COMP 534 Computer & Network Security: Project 3 Report

Due on May 9th

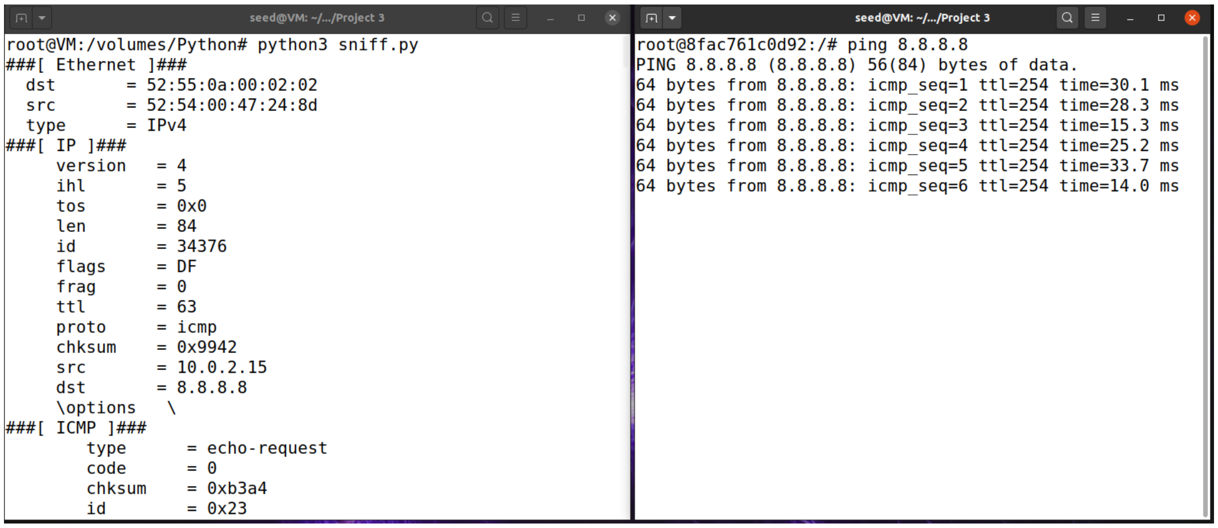
Ismayil Ismayilov

Lab Task Set 1: Using Scapy to Sniff and Spoof Packets

Task 1.1: Sniffing Packets

Task 1.1A

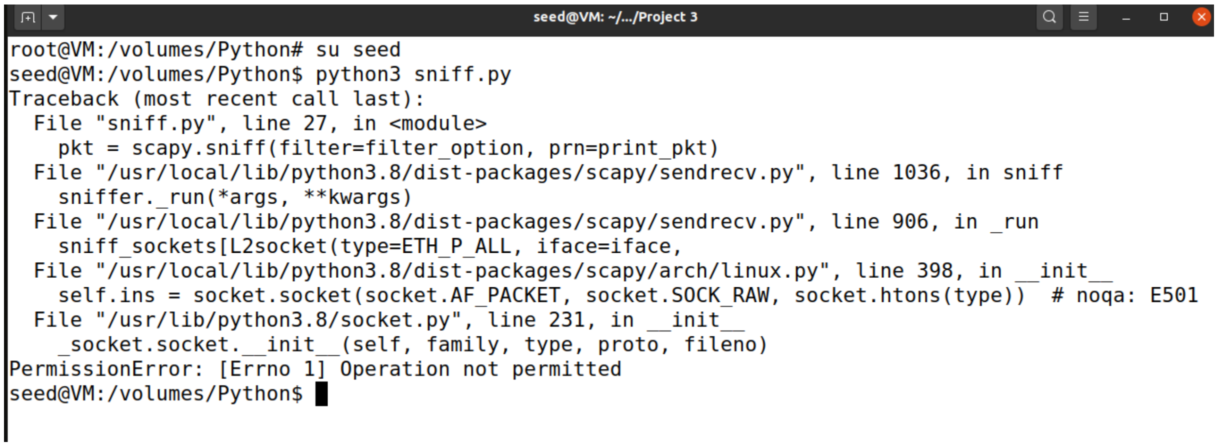
I run *sniff.py* on the attacker container. On HostA, I run *ping 8.8.8.8*. As the screenshot below, shows ICMP packets are sniffed on the attacker container.



```
root@VM:/volumes/Python# python3 sniff.py
###[ Ethernet ]###
  dst       = 52:55:0a:00:02:02
  src       = 52:54:00:47:24:8d
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 34376
     flags     = DF
     frag      = 0
     ttl       = 63
     proto     = icmp
     chksum    = 0x9942
     src       = 10.0.2.15
     dst       = 8.8.8.8
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xb3a4
        id        = 0x23
```

```
root@8fac761c0d92:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=254 time=30.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=254 time=28.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=254 time=15.3 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=254 time=25.2 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=254 time=33.7 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=254 time=14.0 ms
```

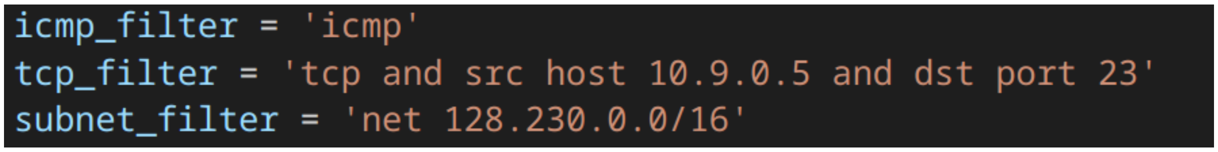
Next, I run *sniff.py* without root privilege. As the screenshot below shows, we get a permission error. This is because creating raw sockets and putting a device to promiscuous mode requires elevated privileges. This is further confirmed by the traceback output. clear



```
root@VM:/volumes/Python# su seed
seed@VM:/volumes/Python$ python3 sniff.py
Traceback (most recent call last):
  File "sniff.py", line 27, in <module>
    pkt = scapy.sniff(filter=filter_option, prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 906, in _run
    sniff_sockets[L2socket(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
seed@VM:/volumes/Python$
```

Task 1.1B

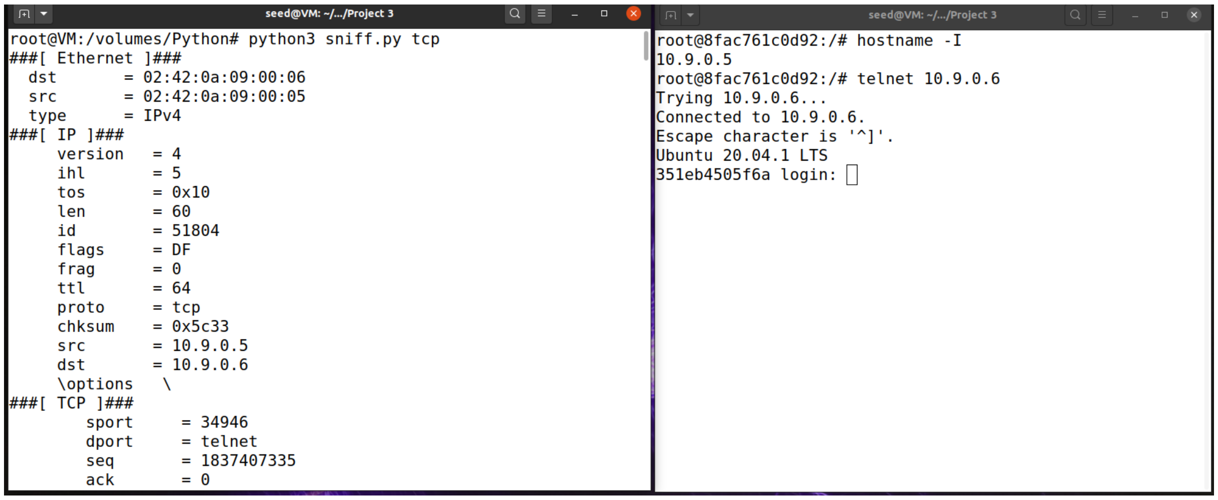
The filters that I used for this task are shown below



```
icmp_filter = 'icmp'
tcp_filter = 'tcp and src host 10.9.0.5 and dst port 23'
subnet_filter = 'net 128.230.0.0/16'
```

The *sniff.py* which I ran in Task 1.1A was already filtering only ICMP packets. Thus, I do not show the output here. The output is the same as in the first screenshot.

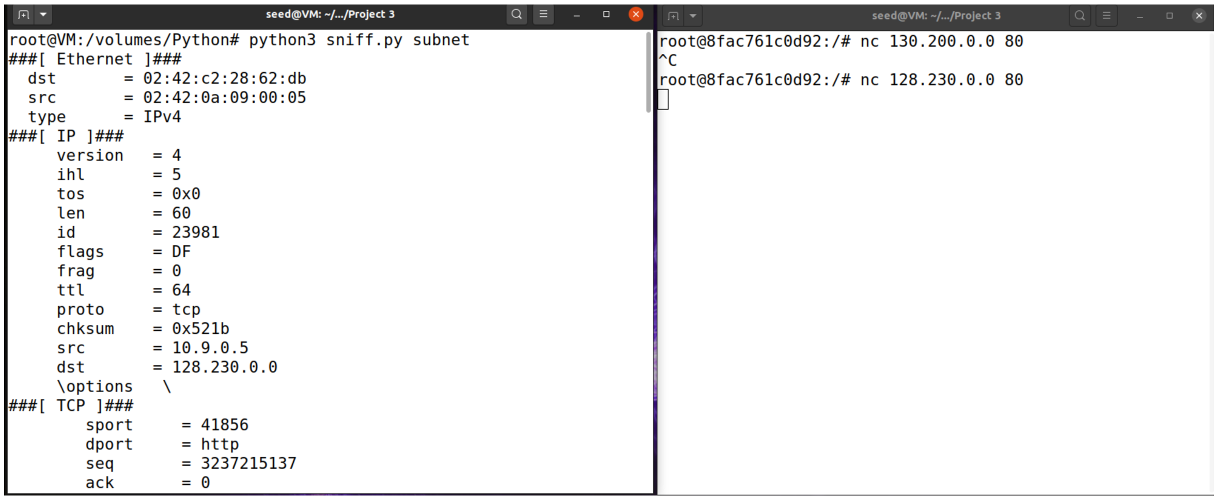
Note that to accomplish the TCP filtering task, I telnet from HostA (10.9.0.5) to HostB (10.9.0.6). Telnet is used because it corresponds to port 23.



```
root@VM:/volumes/Python# python3 sniff.py tcp
###[ 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        = 51804
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x5c33
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 34946
        dport     = telnet
        seq       = 1837407335
        ack       = 0
```

```
root@8fac761c0d92:/# hostname -I
10.9.0.5
root@8fac761c0d92:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
351eb4505f6a login: 
```

For the subnet filtering part, I filter on subnet *128.230.0.0/16*. I use *nc* to send the packets. I first *nc* to an adress that is not part of the filtered subnet *130.200.0.0*. I then *nc* to an address that is part of the filtered subnet *128.230.0.0*. As the screenshot shows, no packets are captured for *130.200.0.0* but packets are captured for *128.230.0.0*.

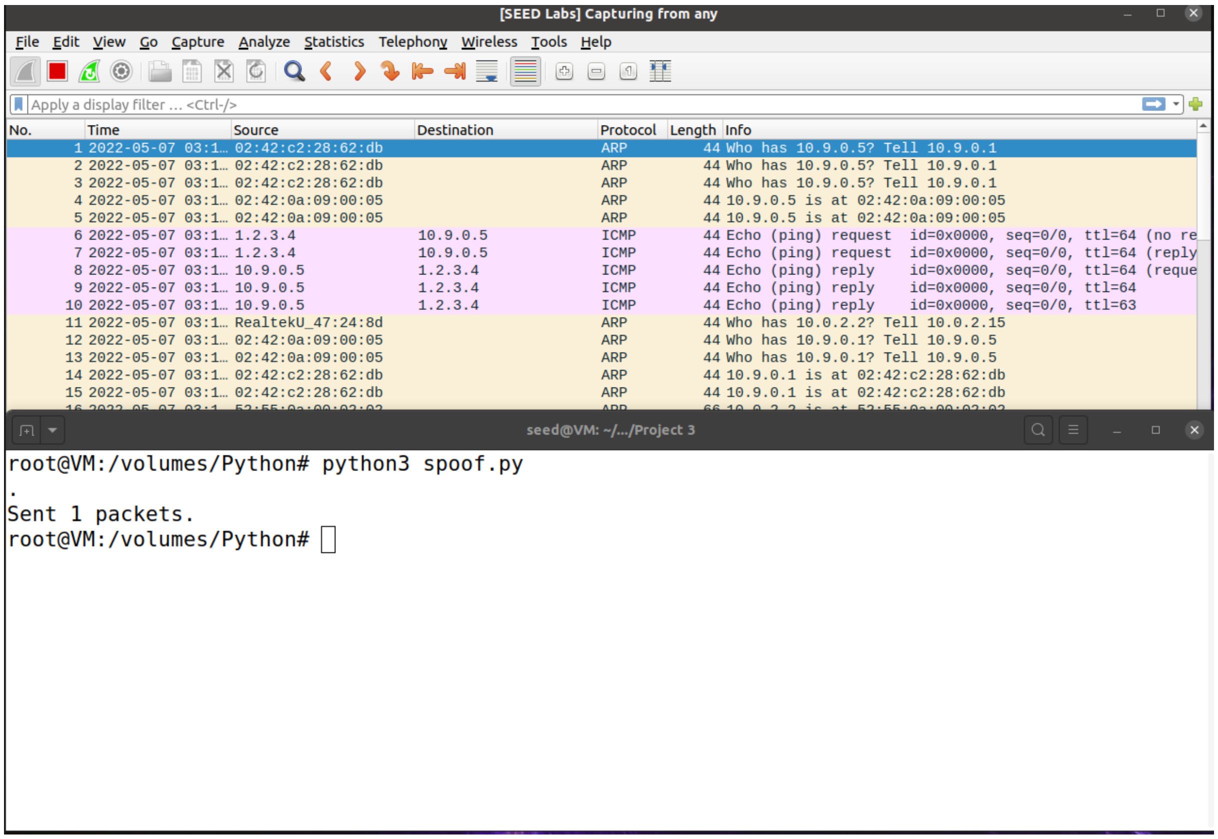


```
root@VM:/volumes/Python# python3 sniff.py subnet
###[ Ethernet ]###
  dst       = 02:42:c2:28:62:db
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 60
     id        = 23981
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x521b
     src       = 10.9.0.5
     dst       = 128.230.0.0
     \options   \
###[ TCP ]###
        sport     = 41856
        dport     = http
        seq       = 3237215137
        ack       = 0
```

```
root@8fac761c0d92:/# nc 130.200.0.0 80
^C
root@8fac761c0d92:/# nc 128.230.0.0 80

```

Task 1.2: Spoofing ICMP Packets

On the attacker container, I run *spoof.py*. In the IP object I set src to 1.2.3.4 (symbolizing an arbitrary IP address) and dst to 10.9.0.5 (IP address of HostA).


[SEED Labs] Capturing from any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
seed@VM: ~/…/Project 3

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-07 03:1…	02:42:c2:28:62:db		ARP	44	Who has 10.9.0.5? Tell 10.9.0.1
2	2022-05-07 03:1…	02:42:c2:28:62:db		ARP	44	Who has 10.9.0.5? Tell 10.9.0.1
3	2022-05-07 03:1…	02:42:c2:28:62:db		ARP	44	Who has 10.9.0.5? Tell 10.9.0.1
4	2022-05-07 03:1…	02:42:0a:09:00:05		ARP	44	10.9.0.5 is at 02:42:0a:09:00:05
5	2022-05-07 03:1…	02:42:0a:09:00:05		ARP	44	10.9.0.5 is at 02:42:0a:09:00:05
6	2022-05-07 03:1…	1.2.3.4	10.9.0.5	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no re
7	2022-05-07 03:1…	1.2.3.4	10.9.0.5	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply
8	2022-05-07 03:1…	10.9.0.5	1.2.3.4	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (reque
9	2022-05-07 03:1…	10.9.0.5	1.2.3.4	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=64
10	2022-05-07 03:1…	10.9.0.5	1.2.3.4	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=63
11	2022-05-07 03:1…	RealtekU_47:24:8d		ARP	44	Who has 10.0.2.2? Tell 10.0.2.15
12	2022-05-07 03:1…	02:42:0a:09:00:05		ARP	44	Who has 10.9.0.1? Tell 10.9.0.5
13	2022-05-07 03:1…	02:42:0a:09:00:05		ARP	44	Who has 10.9.0.1? Tell 10.9.0.5
14	2022-05-07 03:1…	02:42:c2:28:62:db		ARP	44	10.9.0.1 is at 02:42:c2:28:62:db
15	2022-05-07 03:1…	02:42:c2:28:62:db		ARP	44	10.9.0.1 is at 02:42:c2:28:62:db
16	2022-05-07 03:1…	52:55:0a:00:02:02		ARP	66	10.0.2.2 is at 52:55:0a:00:02:02

```
root@VM:/volumes/Python# python3 spoof.py
.
Sent 1 packets.
root@VM:/volumes/Python# 
```

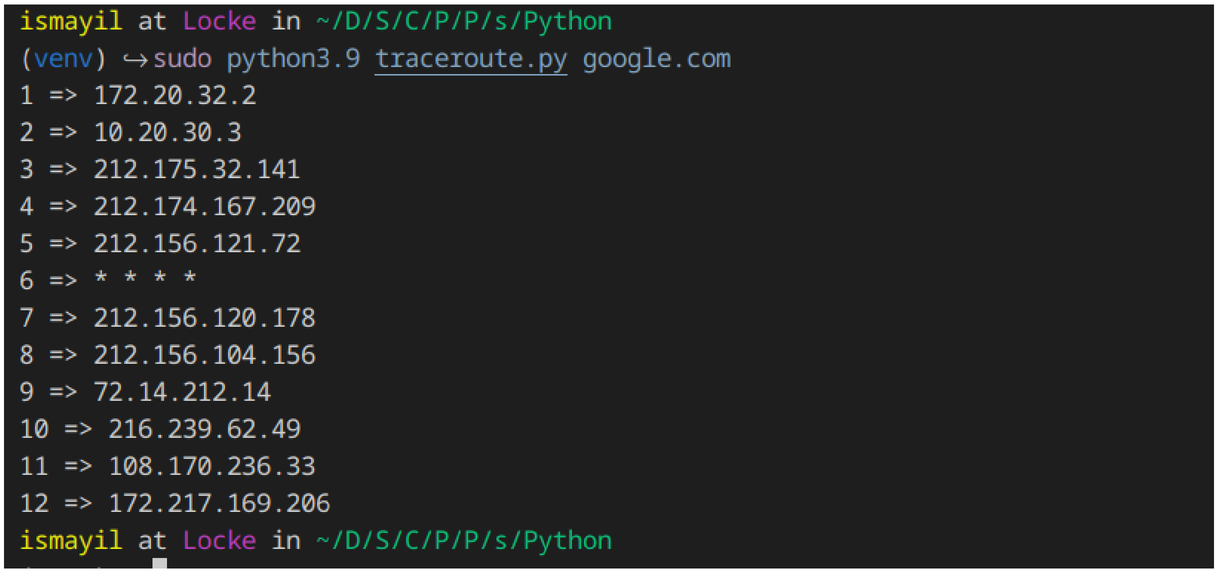
As the screenshot above shows a packet with source 1.2.3.4 is accepted and a reply is sent back. This indicates that our packet spoofing attempts have been successful.

Task 1.3: Traceroute

For this task I am using the *sr1* function to obtain a single reply packet. The program should terminate when the ICMP type of the reply is 0 (corresponding to an Echo Reply). If the reply type is 11 (corresponding to TTL exceeded), the program continues and increments the TTL.

Note that running *traceroute.py* on the VM was return a single hop which is obviously incorrect. I was not able to resolve this issue. Hence, I run *traceroute.py* locally.

The screenshot below shows the obtained output after running *traceroute google.com*

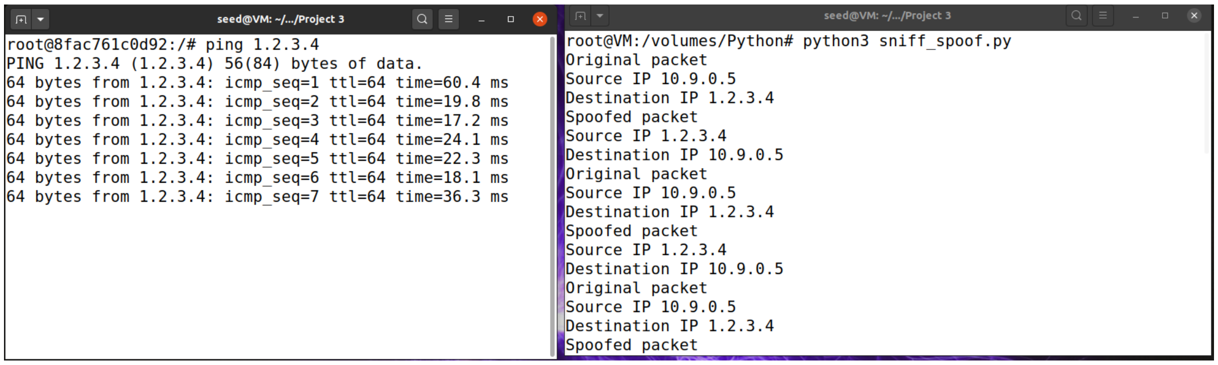


```
ismayil at Locke in ~/D/S/C/P/P/s/Python
(venv) ↪sudo python3.9 traceroute.py google.com
1 => 172.20.32.2
2 => 10.20.30.3
3 => 212.175.32.141
4 => 212.174.167.209
5 => 212.156.121.72
6 => * * * *
7 => 212.156.120.178
8 => 212.156.104.156
9 => 72.14.212.14
10 => 216.239.62.49
11 => 108.170.236.33
12 => 172.217.169.206
ismayil at Locke in ~/D/S/C/P/P/s/Python
```

Task 1.4: Sniffing and-then Spoofing

In all experiments, I run *sniff_spoof.py* on the attacker container and ping from HostA (10.9.0.5)

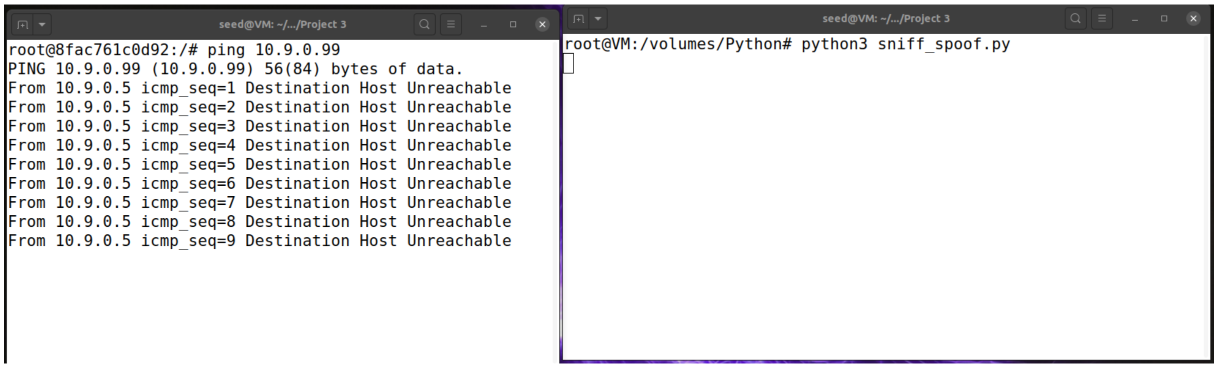
I first ping *1.2.3.4*. As the screenshot shows, HostA receives a reply even though *1.2.3.4* does not exist. Even though *1.2.3.4* is a non-existent host, there is still routing involved. Because we need to route the packet, it is sent to the next gateway / router which is why are we able to intercept as it hits the NIC. Note that this is different from the next example where we are pinging a non-existent host on the same subnet; when the non-existent host is on the same subnet we never receive an ARP reply so the pinging never occurs; when the non-existent host is not on the same subnet we still get an ARP reply since we need to route to the next gateway.



```
root@8fac761c0d92:/# ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=60.4 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=19.8 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=17.2 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=24.1 ms
64 bytes from 1.2.3.4: icmp_seq=5 ttl=64 time=22.3 ms
64 bytes from 1.2.3.4: icmp_seq=6 ttl=64 time=18.1 ms
64 bytes from 1.2.3.4: icmp_seq=7 ttl=64 time=36.3 ms
```

```
root@VM:/volumes/Python# python3 sniff_spoof.py
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
```

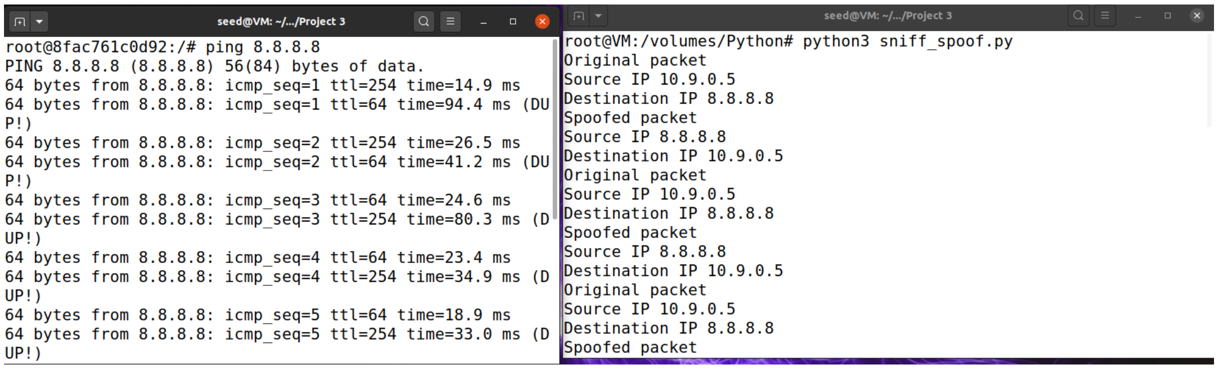
Next, I ping *10.9.0.99*. This time we receive no reply. We can also that no packets are sniffed or spoofed by the attacker even though *sniff_spoof.py* is running. The reason for this is because the hosts are on the same subnet; what happens is that prior to pinging, HostA sends an ARP request for *10.9.0.99*. Since *10.9.0.99* in on the same subnet and is a non-existent host, there is no ARP reply. Because there is no ARP reply, the ping is never performed. Because of this the ICMP packet nevers hits the NIC which is why the program cannot perform sniffing and/or spoofing.



```
root@8fac761c0d92:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.5 icmp_seq=1 Destination Host Unreachable
From 10.9.0.5 icmp_seq=2 Destination Host Unreachable
From 10.9.0.5 icmp_seq=3 Destination Host Unreachable
From 10.9.0.5 icmp_seq=4 Destination Host Unreachable
From 10.9.0.5 icmp_seq=5 Destination Host Unreachable
From 10.9.0.5 icmp_seq=6 Destination Host Unreachable
From 10.9.0.5 icmp_seq=7 Destination Host Unreachable
From 10.9.0.5 icmp_seq=8 Destination Host Unreachable
From 10.9.0.5 icmp_seq=9 Destination Host Unreachable
```

```
root@VM:/volumes/Python# python3 sniff_spoof.py
```

Finally, I ping *8.8.8.8*. We once again get a reply. The explanation for this is analogous to the explanation for the first example.



```
root@8fac761c0d92:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=254 time=14.9 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=94.4 ms (DU
P!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=254 time=26.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=41.2 ms (DU
P!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=24.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=254 time=80.3 ms (D
UP!)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=64 time=23.4 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=254 time=34.9 ms (D
UP!)
64 bytes from 8.8.8.8: icmp_seq=5 ttl=64 time=18.9 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=254 time=33.0 ms (D
UP!)
```

```
root@VM:/volumes/Python# python3 sniff_spoof.py
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
```

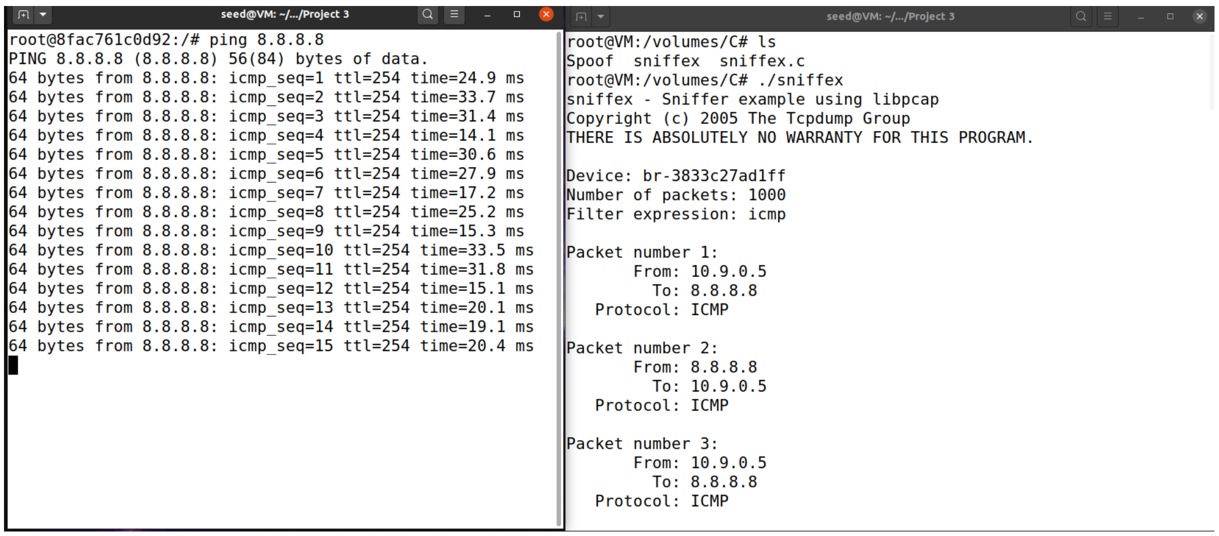
Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

Note that most of the code for this section was borrowed from the SEED Labs book. Additionally, the sniffer implementation (*sniffex.c*) was adapted from `http://www.tcpdump.org/pcap.html` with minor modifications.

Task 2.1: Writing Packet Sniffing Program

Task 2.1A: Understanding How a Sniffer Works

I compile and run *sniffex.c* on the attacker container and the ping *8.8.8.8* from HostA. The screenshot below shows packets being successfully being captured.



```
root@8fac761c0d92:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=254 time=24.9 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=254 time=33.7 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=254 time=31.4 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=254 time=14.1 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=254 time=30.6 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=254 time=27.9 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=254 time=17.2 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=254 time=25.2 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=254 time=15.3 ms
64 bytes from 8.8.8.8: icmp_seq=10 ttl=254 time=33.5 ms
64 bytes from 8.8.8.8: icmp_seq=11 ttl=254 time=31.8 ms
64 bytes from 8.8.8.8: icmp_seq=12 ttl=254 time=15.1 ms
64 bytes from 8.8.8.8: icmp_seq=13 ttl=254 time=20.1 ms
64 bytes from 8.8.8.8: icmp_seq=14 ttl=254 time=19.1 ms
64 bytes from 8.8.8.8: icmp_seq=15 ttl=254 time=20.4 ms
```

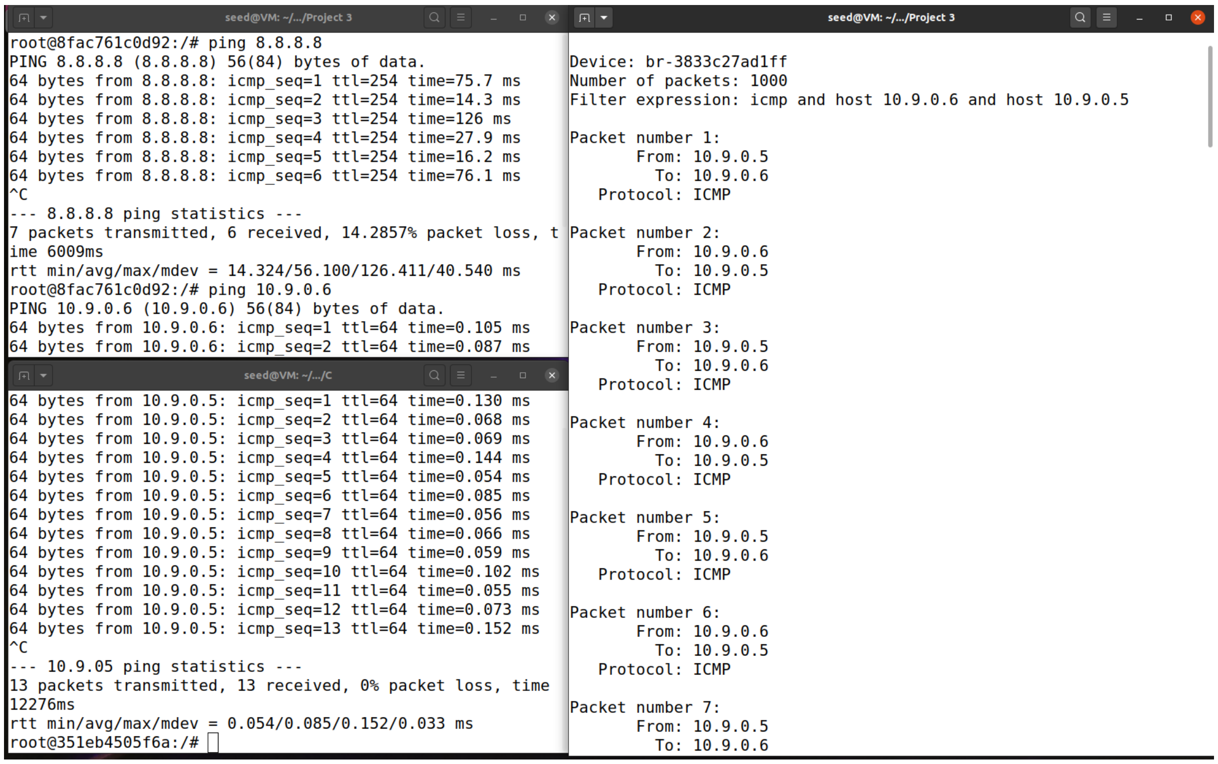
```
root@VM:/volumes/C# ls
Spoof  sniffex  sniffex.c
root@VM:/volumes/C# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: br-3833c27ad1ff
Number of packets: 1000
Filter expression: icmp

Packet number 1:
       From: 10.9.0.5
         To: 8.8.8.8
   Protocol: ICMP

Packet number 2:
       From: 8.8.8.8
         To: 10.9.0.5
   Protocol: ICMP

Packet number 3:
       From: 10.9.0.5
         To: 8.8.8.8
   Protocol: ICMP
```

- **Question 1**: First, the *pcap_open_live* function is used to create a raw socket, set the NIC to promiscuous mode and bind the socket. The *pcap_compile* and *pcap_setfilter* calls are used to compile and set the filter expression. The *pcap_loop* function eneters a loop and captures packets (a callback handler can be provided as well). Finally, the *pcap_close* function is used to close the handle to the NIC
- **Question 2**: Because elevated privileges are required to create raw sockest and to put a NIC into promiscuous mode. If the program is executed with root privileges, it will fail when calling the *pcap_open_live* function.
- **Question 3**: Going by the definition when promiscuous mode is turned off, all frames not intended for a given NIC will be dropped. In this situation, performing sniffing would obviously be limited if our goal is capturing packets not intended for the current NIC. The difference between promiscuous mode is on versus off can be demonstrated by first considering the name the name of the NIC and then turning promiscuous on and then off. When promiscuous mode is off, the number of sniffed packets will be limited as the output will only include packets intended for the current NIC. When promiscuous mode is on, however, the number of sniffed packets will be greatly increased. We can also differentiate between the two modes by observing that there are packets not intended for the current NIC when promiscuous mode is on.

Task 2.1B: Writing Filters

To capture ICMP packets between two specific hosts, I use the filter expression *[icmp and host 10.9.0.6 and host 10.9.0.5]*

In the screenshot below, I first ping *8.8.8.8* from HostA. The screenshot to the right shows that no packets are captured. When I ping HostB (10.9.0.6) from HostA (10.9.0.5), however, packets start being captured. Analogously, pinging HostA (10.9.0.5) from HostB (10.9.0.6) also results in packets being captured

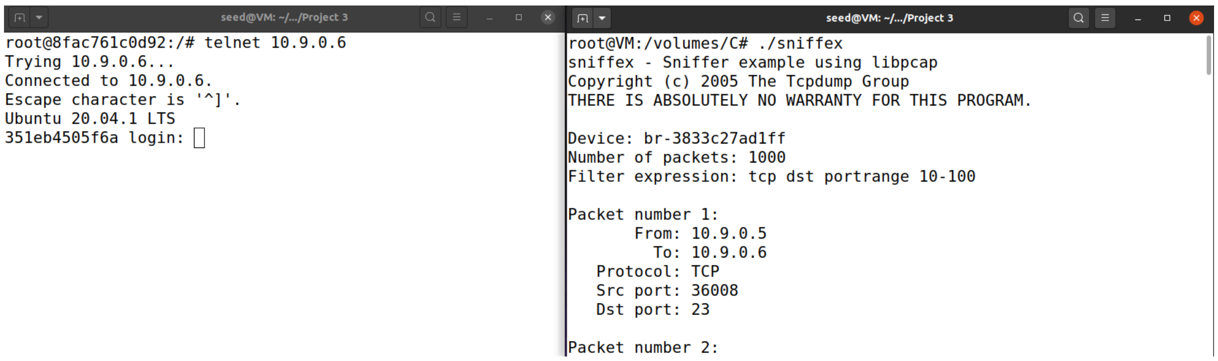


```
root@8fac761c0d92:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=254 time=75.7 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=254 time=14.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=254 time=126 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=254 time=27.9 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=254 time=16.2 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=254 time=76.1 ms
^C
--- 8.8.8.8 ping statistics ---
7 packets transmitted, 6 received, 14.2857% packet loss, t
ime 6009ms
rtt min/avg/max/mdev = 14.324/56.100/126.411/40.540 ms
root@8fac761c0d92:/# 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=0.105 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.087 ms
```

```
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.130 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.068 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.069 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.144 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=64 time=0.054 ms
64 bytes from 10.9.0.5: icmp_seq=6 ttl=64 time=0.085 ms
64 bytes from 10.9.0.5: icmp_seq=7 ttl=64 time=0.056 ms
64 bytes from 10.9.0.5: icmp_seq=8 ttl=64 time=0.066 ms
64 bytes from 10.9.0.5: icmp_seq=9 ttl=64 time=0.059 ms
64 bytes from 10.9.0.5: icmp_seq=10 ttl=64 time=0.102 ms
64 bytes from 10.9.0.5: icmp_seq=11 ttl=64 time=0.055 ms
64 bytes from 10.9.0.5: icmp_seq=12 ttl=64 time=0.073 ms
64 bytes from 10.9.0.5: icmp_seq=13 ttl=64 time=0.152 ms
^C
--- 10.9.05 ping statistics ---
13 packets transmitted, 13 received, 0% packet loss, time
12276ms
rtt min/avg/max/mdev = 0.054/0.085/0.152/0.033 ms
root@351eb4505f6a:/#
```

```
Device: br-3833c27ad1ff
Number of packets: 1000
Filter expression: icmp and host 10.9.0.6 and host 10.9.0.5

Packet number 1:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: ICMP

Packet number 2:
       From: 10.9.0.6
         To: 10.9.0.5
   Protocol: ICMP

Packet number 3:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: ICMP

Packet number 4:
       From: 10.9.0.6
         To: 10.9.0.5
   Protocol: ICMP

Packet number 5:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: ICMP

Packet number 6:
       From: 10.9.0.6
         To: 10.9.0.5
   Protocol: ICMP

Packet number 7:
       From: 10.9.0.5
         To: 10.9.0.6
```

To capture TCP packets on ports between 10 and 100, I use the filter expression *[tcp dst portrange 10-100]*. To test that the program is working correct, I telnet from HostA to HostB (Telnet uses TCP and is on port 23). The screenshot below shows that packets are successfully captured.

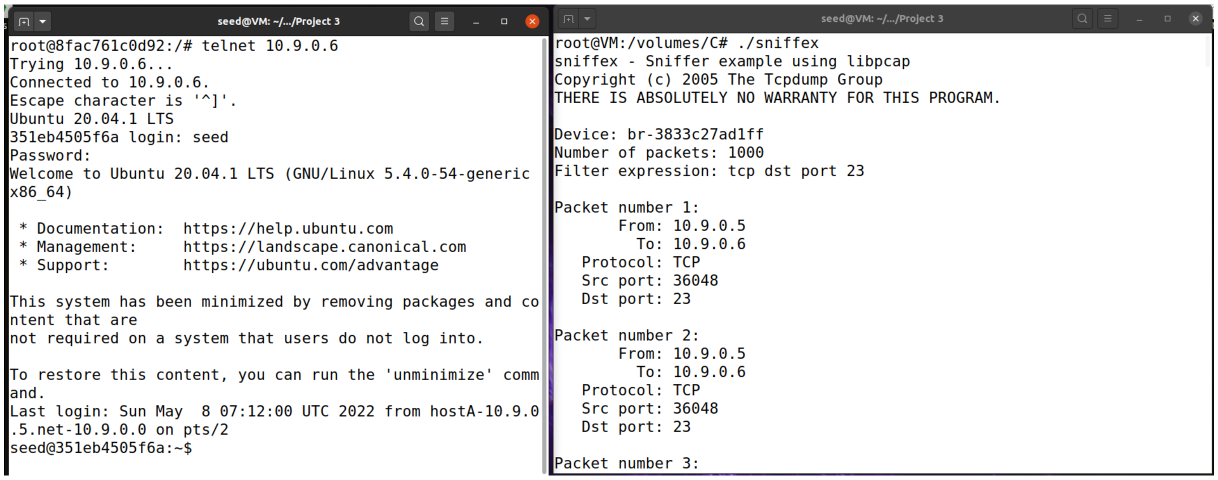


```
root@8fac761c0d92:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
351eb4505f6a login:
```

```
root@VM:/volumes/C# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: br-3833c27ad1ff
Number of packets: 1000
Filter expression: tcp dst portrange 10-100

Packet number 1:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 36008
   Dst port: 23

Packet number 2:
```

Task 2.1C: Sniffing Passwords

Note that the default username and password for SEED Lab VMs are *seed* and *dees* respectively. To test the correctness of the program, I telnet from HostA (10.9.0.5) to HostB (10.9.0.6). I use the default username and password mentioned above. As shown below, the TCP payload sniffer is successfully printing out the payloads of captured packets.



```
root@8fac761c0d92:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
351eb4505f6a 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 co
ntent that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' comm
and.
Last login: Sun May  8 07:12:00 UTC 2022 from hostA-10.9.0
.5.net-10.9.0.0 on pts/2
seed@351eb4505f6a:~$
```

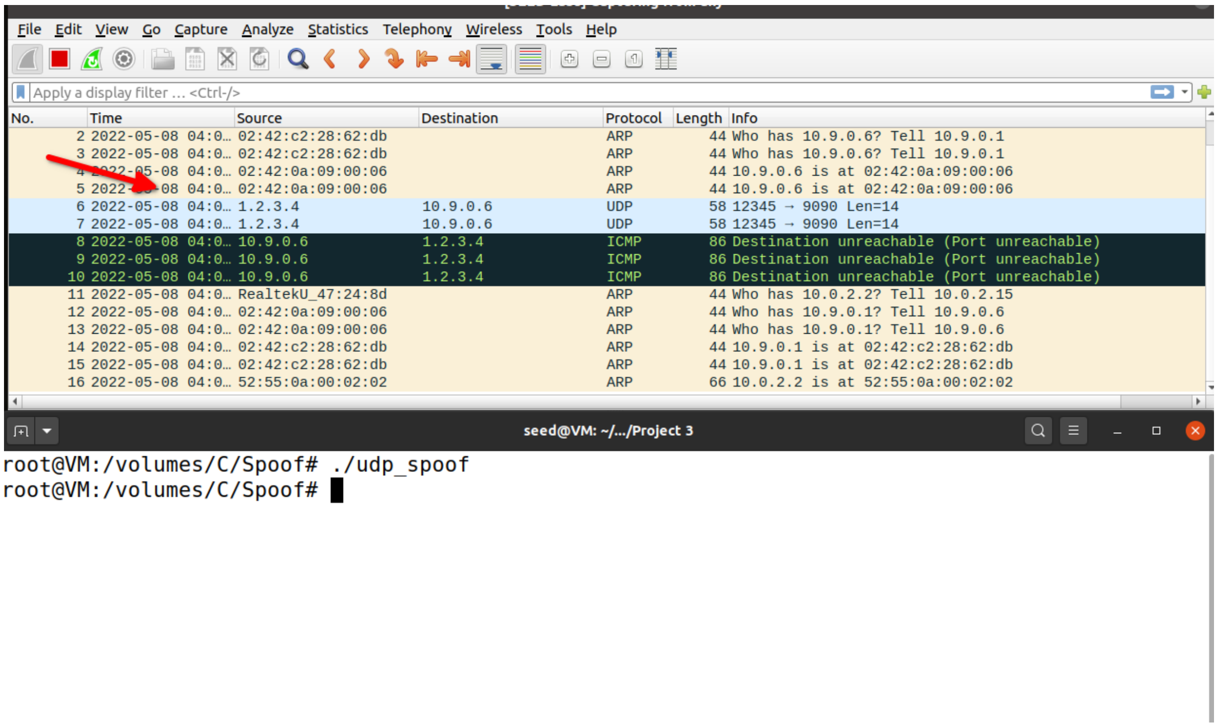
```
root@VM:/volumes/C# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: br-3833c27ad1ff
Number of packets: 1000
Filter expression: tcp dst port 23

Packet number 1:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 36048
   Dst port: 23

Packet number 2:
       From: 10.9.0.5
         To: 10.9.0.6
   Protocol: TCP
   Src port: 36048
   Dst port: 23

Packet number 3:
```

Scrolling down, we can see that packets 27 through 30 contain the captured password (*dees*)

Task 2.2: Spoofing

Task 2.2A: Write a Spoofing Program

For this task, I use *spoof_udp.c* (borrowed from book's code) with slight modifications. The spoofed packets will be sent from HostA (10.9.0.5), have a source address of 1.2.3.4 (a bogus address) and a destination address of 10.9.0.6 (adress of HostB). The Wireshark output shows that the program successfully spoofed a UDP packet with source 1.2.3.4 and destination 10.9.0.6


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
2	2022-05-08 04:0…	02:42:c2:28:62:db		ARP	44	Who has 10.9.0.6? Tell 10.9.0.1
3	2022-05-08 04:0…	02:42:c2:28:62:db		ARP	44	Who has 10.9.0.6? Tell 10.9.0.1
4	2022-05-08 04:0…	02:42:0a:09:00:06		ARP	44	10.9.0.6 is at 02:42:0a:09:00:06
5	2022-05-08 04:0…	02:42:0a:09:00:06		ARP	44	10.9.0.6 is at 02:42:0a:09:00:06
6	2022-05-08 04:0…	1.2.3.4	10.9.0.6	UDP	58	12345 → 9090 Len=14
7	2022-05-08 04:0…	1.2.3.4	10.9.0.6	UDP	58	12345 → 9090 Len=14
8	2022-05-08 04:0…	10.9.0.6	1.2.3.4	ICMP	86	Destination unreachable (Port unreachable)
9	2022-05-08 04:0…	10.9.0.6	1.2.3.4	ICMP	86	Destination unreachable (Port unreachable)
10	2022-05-08 04:0…	10.9.0.6	1.2.3.4	ICMP	86	Destination unreachable (Port unreachable)
11	2022-05-08 04:0…	RealtekU_47:24:8d		ARP	44	Who has 10.0.2.2? Tell 10.0.2.15
12	2022-05-08 04:0…	02:42:0a:09:00:06		ARP	44	Who has 10.9.0.1? Tell 10.9.0.6
13	2022-05-08 04:0…	02:42:0a:09:00:06		ARP	44	Who has 10.9.0.1? Tell 10.9.0.6
14	2022-05-08 04:0…	02:42:c2:28:62:db		ARP	44	10.9.0.1 is at 02:42:c2:28:62:db
15	2022-05-08 04:0…	02:42:c2:28:62:db		ARP	44	10.9.0.1 is at 02:42:c2:28:62:db
16	2022-05-08 04:0…	52:55:0a:00:02:02		ARP	66	10.0.2.2 is at 52:55:0a:00:02:02

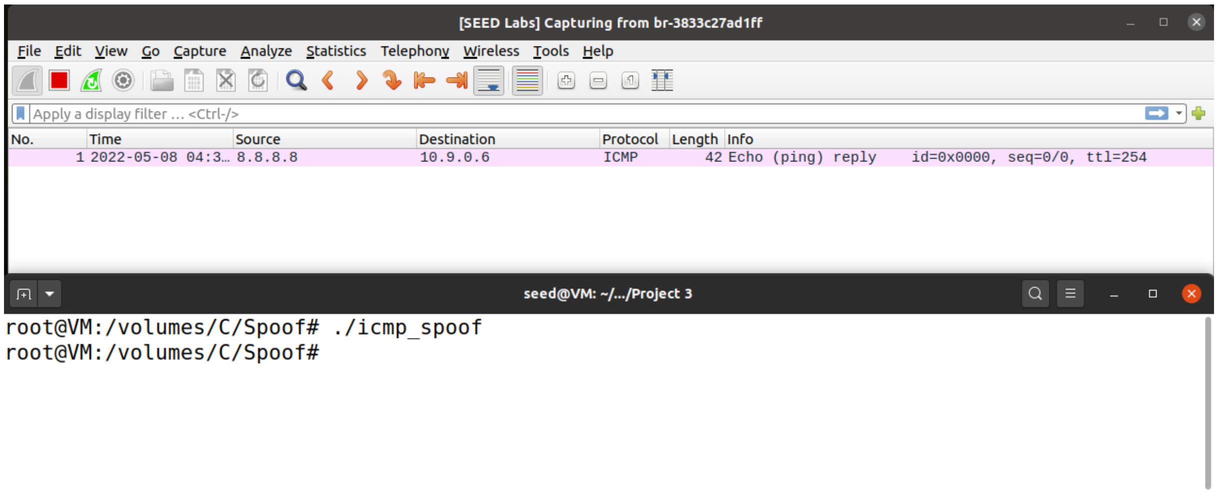
```
root@VM:/volumes/C/Spoof# ./udp_spoof
root@VM:/volumes/C/Spoof#
```

Task 2.2B: Spoof an ICMP Echo Request

Note that for this task, I use *spoof_icmp.c* with slight modifications (borrowed from the book's code).

Note that as the attacker I will spoof an ICMP request with source address *10.9.0.6* (address of HostA) and destination address *8.8.8.8*. If the spoofing is successful, HostA should receive an ECHO reply from *8.8.8.8*.

First, I run *spoof_icmp.c* from the attacker container. As the Wireshark output shows an ECHO request with source *10.9.0.6* and destination *8.8.8.8* is sent out and a subsequent ECHO reply from *8.8.8.8* is received.


[SEED Labs] Capturing from br-3833c27ad1ff
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
seed@VM: ~/.../Project 3

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-08 04:3…	8.8.8.8	10.9.0.6	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=254

```
root@VM:/volumes/C/Spoof# ./icmp_spoof
root@VM:/volumes/C/Spoof#
```

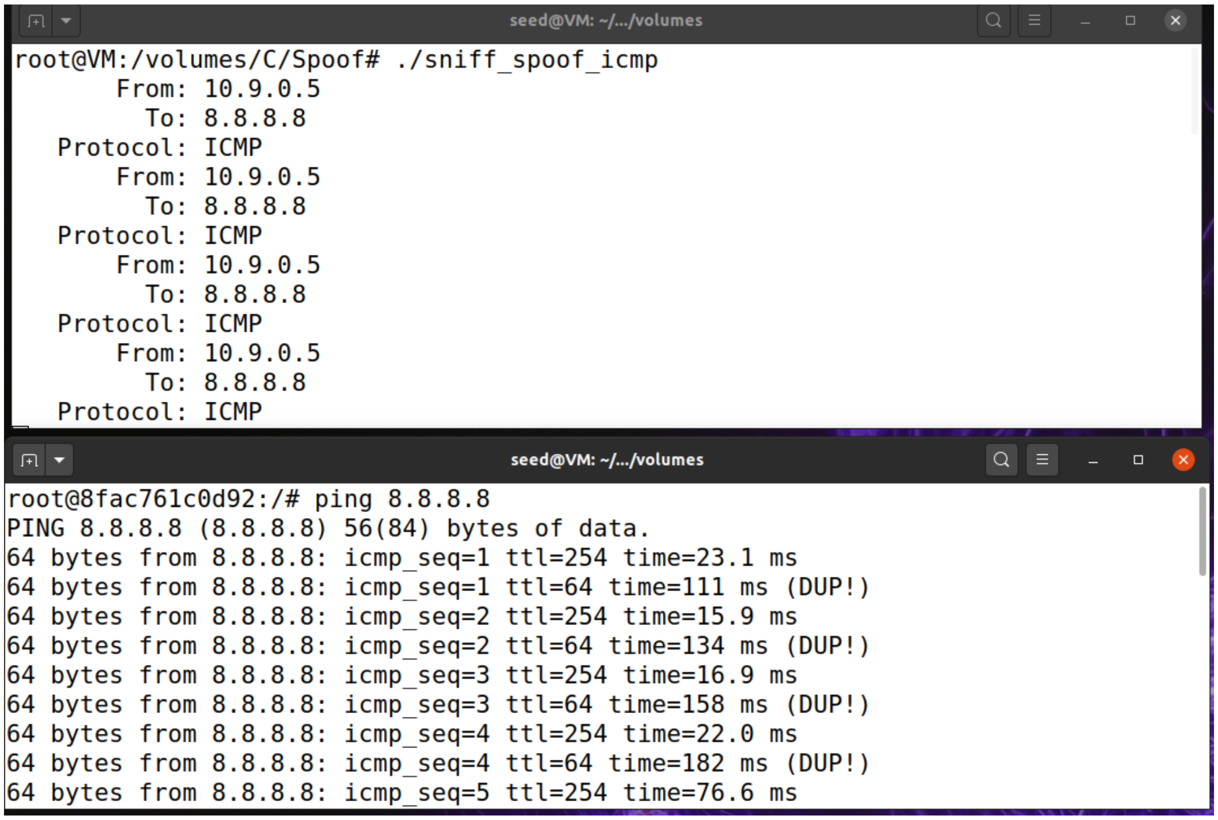
- **Question 4**: Yes, the IP packet length can be set to any value. I verified this by manually setting the packet length to an arbitrarily high number. No errors were observed
- **Question 5**: No, we do not have to calculate the checksum for the IP header. The checksum is automatically calculated by the system.
- **Question 6**: We need root privilege to create raw sockets and to put the NIC into promiscuous mode. When running as a normal user, the program will fail when calling the *pcap_open_live* function (which is responsible for creating a raw socket and putting the device into promiscuous mode)

Task 2.3: Sniff and then Spoof

The sniffer-spoofer program is defined in *sniff_spoof_icmp.c*. The most important addition to the previous programs is the following code snippet. In the snippet, the original ipheader is copied into a new ipheader which will serve as the spoofed ipheader. A new icmpheader is then created which will serve as the spoofed icmpheader. Next, the spoofed ip's source and destination are set to the original ip's destination and source respectively. Additionally, the ICMP type is set to 0 since we are expected to send an ICMP reply. Finally, the function used in previous tasks is used to send the raw packet.

```c
void send_icmp_reply(struct ipheader *ip)
{
  char buffer[512];

  memset((char *)buffer, 0, 512);
  memcpy((char *)buffer, ip, ntohs(ip->iph_len));
  struct ipheader *spoofed_ip = (struct ipheader *)buffer;
  struct icmpheader *spoofed_icmp = (struct icmpheader *)(buffer + ip->iph_ihl * 4);

  // Exchange source and destination addresses
  spoofed_ip->iph_sourceip = ip->iph_destip;
  spoofed_ip->iph_destip = ip->iph_sourceip;

  // Since this is an ICMP reply the type must be 0
  spoofed_icmp->icmp_type = 0;

  send_raw_ip_packet(spoofed_ip);
}
```

First, I run *sniif_spoof_icmp.c* on the attacker container. Then from HostA (10.9.0.5), I ping 8.8.8.8. The screenshot below shows that the sniffer-spoofer program sniffs the ICMP packets and successfully sends out spoofed packets. This is further confirmed by bottom terminal window which shows HostA successfully receiving ICMP replies.



```
root@VM:/volumes/C/Spoof# ./sniff_spoof_icmp
       From: 10.9.0.5
         To: 8.8.8.8
   Protocol: ICMP
       From: 10.9.0.5
         To: 8.8.8.8
   Protocol: ICMP
       From: 10.9.0.5
         To: 8.8.8.8
   Protocol: ICMP
       From: 10.9.0.5
         To: 8.8.8.8
   Protocol: ICMP
```

```
root@8fac761c0d92:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=254 time=23.1 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=111 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=254 time=15.9 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=134 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=254 time=16.9 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=158 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=254 time=22.0 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=64 time=182 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=5 ttl=254 time=76.6 ms
```