
       From: 10.0.2.5
         To: 10.0.2.3
   Protocol: UDP
       From: 10.0.2.3
```

**Observation**: The image capture above shows the traffic being sniffed using the PCAP library. The program was set to capture all packets and, in the screenshot above, you can see packets being captured.

**Explanation**:  The PCAP library can be used to sniff network traffic. There is a filter that can be used if there is only a certain type of packet is desired.

• **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: we need to turn on the promiscuous mode so can capture all the traffic and then use pcap library to capture the packet and show on the screen.

• **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: we need the root privilege to run a sniffer program because we need to turn on the promiscuous mode which is only done by root privilege.

• **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: when promiscuous mode is turned off, machine will only accept the packets that are coming in or out of the machine itself. And when promiscuous mode is turned on it will accept every packet that Is flooded on the network, even the packet is for the machine itself or any other, it accept every packet.

**2.1B Writing Filters**

Various filters can be used to capture only specific types of traffic, below are examples some examples of those filters :

Code for filter for capturing only ICMP packets between two specific hosts (10.0.2.5 and 8.8.8.8):

```
char filter_exp[] = "icmp and src host 10.0.2.5 and dst host 8.8.8.8";
```

```
/bin/bash
[07/28/21]seed@VM:~$ cd CODE/
[07/28/21]seed@VM:~/CODE$ ls
sniff  sniffer  sniffer.c  using scapy
[07/28/21]seed@VM:~/CODE$ gcc -o sniffer sniffer.c -lpcap
[07/28/21]seed@VM:~/CODE$ sudo ./sniffer
       From: 10.0.2.5
         To: 8.8.8.8
   Protocol: ICMP
       From: 10.0.2.5
         To: 8.8.8.8
   Protocol: ICMP
```

The above screenshot shows the icmp packet captured between two specific hosts (10.0.2.5 and 8.8.8.8).

Filter code for capturing only TCP packets in port range 10-100:

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

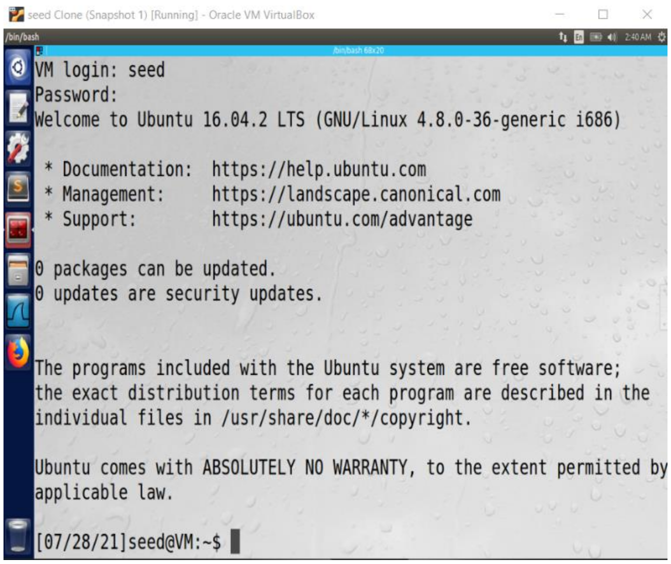
**2.1C Sniffing Passwords**

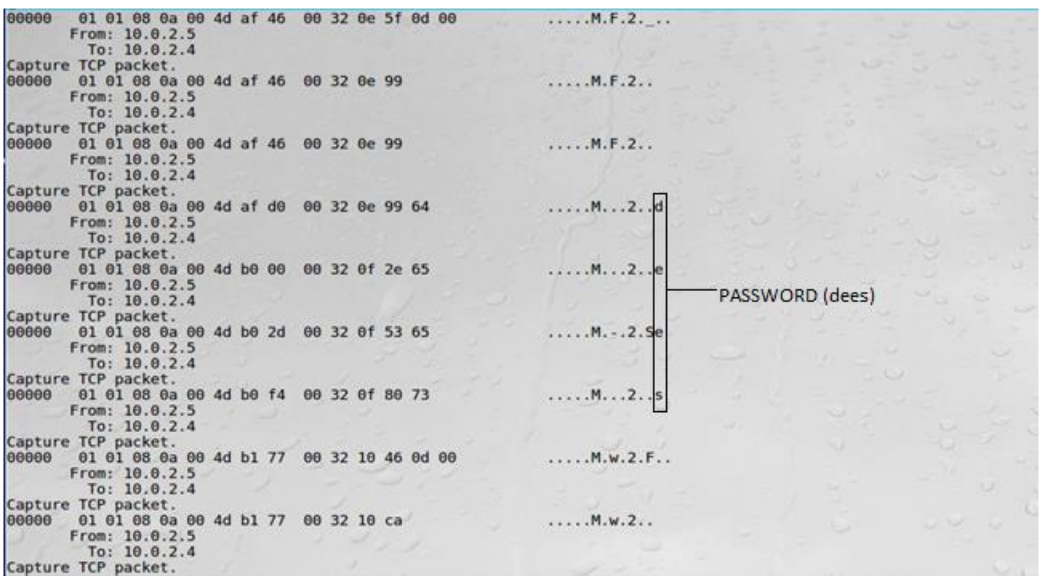
Packet sniffing can be used to exploit passwords in certain cases.  Below is an example of sniffing telnet traffic to find out the user’s password:

Code for sniffing passwords:

```
sniffer.c        sniffpwd.c        sniff.c
            print_hex_ascii_line(ch, len_rem, offset);
            break;
        }
    }
}

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


        if (ip->iph_protocol == IPPROTO_TCP)
        {
            printf("Capture TCP packet.\n");
            char *payload = (u_char *)(packet + sizeof(struct ethheader) + sizeof(struct ipheader) + sizeof(struct sniff_tcp));
            int payload_len = ntohs(ip->iph_len) - sizeof(struct ipheader) - sizeof(struct sniff_tcp);
            print_payload(payload, payload_len);
        }
    }
}

int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "tcp port 23 and src host 10.0.2.5";
    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);
```

Example of Sniffed Password ('dees'):




**Observation**: In the screenshots above you can see the telnet traffic being sniffed and the password 'dees' being captured. we used out program to print the data out to the screen. The password was sent from 10.0.2.5 to 10.0.2.6 and intercepted using the packet sniffing.

**Explanation**: Using the PCAP library and filtering for telnet traffic we were able to capture the packets. We then processed them and printed the info to the screen and discovered that the user's password was 'dees' marked in the above screenshot with a box. This task showed a practical use of a sniffing program.

### 2.2 Spoofing

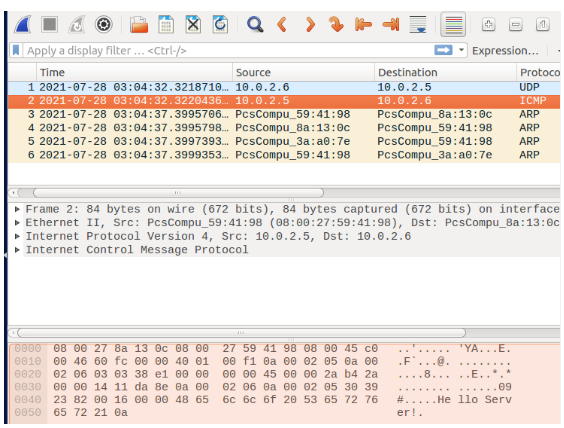
### 2.2A Write a Spoofing Program

This task shows the ability to send out an ICMP packet from spoofed IP of 1.2.3.5 to IP 10.0.2.6 by using raw sockets:

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

struct udpheader {…};
struct ipheader {…};

void send_raw_ip_packet (struct ipheader *ip) {
  int sd;
  int enable = 1;
  struct sockaddr_in sin;
  /* 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);
  }
  setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
  /* This data structure is needed when sending the packets using sockets. Normally, we need to fill out several
  * fields, but for raw sockets, we only need to fill out this one field */
  sin.sin_family = AF_INET;
  sin.sin_addr = ip->iph_destip;
  /* Send out the IP packet. ip_len is the actual size of the packet. */
  if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin,sizeof(sin)) < 0) {
    perror("sendto() error"); exit(-1);
  }
}

int main() {
  char buffer[1500];
  memset(buffer, 0, 1500);
  struct ipheader *ip = (struct ipheader *) buffer;
  struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
  // Filling in UDP Data field
  char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
  const char *msg="Hello Server!\n";
  int data_len = strlen(msg);
  strncpy(data, msg, data_len);
```

```c
  strncpy(data, msg, data_len);
  // Fill in the UDP header
  udp->udp_sport = htons(12345);
  udp->udp_dport = htons(9090);
  udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
  udp->udp_sum = 0;
  // Fill in the IP header
  ip->iph_ver = 4;
  ip->iph_ihl = 5;
  ip->iph_ttl = 20;
  ip->iph_sourceip.s_addr = inet_addr("10.0.2.5");
  ip->iph_destip.s_addr = inet_addr("10.0.2.6");
  ip->iph_protocol = IPPROTO_UDP;
  ip->iph_len=htons(sizeof(struct ipheader)+sizeof(struct udpheader) + data_len);
  // Send the spoofed packet
  send_raw_ip_packet(ip);
  return 0;
}
```

On machine 10.0.2.6 you can see the spoofed packet that it received:

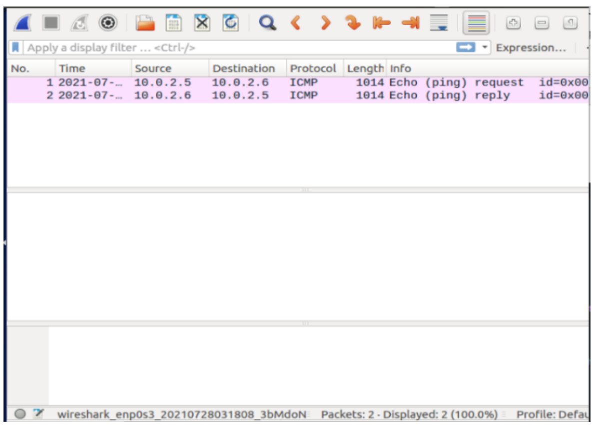



**Observation**: The screenshot shows that a packet was received from source IP 10.2.0.5. This was not the actual origination of the packet; it instead came from machine 10.0.2.4 who was sending the spoofed packet.

**Explanation:** By using a raw socket, we were able to make a packet and send it to a machine but with some other ip address and faked the existence of machine.

### 2.2B Spoof an ICMP Echo Request

```
  sum = (sum >> 16) + (sum & 0xffff);
  sum += (sum>>16);
  return (unsigned short)(~sum);
}
int main() {
  char buffer[1500];
  memset(buffer, 0, 1500);
  struct ipheader *ip = (struct ipheader *) buffer;
  struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct ipheader));
  // Fill in the ICMP header
  icmp->icmp_type=8;
  icmp->icmp_chksum=0;
  icmp->icmp_chksum = in_chksum((unsigned short *)icmp, sizeof(struct ipheader));

  // Fill in the IP header
  ip->iph_ver = 4;
  ip->iph_ihl = 5;
  ip->iph_ttl = 20;
  ip->iph_sourceip.s_addr = inet_addr("10.0.2.5");
  ip->iph_destip.s_addr = inet_addr("10.0.2.6");
  ip->iph_protocol = IPPROTO_ICMP;
  ip -> iph_len = htons(1000);
  // ip->iph_len=htons(sizeof(struct ipheader)+sizeof(struct icmpheader));
  // Send the spoofed packet
  send_raw_ip_packet(ip);
  return 0;
}
```




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

Answer: If we set the length to some random value, the packet will not be formed properly. If the packet is too big when it is sent, it will be not be sent.

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

Answer: no we don't need to calculate the checksum of ip header. Because OS do that before transmitting 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?

Answer:. We can use raw sockets to spoof a packet and set arbitrary values to any field in the packet headers. So in order to perform these tasks, it requires root privileges. When the spoofing program is run as a non-root user, it gives an error that the program needs to access the Network Interface Card in order to send the packet.

### 2.3 Sniff and then Spoof

Code for sniffing an ICMP request and spoofing an ICMP reply:

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

struct ethheader {
  u_char ether_dhost[6];
  u_char ether_shost[6];
  u_short ether_type;
};
struct icmpheader {
  unsigned char icmp_type;
  unsigned char icmp_code;
  unsigned short int icmp_chksum;
  unsigned short int icmp_id;
  unsigned short int icmp_seq;
};
struct ipheader {
  unsigned char iph_ihl:4, iph_ver:4;
  unsigned char iph_tos;
  unsigned short int iph_len;
  unsigned short int iph_ident;
  unsigned short int iph_flag:3, iph_offset:13;
  unsigned char iph_ttl;
  unsigned char iph_protocol;
  unsigned short int iph_chksum;
  struct in_addr iph_sourceip;
  struct in_addr iph_destip;
};
void send_raw_ip_packet (struct ipheader *ip) {
  int sd;
  int enable = 1;
  struct sockaddr_in sin;
```

```c
  sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
  if(sd < 0) {
    perror("socket() error"); exit(-1);
  }
  // Set socket options
  setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
  /* This data structure is needed when sending the packets using sockets. Normally, we need to fill out several
  * fields, but for raw sockets, we only need to fill out this one field */
  sin.sin_family = AF_INET;
  sin.sin_addr = ip->iph_destip;
  /* Send out the IP packet. ip_len is the actual size of the packet. */
  if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin,sizeof(sin)) < 0) {
    perror("sendto() error"); exit(-1);
  }
  else {
    printf(" Packet Sent from Attacker to host:%s\n",inet_ntoa(ip->iph_destip) );
  }
}

unsigned short in_chksum(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 & 0xfff);
  sum += (sum>>16);
  return (unsigned short)(~sum);
}
```

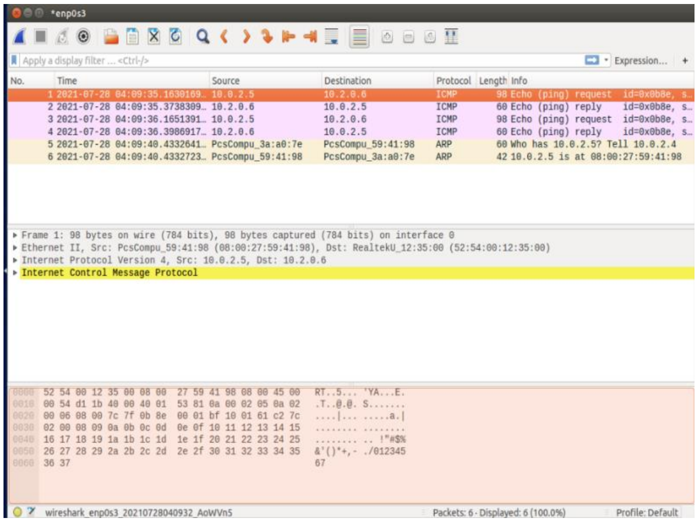
```c

void spoof_reply(struct ipheader *ip) {
  const char buffer[1500];
  int ip_header_len = ip->iph_ihl * 4;
  struct icmpheader *icmp = (struct icmpheader *) ((u_char *)ip + ip_header_len);
  if(icmp->icmp_type != 8) return;

  memset((char *)buffer, 0, 1500);
  memcpy((char *)buffer, ip, ntohs(ip->iph_len));
  struct ipheader *newip = (struct ipheader *) buffer;
  struct icmpheader *newicmp = (struct icmpheader *) (buffer + ip_header_len);
  // Fill in the ICMP header
  newicmp->icmp_type=0;
  newicmp->icmp_chksum=0;
  newicmp->icmp_chksum = in_chksum((unsigned short *)icmp, ip_header_len);

  // Fill in the IP header
  newip->iph_ttl = 50;
  newip->iph_sourceip = ip->iph_destip;
  newip->iph_destip = ip->iph_sourceip;
  newip->iph_protocol = IPPROTO_ICMP;
  newip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
  // Send the spoofed packet
  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){
    struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
    int ip_header_len = ip->iph_ihl * 4;
    if (ip->iph_protocol == IPPROTO_ICMP) {
      spoof_reply(ip);
    }
  }
}

int main(){
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "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); //Close the handle
  return 0;
}
```

VM A running sniffing/spoofing:

```
/bin/bash 88x28
[07/28/21]seed@VM:~/CODE$ sudo ./sniffandspoof
 Packet Sent from Attacker to host:10.0.2.5
 Packet Sent from Attacker to host:10.0.2.5
```

VM B after pinging 10.2.0.6:

```
[07/28/21]seed@VM:~$ ping 10.2.0.6
PING 10.2.0.6 (10.2.0.6) 56(84) bytes of data.
8 bytes from 10.2.0.6: icmp_seq=1 ttl=50 (truncated)
8 bytes from 10.2.0.6: icmp_seq=2 ttl=50 (truncated)
^Z
[3]+  Stopped                 ping 10.2.0.6
[07/28/21]seed@VM:~$
```




**Observation:** We first run the sniffing and spoofing program on VM A and then from VM B we are going to ping an IP address that is not alive 10.0.2.6. If the sniff and spoof program wasn't running on VM A then VM B's ping to 10.0.2.6 would have timed out, but since the program was running, VM B did get a reply. We used Wireshark to show the spoofed packet above.

**Explanation:** By using promiscuous mode, PCAP and raw sockets we were able to build a program that will hijack and respond to all ICMP requests. The packet was captured, and a raw socket was sent as a response to the ICMP request. This could be used for other types of traffic as well for various network-based attacks.