Task 1.1: Sniffing Packets

Task 1.1A

To obtain the correct interface name, I entered the command 'ifconfig' in the terminal to check the available interfaces. By running this command, I was able to view the list of interfaces along with their respective details. This allowed me to identify and determine the appropriate interface name needed for the further configuration or implementation of the program.

```
[05/02/23]seed@VM:~/.../volumes$ ifconfig
br-e1c8d2be6875: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:b5ff:fe81:a77f  prefixlen 64  scopeid 0x20<link>
        ether 02:42:b5:81:a7:7f  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 41  bytes 4993 (4.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:0b:a1:65:69  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.4  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::1c23:d622:e791:f461  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:06:d9:67  txqueuelen 1000  (Ethernet)
        RX packets 498211  bytes 748500912 (748.5 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 203614  bytes 12469004 (12.4 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 466  bytes 41556 (41.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 466  bytes 41556 (41.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Based on the information obtained from 'ifconfig', I will configure the sniff function from Scapy accordingly. I will monitor two interfaces and filter the packets to only consider ICMP control packets, such as ping packets. To print the packets, we will specifically focus on packets that use the ICMP protocol. By implementing this configuration, we can effectively capture and analyze ICMP packets, allowing us to observe and process the desired network traffic.

task1.py
~/Desktop/Lab2/volumes

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

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

iface = ['br-e1c8d2be6875','enp0s3']
pkt = sniff(iface=iface, filter='icmp', prn=print_pkt)
```

seed@VM: ~/.../Lab2

```
[05/02/23]seed@VM:~/.../Lab2$ ping www.ku.edu.tr
PING d6jmtx3ydlt78.cloudfront.net (18.66.97.99) 56(84) bytes of data.
64 bytes from server-18-66-97-99.fra56.r.cloudfront.net (18.66.97.99):
 icmp_seq=1 ttl=245 time=42.4 ms
64 bytes from server-18-66-97-99.fra56.r.cloudfront.net (18.66.97.99):
 icmp_seq=2 ttl=245 time=45.4 ms
^C
--- d6jmtx3ydlt78.cloudfront.net ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 42.429/43.922/45.416/1.493 ms
[05/02/23]seed@VM:~/.../Lab2$
```

seed@VM: ~/.../volumes

```
[05/02/23]seed@VM:~/.../volumes$ sudo python3 task1.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:06:d9:67
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 55230
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xe341
     src       = 10.0.2.4
     dst       = 18.66.97.99
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf1a4
        id        = 0x6
        seq       = 0x1
###[ Raw ]###
           load      = '\x06-Qd\x00\x00\x00\x00\xe2\xef\r\x00\x00\x00\
x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1
f !"#$%&\'()*+,-./01234567'
```

I initiated a ping request to our university's official webpage. From the right terminal, where our Python code is running, we can observe two types of packets. The first type is the echo-request packet, which has my address as the source and the address of Koc University's webpage as the destination. Following the echo-request packet, we can observe an echo-reply packet where the source and destination addresses are flipped.

It's important to note that to view the packets in the terminal, the command requires superuser privileges. Therefore, the command needs to be run with sudo to see the packets. Without using sudo, the packets cannot be displayed in the terminal.

```
[05/02/23]seed@VM:~/.../volumes$ python3 task1.py
/usr/local/lib/python3.8/dist-packages/scapy/layers/ipsec.py:471: Cryp
tographyDeprecationWarning: Blowfish has been deprecated
  cipher=algorithms.Blowfish,
/usr/local/lib/python3.8/dist-packages/scapy/layers/ipsec.py:485: Cryp
tographyDeprecationWarning: CAST5 has been deprecated
  cipher=algorithms.CAST5,
Traceback (most recent call last):
  File "task1.py", line 9, in <module>
    pkt = sniff(iface=iface, filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", lin
e 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", lin
e 894, in _run
    sniff_sockets.update(
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", lin
e 895, in <genexpr>
    (L2socket(type=ETH_P_ALL, iface=ifname, *arg, **karg),
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", l
ine 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
[05/02/23]seed@VM:~/.../volumes$
```

Task 1.1B

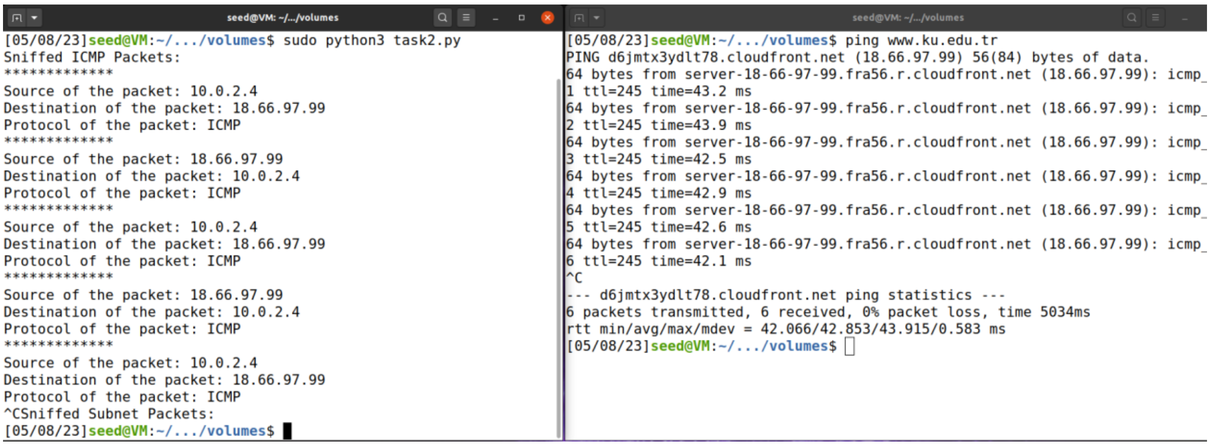
To address this question, I have declared several variables. Firstly, I have assigned a value to the "protocol_type" variable to identify the protocol type of the packet. Additionally, I have created three separate filters: one for ICMP packets, one for TCP packets, and another for subnet filtering. These filters allow me to selectively capture and analyze packets based on their respective protocols or subnet criteria. By incorporating these variables and filters into my code, I can effectively differentiate, and process packets based on their specific characteristics.

```python
from scapy.all import *

protocols = {
        1: "ICMP",
        6: "TCP"
}


icmp_filter = 'icmp'
tcp_filter = 'tcp and dst port 23'
subnet_filter = 'dst net 128.230.0.0/16'

iface = ['br-e1c8d2be6875','enp0s3']
```

In the first part, I want to filter out the ICMP packets. I wrote my code accordingly.

```python
icmp_filtered_pkts = sniff(iface=iface, filter=icmp_filter, count=5)

print('Sniffed ICMP Packets:')
for pkt in icmp_filtered_pkts:

        print("*************")
        print('Source of the packet: ' + str(pkt[IP].src))
        print('Destination of the packet: ' + str(pkt[IP].dst))
        print('Protocol of the packet: ' + protocols.get(pkt[IP].proto))
```

To verify the functionality of my packet filter, I will incorporate code to print out relevant information about the packet if the filter successfully captures it. To generate ICMP packets for testing, I will send a ping request to our university's website. You can observe the initiation of the ping request from the right terminal. From the left terminal, you can examine the captured packets and view their details, including the ICMP packets generated from the ping request to the university's website. This allows for analysis and verification of the packet filtering mechanism.



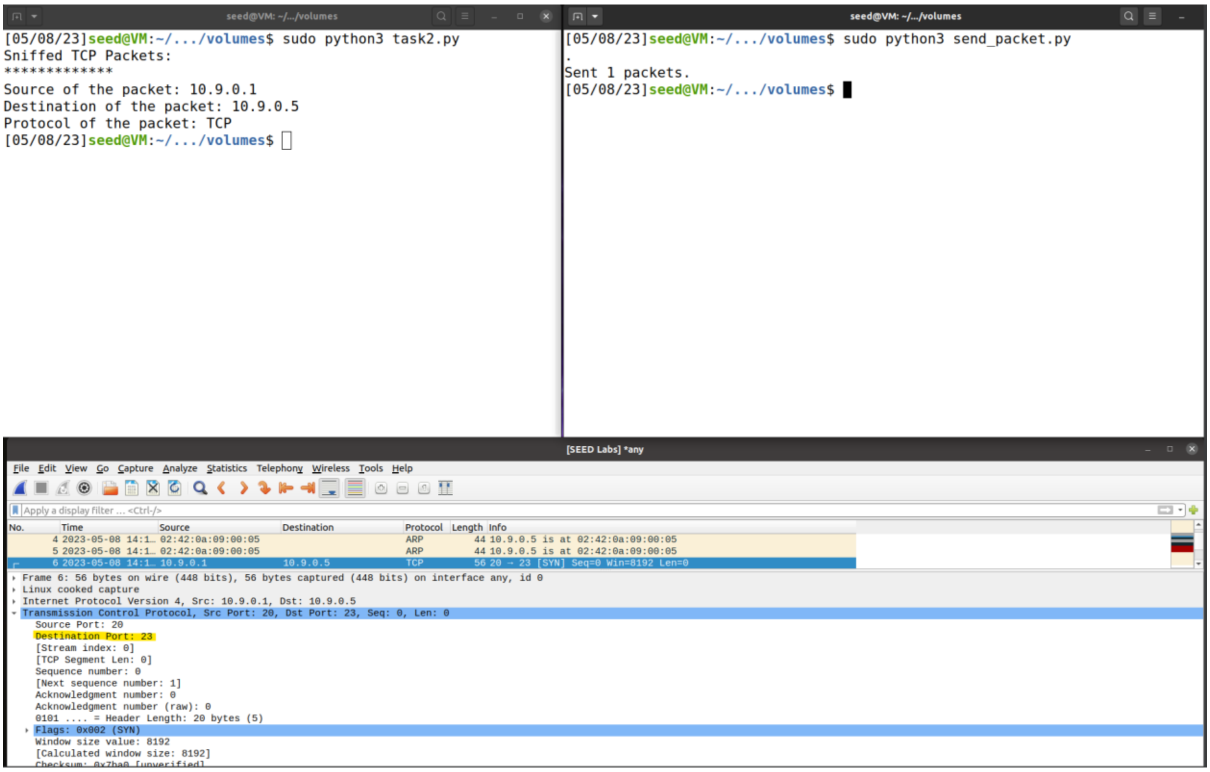
For the second part of the task, I implemented code to filter out TCP packets with a destination port number of 23. This filter allows me to specifically capture and process TCP packets that are intended for port 23. By customizing my code accordingly, I can focus on the desired packets and perform specific actions or analysis based on this filtered subset of TCP traffic.

```python
tcp_filtered_pkts = sniff(iface=iface, filter=tcp_filter, count=1)

print('Sniffed TCP Packets:')
for pkt in tcp_filtered_pkts:

        print("*************")
        print('Source of the packet: ' + str(pkt[IP].src))
        print('Destination of the packet: ' + str(pkt[IP].dst))
        print('Protocol of the packet: ' + protocols.get(pkt[IP].proto))
```

To verify the effectiveness of my packet filter, I will incorporate code to print out relevant information about the packet if the filter successfully captures it. To send a TCP packet with a destination port number of 23, I will create a Python script utilizing the basic send function from Scapy. This script will allow me to send the TCP packet to the desired destination with the specified port number. Additionally, you can also send text using Telnet, which also utilizes the TCP protocol. This will further facilitate testing and analysis of the captured packets within the specified scenario.

```
target_ip = '10.9.0.5'
target_port = 23


packet = IP(dst=target_ip) / TCP(dport=target_port)

send(packet)
```

Using the right terminal, I transmitted a TCP packet with a destination port number of 23. On the left terminal, I applied a filter to capture and isolate this specific packet. By examining the captured packets using Wireshark, you can observe the presence of the TCP packet with its corresponding port number, providing insights into the communication occurring on port 23.



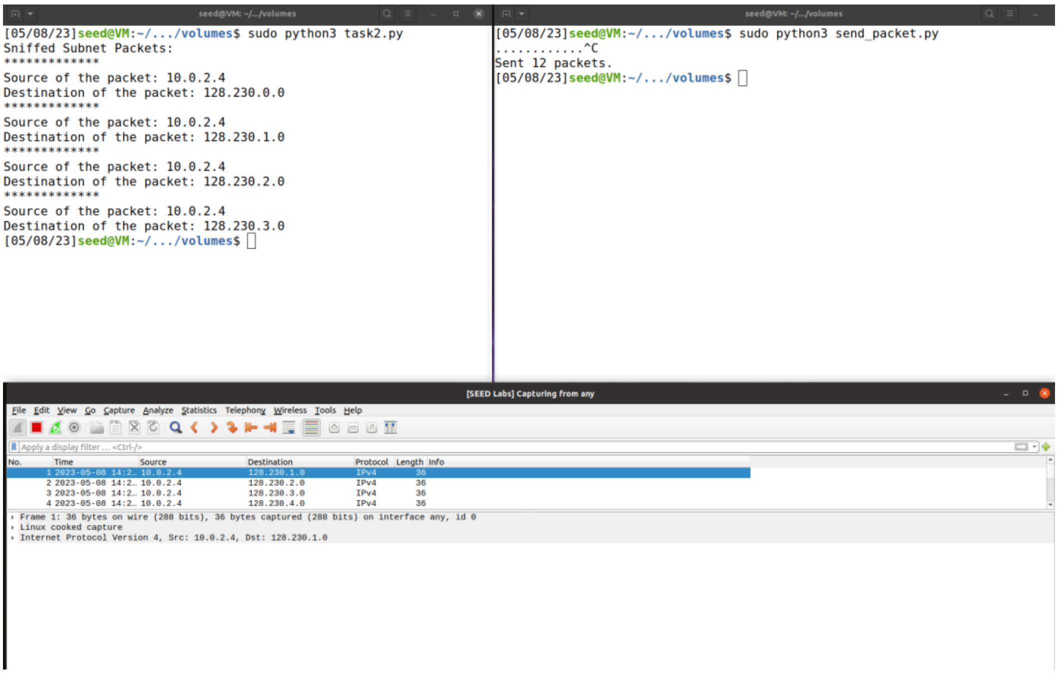
For the third part of the task, I implemented a packet filtering mechanism to specifically capture packets originating from or destined for a particular subnet. I customized my code accordingly to filter and process only those packets that meet this subnet criteria. By incorporating this filtering logic into my code, I can focus on packets that are relevant to the specified subnet, ensuring accurate analysis and handling of the captured packets.

```
subnet_filtered_pkts = sniff(iface=iface, filter=subnet_filter, count=4)

print('Sniffed Subnet Packets:')
for pkt in subnet_filtered_pkts:

        print("*************")
        print('Source of the packet: ' + str(pkt[IP].src))
        print('Destination of the packet: ' + str(pkt[IP].dst))
```

To verify the functionality of my packet filter, I will print out relevant information about the packet if the filter successfully captures it. To transmit the packet to the desired subnet, I will utilize the previous send function with minor modifications to accommodate the specific subnet settings. By incorporating these changes, I can ensure that the packet is correctly sent to the intended subnet and the filter is properly applied.

```
ip = IP()
ip.dst = '128.230.0.0/16'
send(ip,4)
```

Using the right terminal, I transmitted a packet to a specific subnet. On the left terminal, I filtered and captured this packet. By examining the packet using Wireshark, you can observe the presence of a TCP packet along with its corresponding port number.

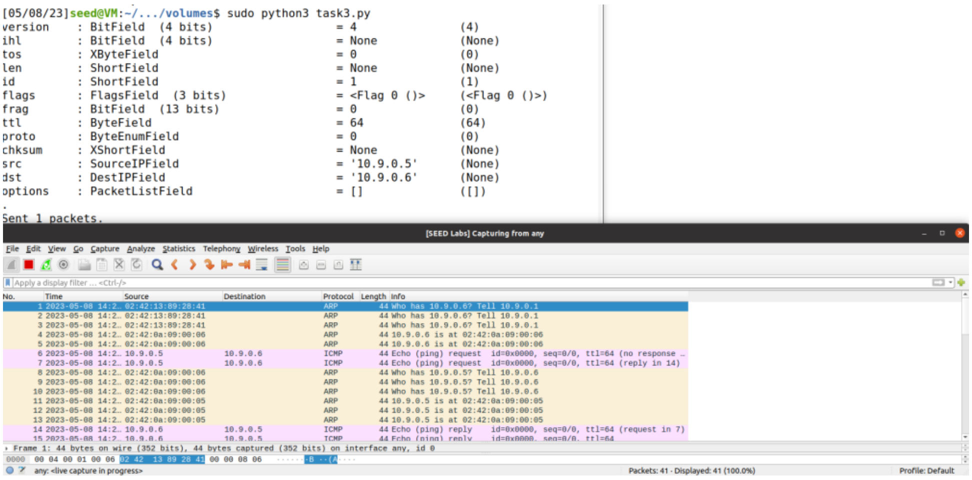


Task 1.2

To perform ICMP packet spoofing, I developed a Python script. Within the script, I instantiated an IP object, specifying the source and destination addresses based on the containers defined in the Docker Compose configuration. By setting the appropriate source and destination addresses, I was able to manipulate and spoof the ICMP packets as required.

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

from scapy.all import *

a = IP()
a.src = '10.9.0.5'
a.dst = '10.9.0.6'
b = ICMP()
p = a/b
ls(a)
send(p)
```

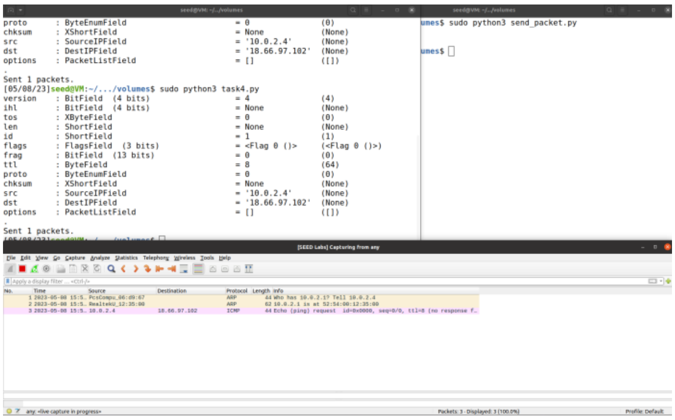
I executed the script from the terminal and used the ls() function from Scapy to view the packet details. You can observe the packet details directly from the terminal output. Additionally, you can also examine the packets using Wireshark, and the captured packets are displayed below for your reference.



Task 1.3

I initiated the ping request from the user container using the source address 10.9.0.5. In the left terminal, you can observe the captured ICMP echo-request and echo-reply packets using the sniff-and-then-spoof program. In the right terminal, you can see the execution of the ping command, which sends ICMP echo-request packets. Below, you can verify that both the echo-request and echo-reply packets were successfully delivered. To reach a specific IP address, I implemented a brute force search starting from a TTL (Time to Live) value of 1. Through this search, I discovered that the minimum TTL required for my packet to successfully reach the specific IP is 8. The TTL value determines the maximum number of network hops a packet can take before being discarded. By incrementally increasing the TTL and observing successful packet delivery, I determined the minimum TTL needed for my target IP address.

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

from scapy.all import *

a = IP()
a.dst = '18.66.97.102'
a.ttl = 8
b = ICMP()
p = a/b
ls(a)
send(p)
```



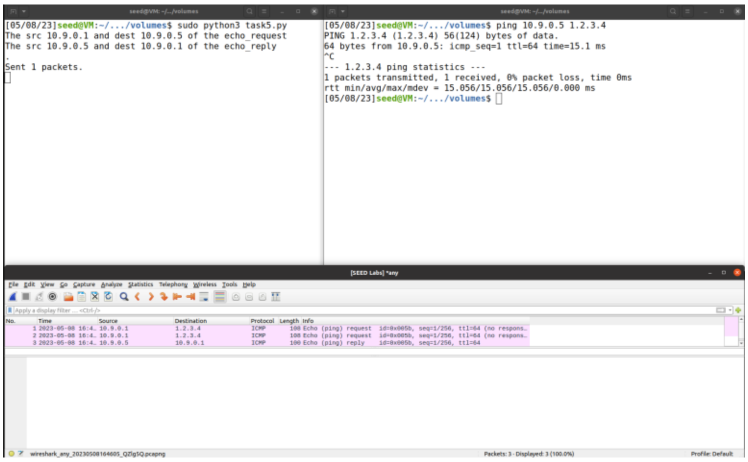
Task 1.4

To perform packet spoofing, I have written a Python script like the ones used in previous tasks. However, in this case, I am specifically checking if the ICMP packet contains an error message. If an error message is present in the ICMP packet, it will typically indicate an "unreachable address." To create a spoofed echo reply, I swap the source and destination addresses in the packet. Additionally, I include sequential, ID, and payload values in the packet, as they are essential for generating an appropriate echo reply for a non-existing host. Finally, I use the send function to transmit the spoofed packet.

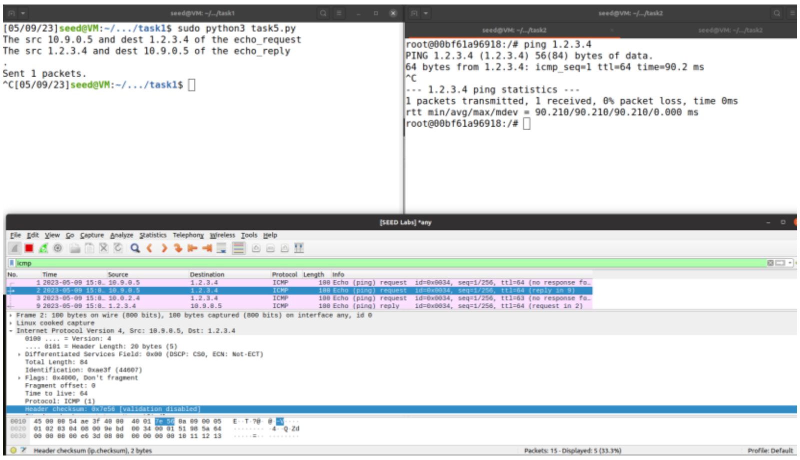
```python
#!/usr/bin/env python3
from scapy.all import *

def check_for_icmp_packets(pkt):

        pkt.show()

        # Protocol type 1 means ICMP
        if pkt[IP].proto == 1:

                temp_src = 0
                temp_dest = 0

                # Type 3 = ICMP.type.ICMP_DEST_UNREACH
                if pkt[ICMP].type == 3:
                        temp_src = pkt[ICMP].payload.src
                        temp_dest = pkt[ICMP].payload.dst
                else:
                        temp_src = pkt[IP].src
                        temp_dest = pkt[IP].dst

                seq = pkt[ICMP].seq
                id = pkt[ICMP].id
                load = pkt[Raw].load
                print("The src {src} and dest {dest} of the echo_request".format(src=temp_src, dest=temp_dest))
                print("The src {src} and dest {dest} of the echo_reply".format(src=temp_dest, dest=temp_src))

                a = IP()
                a.dst = temp_src
                a.src = temp_dest
                b = ICMP(type=0, id=id, seq=seq)
                p = a/b/load
                send(p)

iface = ['enp0s3', 'lo']
pkt = sniff(iface=iface, filter='icmp', prn=check_for_icmp_packets, count =1)
```

I sent the ping request from the source address of the user container, which is 10.9.0.5. In the left terminal, you can observe the sniffed ICMP echo-request and echo-reply packets. In the right terminal, you can see that the ping command has been executed and ICMP echo-request packets have been sent. Below, you can confirm that the echo-request and echo-reply packets were successfully delivered.

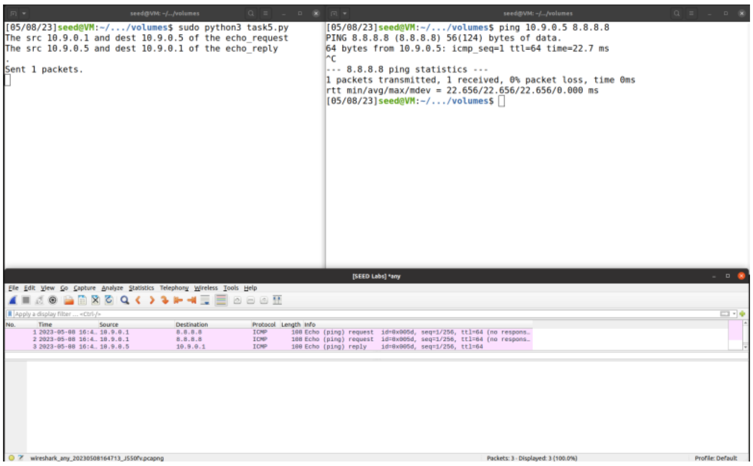
For the address 1.2.3.4



For the address 10.9.0.99



For the address 8.8.8.8

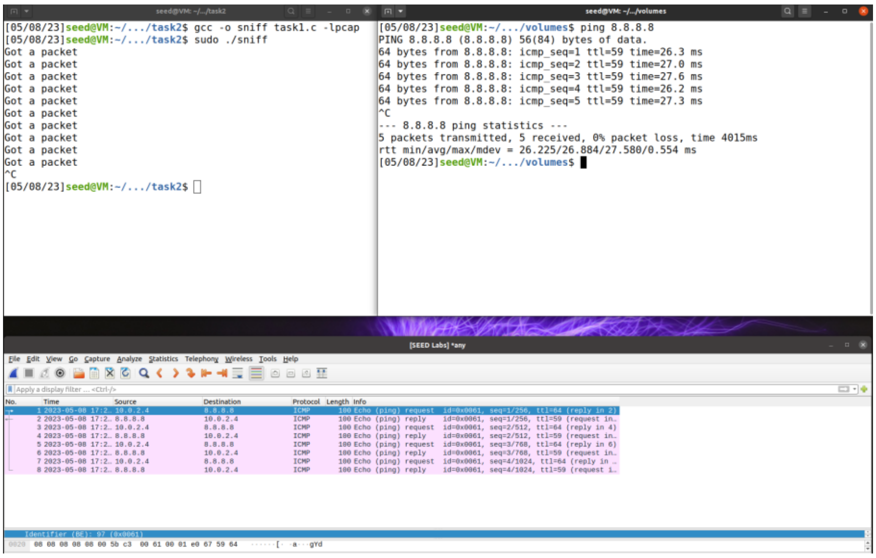


Task 2.1 Writing Packet Sniffing Program

Task 2.1A

I wrote the same code in the instructions. You can see the output in the below:

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <netinet/ether.h>
#include <netinet/ip.h>

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

int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  //char filter_exp[] = "host 10.0.2.4 and host 7.7.7.7 and icmp";
  //char filter_exp[] = "tcp dst portrange 10-100";
  char filter_exp[] = "icmp";
  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);

  // Step 2: Compile filter_exp into BPF psuedo-code
  pcap_compile(handle, &fp, filter_exp, 0, net);
  if (pcap_setfilter(handle, &fp) != 0) {
        pcap_perror(handle, "Error:");
        exit(EXIT_FAILURE);
  }

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

  pcap_close(handle);   //Close the handle
  return 0;
}
```



Question 1:

- The essential sequence of library calls for sniffer programs involves opening a session on the Network Interface Card (NIC) to capture packets (this is done by the pcap_open_live). Once the session is established, a filter is applied to selectively capture specific packets of interest. As packets are captured, a designated function (such as `got_packet`) is executed to process and analyze the captured packets.

Question 2:

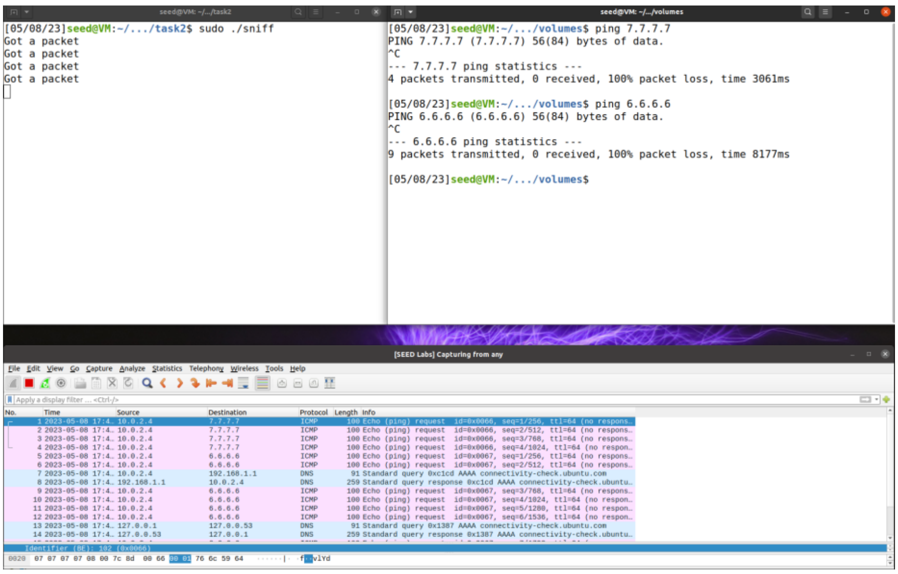
- When executing a sniffer program without using the sudo keyword or without having root privileges, you may encounter a segmentation error. This error occurs because sniffer programs require access to network interfaces, as we discussed earlier. Low-level access to the network interfaces necessitates root privileges, which is why the sudo keyword is needed. By using sudo and running the sniffer program with root access, you can overcome these limitations and prevent the segmentation error caused by insufficient privileges.

Question 3:

- Enabling promiscuous mode in a sniffer program (setting the value to 1 in pcap_open_live()) allows the network interface card (NIC) to capture all packets on the network. On the other hand, when promiscuous mode is disabled (setting the value to 0 in pcap_open_live()), the NIC operates in non-promiscuous mode. In this mode, the NIC only captures packets that are specifically addressed to it or broadcast/multicast packets. This allows for a more targeted capture of network traffic, filtering out irrelevant packets that are not intended for the specific interface.

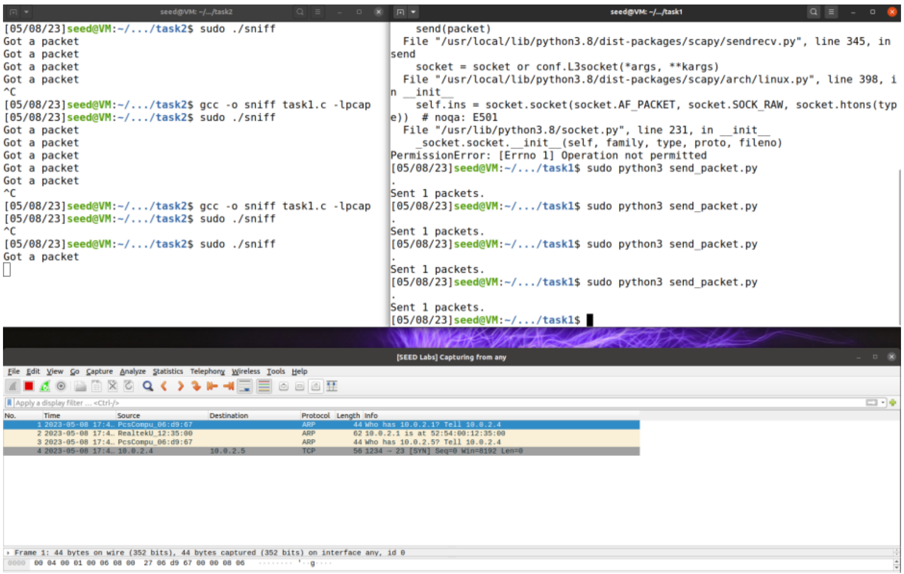
Task 2.1B

My first pcap filter is “host 10.0.2.4 and host 7.7.7.7 and icmp”. I sent two pings one with 7.7.7.7 and the other is 6.6.6.6. As you can see from the image, my sniffer only captures the packets from the command ‘ping 7.7.7.7’.



My second pcap filter is “tcp dst portrange 10-100”. I sent TCP packet from one container address to another container address. I write a python script to send packet. You can see the code and the terminals below:

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

from scapy.all import *
import socket

src_ip = '10.0.2.4'
src_port = 1234

target_ip = '10.0.2.5'
target_port = 23

packet = IP(src=src_ip, dst=target_ip) / TCP(sport=src_port, dport=target_port)
send(packet)
```

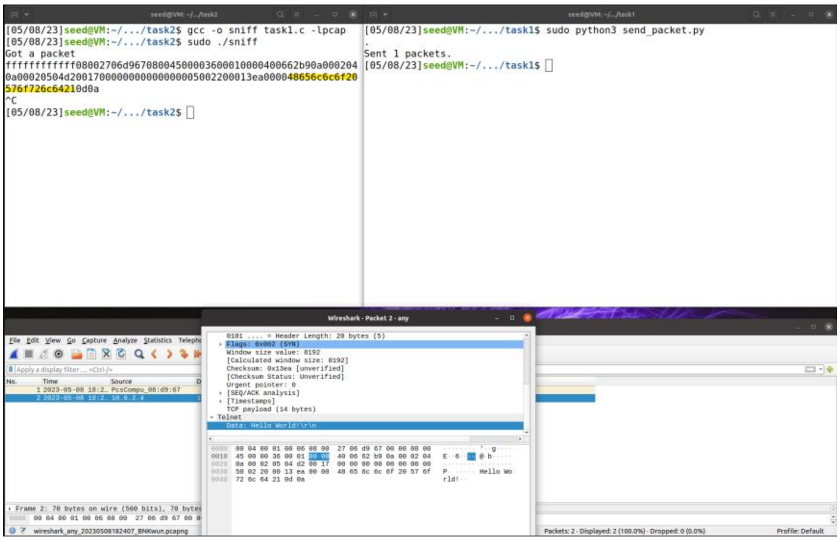


Task 2.1C

Using the same Python script to send packets, I added an extra payload message ("Hello, World!"). By examining the packet data in hexadecimal format, I successfully detected this message (referred to as a password) within the packet contents. The hexdecimal value of the password can be observed both in WireShark, a network protocol analyzer, and in my custom sniffer program.

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

        for (int i =0; i<header->len;i++){
                printf("%02x", packet[i]);
        }

        printf("\n");

}
```

```python
from scapy.all import *
import socket

src_ip = '10.0.2.4'
src_port = 1234

target_ip = '10.0.2.5'
target_port = 23

payload = "Hello World!\r\n"

packet = IP(src=src_ip, dst=target_ip) / TCP(sport=src_port, dport=target_port) / payload
send(packet)
```

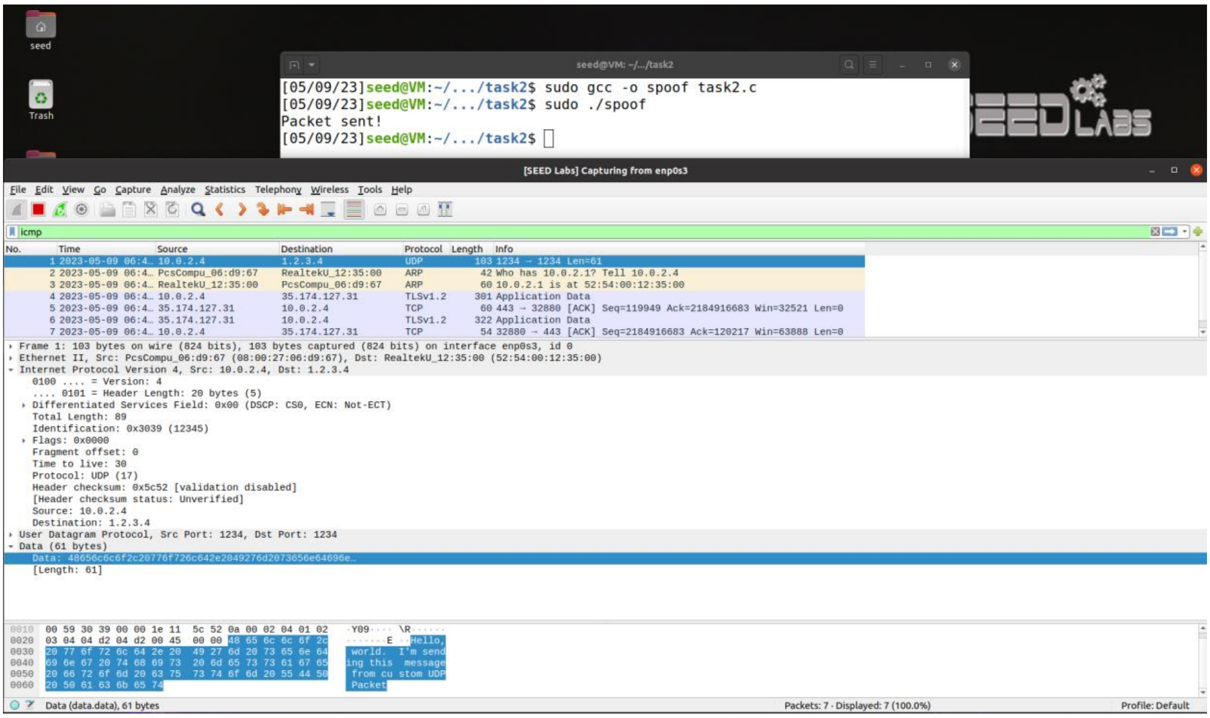


Task 2.2 Spoofing

Task 2.2A

Below, you will find my personal C program for packet spoofing, where I aim to send a UDP packet containing a specific message for easier identification. The UDP packet is being sent from the IP address '10.0.2.4' to the destination IP address '1.2.3.4'.

```
int main() {
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sock < 0) {
        perror("Socket");
        exit(1);
    }

    char packet[4096];
    memset(packet, 0, sizeof(packet));

    struct iphdr *ip_header = (struct iphdr *) packet;
    ip_header->ihl = 5;
    ip_header->version = 4;
    ip_header->tos = 0;
    ip_header->tot_len = sizeof(struct iphdr) + sizeof(struct udphdr) + strlen("Hello, world. I'm sending this message from custom UDP 
Packet");
    ip_header->id = htons(12345);
    ip_header->frag_off = 0;
    ip_header->ttl = 30;
    ip_header->protocol = IPPROTO_UDP;
    ip_header->saddr = inet_addr("10.0.2.4");
    ip_header->daddr = inet_addr("1.2.3.4");

    struct udphdr *udp_header = (struct udphdr *) (packet + sizeof(struct iphdr));
    udp_header->source = htons(1234);
    udp_header->dest = htons(1234);
    udp_header->len = htons(sizeof(struct udphdr) + strlen("Hello, world. I'm sending this message from custom UDP Packet"));
    udp_header->check = 0;

    char *msg = packet + sizeof(struct iphdr) + sizeof(struct udphdr);
    strcpy(msg, "Hello, world. I'm sending this message from custom UDP Packet");

    struct sockaddr_in sin;
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = ip_header->daddr;

    if (sendto(sock, packet, ip_header->tot_len, 0, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
        perror("SENDTO");
        exit(1);
```

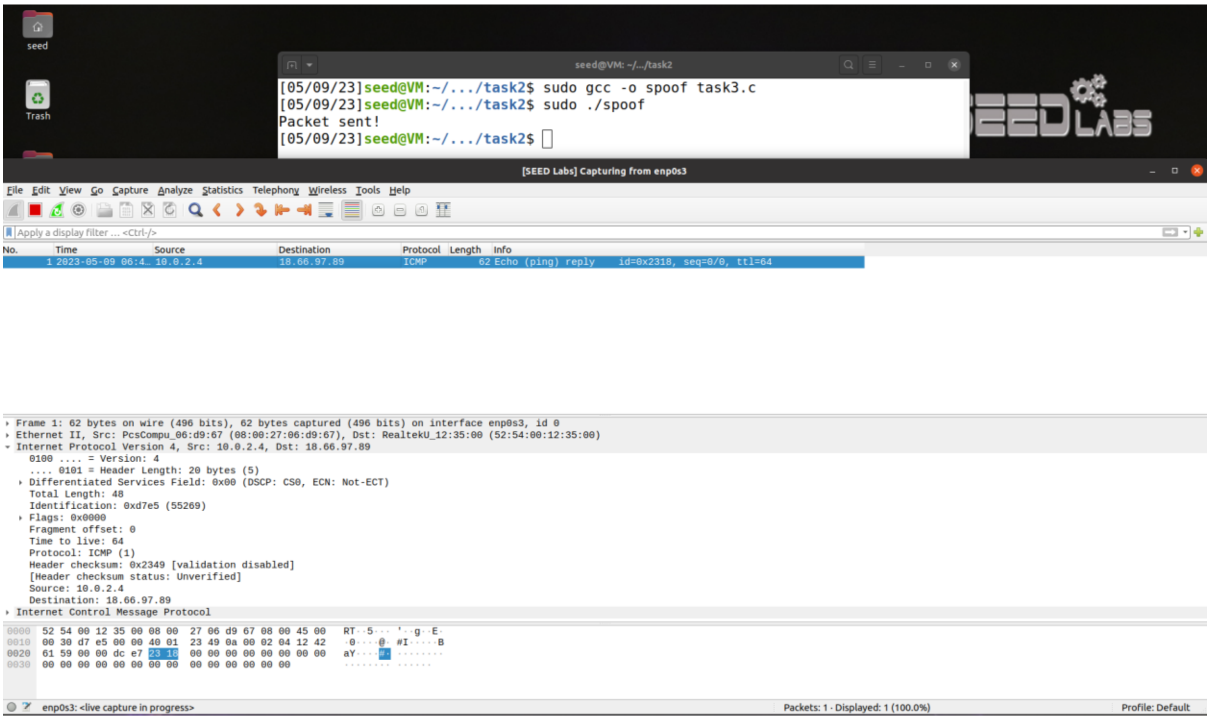


You can observe the correct source and destination addresses in the packet. Additionally, within the packet itself, you can find the message that I have included.

Task 2.2B

In this task, my objective is to send an ICMP echo request packet on behalf of our university's webpage to my virtual machine (VM). Below, you will find my customized C program for packet spoofing, designed specifically to send ICMP echo-request packets. The ICMP packet is being sent from the IP address '18.66.97.89' (obtained through a ping to www.ku.edu.tr) to the destination IP address '10.0.2.4'.

```
int main() {
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sock < 0) {
        perror("Socket");
        exit(1);
    }

    char packet[PACKET_SIZE];
    memset(packet, 0, PACKET_SIZE);

    struct iphdr *ip_header = (struct iphdr *) packet;
    ip_header->ihl = 5;
    ip_header->version = 4;
    ip_header->tos = 0;
    ip_header->tot_len = sizeof(struct iphdr) + sizeof(struct icmp);
    ip_header->id = htons(12345);
    ip_header->frag_off = 0;
    ip_header->ttl = 30;
    ip_header->protocol = IPPROTO_ICMP;
    ip_header->saddr = inet_addr("18.66.97.89");
    ip_header->daddr = inet_addr("10.0.2.4");

    struct icmp *icmp_header = (struct icmp *) (packet + sizeof(struct iphdr));
    icmp_header->icmp_type = ICMP_ECHO;
    icmp_header->icmp_code = 0;
    icmp_header->icmp_id = getpid();
    icmp_header->icmp_seq = 0;
    icmp_header->icmp_cksum = 0;
    icmp_header->icmp_cksum = checksum(icmp_header, sizeof(struct icmp));

    struct sockaddr_in sin;
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = ip_header->daddr;

    if (sendto(sock, packet, ip_header->tot_len, 0, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
        perror("SENDTO");
        exit(1);
    }

    printf("Packet sent!\n");
```



As a result of our packet spoofing efforts, we have received an echo-reply from our virtual machine (VM) to the address 18.66.97.89, which represents the university webpage. This outcome confirms that our echo-request packet was successfully delivered on behalf of another machine, demonstrating the effectiveness of our packet spoofing program.

Question 4

- While you can assign an arbitrary value to the IP packet length field, the actual packet's total length is typically overwritten with the correct value when it's sent. Modifying the packet length field may result in the packet being rejected or dropped by the destination address due to mismatched lengths.

Question 5

- You typically do not need to explicitly calculate the checksum for the IP header when utilizing raw socket programming. The IP checksum is calculated for you by the operating system. The operating system will automatically calculate and fill in the right checksum value before transmitting the packet if you set the IP checksum field in the IP header to 0. If you decide to calculate the checksum manually, you must do so in accordance with the IP header structure and protocol requirements.

Question 6

- Root privilege is necessary to run programs that utilize raw sockets because they provide low-level access to the network interface (NIC). This type of access bypasses certain built-in protections and can be susceptible to misuse for malicious purposes. Thus, the requirement for root privileges ensures that users have the necessary authority to manipulate network traffic. Without root privileges, executing programs that utilize raw sockets will be unsuccessful due to the operating system's restrictions on low-level network access.

Task 2.3

In this task, I have combined the code from the sniffing and spoofing tasks. The merged code is quite similar, with the main difference being the modification of the source and destination addresses to send an echo-reply packet. Here is the essential part of the code where the function is executed every time pcap receives a packet based on our filter, which in this case is set to capture only ICMP packets.

```c
struct ether_header *eth_header = (struct ether_header *) packet;
struct ip *fetched_ip_header = (struct ip *) (packet + sizeof(struct ether_header));
struct icmp *fetched_icmp_header = (struct icmp *) (packet + sizeof(struct ether_header) + fetched_ip_header->ip_hl * 4);

if (fetched_icmp_header->icmp_type != ICMP_ECHO) {
    return;
}

int ip_header_len = fetched_ip_header->ip_hl * 4;
int PACKET_LEN = ntohs(fetched_ip_header->ip_len);
char buffer[PACKET_LEN];

memset(buffer, 0, PACKET_LEN);
memcpy(buffer, fetched_ip_header, ntohs(fetched_ip_header->ip_len));
struct ip *newip = (struct ip *) buffer;
struct icmp *newicmp = (struct icmp *) (buffer + ip_header_len);

newip->ip_src.s_addr = fetched_ip_header->ip_dst.s_addr;
newip->ip_dst.s_addr = fetched_ip_header->ip_src.s_addr;
newip->ip_ttl = 64;

newicmp->icmp_type = ICMP_ECHOREPLY;
newicmp->icmp_cksum = 0;
newicmp->icmp_cksum = checksum(newicmp, ntohs(newip->ip_len) - newip->ip_hl * 4);

struct sockaddr_in dest_info;
dest_info.sin_family = AF_INET;
dest_info.sin_addr = newip->ip_dst;

int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
int enable = 1;
setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
```

I initiated the Docker console from the terminal to perform a LAN ping. I chose '1.2.3.4' as the destination address. On the left terminal, you can observe that I started my packet spoofing program. Meanwhile, on the right terminal, I sent a ping command from the Docker's terminal to '1.2.3.4'. At the bottom, Wireshark displays the captured echo-request and echo-reply packets. As evident, there is no packet loss, as indicated by the output in the right terminal.