CPS 633 Section 09

Packet Sniffing and Spoofing Lab

Group 18

Roxie Reginold (501087897)
Hetu Virajkumar Patel (501215707)
Sayyada Aisha Mehvish (501106795)

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

Task 1.1: Sniffing Packets

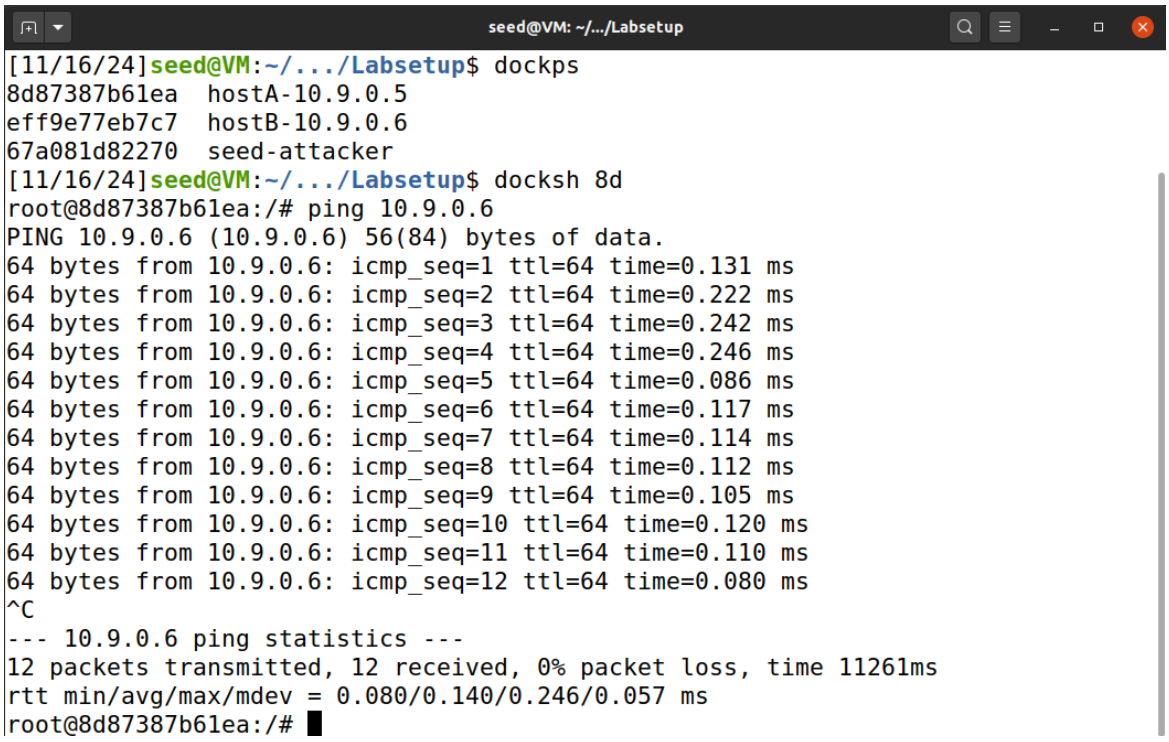
Task 1.1A.

Run the program with the root privilege and demonstrate that you can indeed capture packets. After that, run the program again, but without using the root privilege; describe and explain your observations.

```
[11/16/24]seed@VM:~/.../Labsetup$ ifconfig
br-9ec536502146: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:73ff:feb1:8f0f  prefixlen 64  scopeid 0x20<link>
        ether 02:42:73:b1:8f:0f  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 38  bytes 5575 (5.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

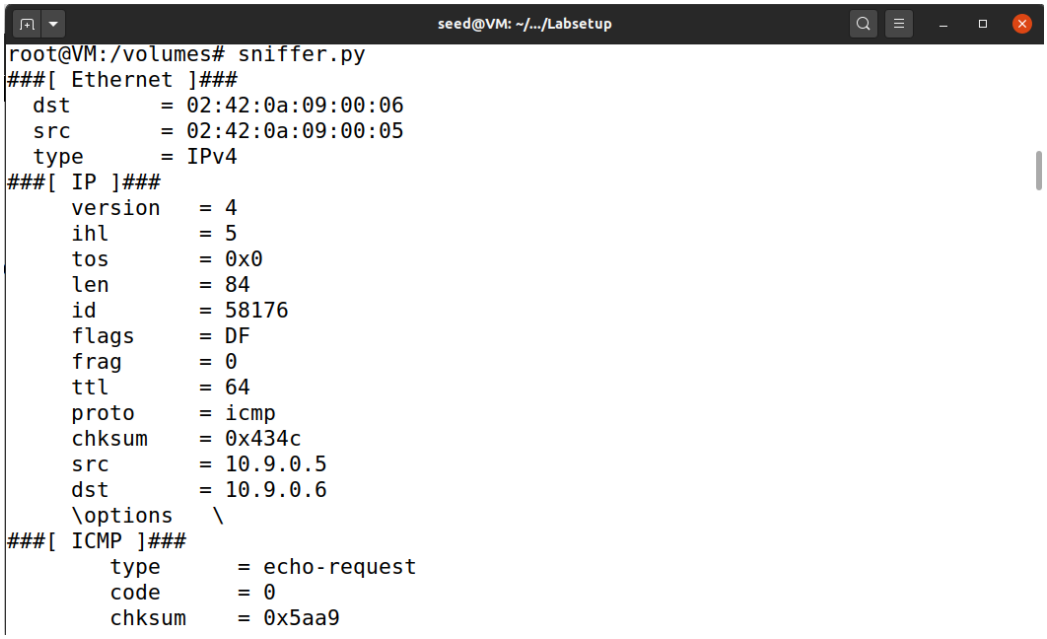
Here is our sniffer program in which we sniff our network interface from above.

To create network traffic, I will ping host B from host A.

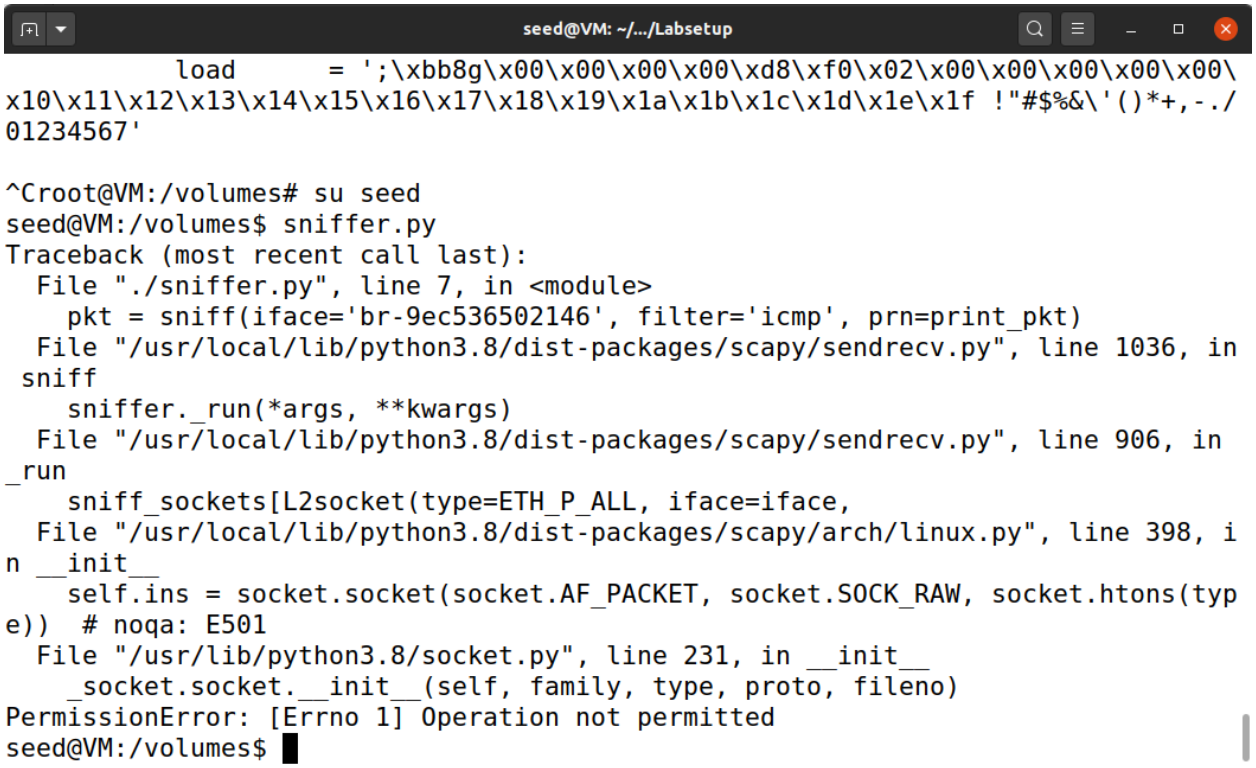


The sniffer program successfully captures packets on the specified network interface, displaying their details as intended.

When we run the program with root privileges:



When we run the program without root privileges:



We encounter an issue running the program correctly because packet sniffing requires interacting with low-level network interfaces, a privilege typically restricted to root users for security reasons. Non-root users lack access to raw sockets or interfaces necessary for sniffing network traffic.

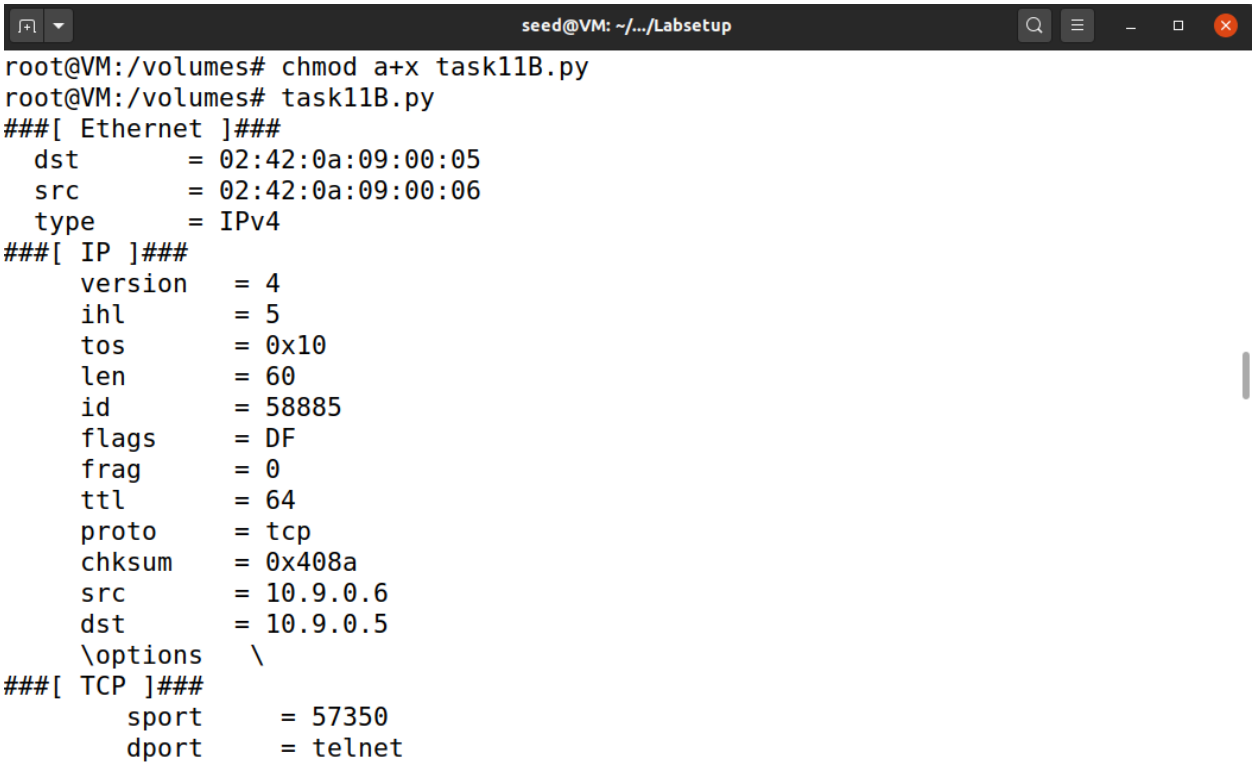
Task 1.1B.

- **Capture only the ICMP packet**
  - **(already done in task 1.1A)**
  - **Successfully demonstrated earlier in Task 1.1A by filtering and displaying ICMP traffic.**

- **Capture any TCP packet that comes from a particular IP and with a destination port number 23.**
  Created a new program to run the task and use a source IP of our choice and destination port of 23.

We run this program using the telnet command because this protocol is used to establish connection to TCP port number 23.

With this we are able to capture TCP packets that match the specified source IP and destination port 23.



```
seed@VM: ~/.../Labsetup
root@eff9e77eb7c7:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
8d87387b61ea 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
```

- **Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to**

```
seed@VM: ~/.../volumes
  GNU nano 4.8                                  task11B.py
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

# ICMP
#pkt = sniff(iface='br-9ec536502146', filter='icmp', prn=print_pkt)

# any TCP packet that comes from a particular IP and with a destination port number 23.
#pkt = sniff(iface='br-9ec536502146', filter='tcp && src host 10.9.0.6 && dst port 23', prn=print_pkt)

# packets comes from or to go to a particular subnet. You can pick any subnet, such as
# 128.230.0.0/16; you should not pick the subnet that your VM is attached to.
pkt = sniff(iface='br-9ec536502146', filter='net 128.230.0.0/16', prn=print_pkt)
```

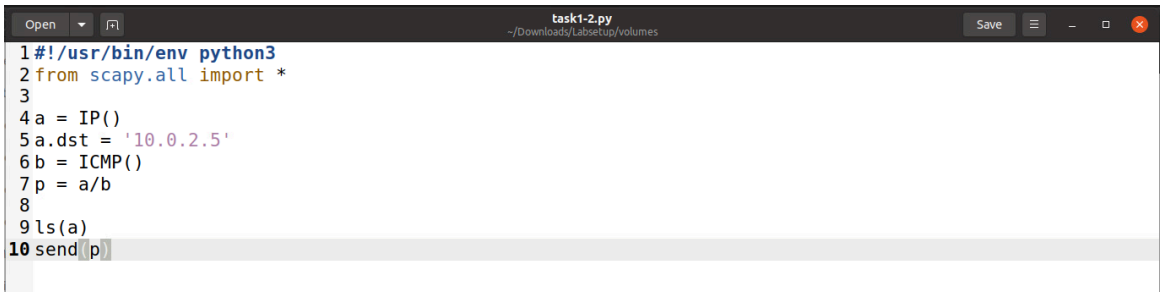
```
seed@8d87387b61ea:~$ ping 128.230.0.11
PING 128.230.0.11 (128.230.0.11) 56(84) bytes of data.
^C
--- 128.230.0.11 ping statistics ---
20 packets transmitted, 0 received, 100% packet loss, time 19577ms
```

The filter captures all traffic matching the specified subnet i.e. 128.230.0.0, demonstrating the program's ability to focus on network subsets.

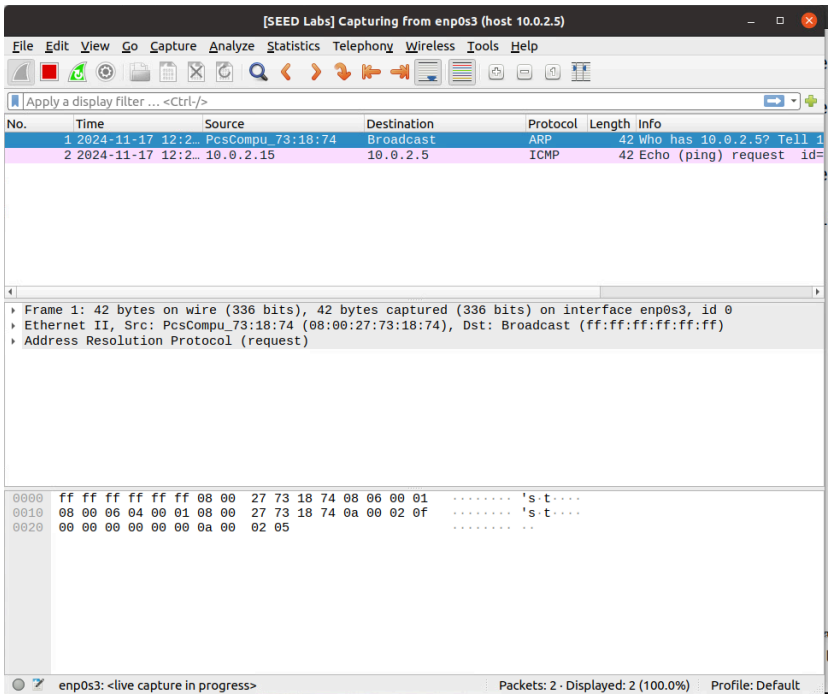
Task 1.2: Spoofing ICMP Packets

```
root@VM:/volumes# task1-2.py
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                       = '10.0.2.15'     (None)
dst        : DestIPField                         = '10.0.2.5'      (None)
options    : PacketListField                     = []              ([])
.
Sent 1 packets.
root@VM:/volumes#
```

The program generates ICMP Echo Request packets with a spoofed source IP and sends them to a target host:



The code to spoof an ICMP echo request with any arbitrary source IP address – here 10.0.2.15.

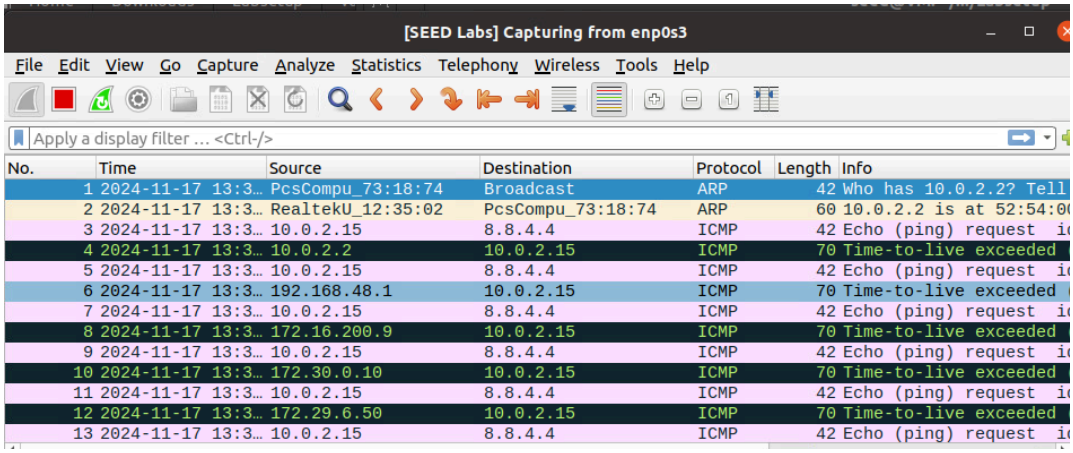


Wireshark confirms receipt of the ICMP Echo Reply sent back to the spoofed IP address, proving the success of the spoofing operation.

Task 1.3: Traceroute

task1-3.py
~/Downloads/Labsetup/volumes

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

TTL = 0
while(True):
        TTL += 1
        a = IP(dst="8.8.4.4", ttl=TTL)
        b = ICMP()
        p = a/b
        reply = sr1(p)
        print ("Source IP: ", reply[IP].src)
        if (reply[IP].src == "8.8.4.4"):
                break
```



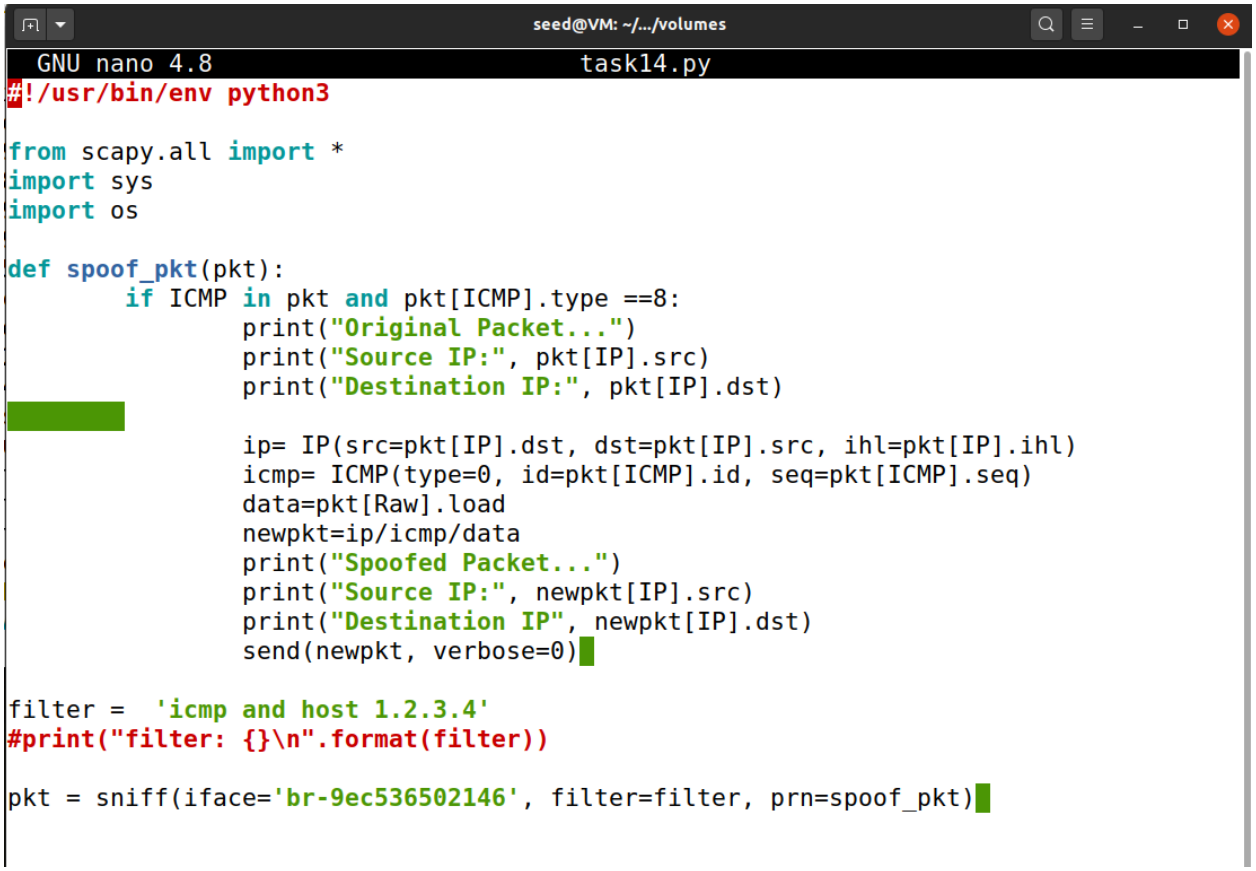

```
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Source IP:  74.125.48.230
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Source IP:  192.178.98.113
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Source IP:  216.239.49.189
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Source IP:  8.8.4.4
root@VM:/volumes#
```

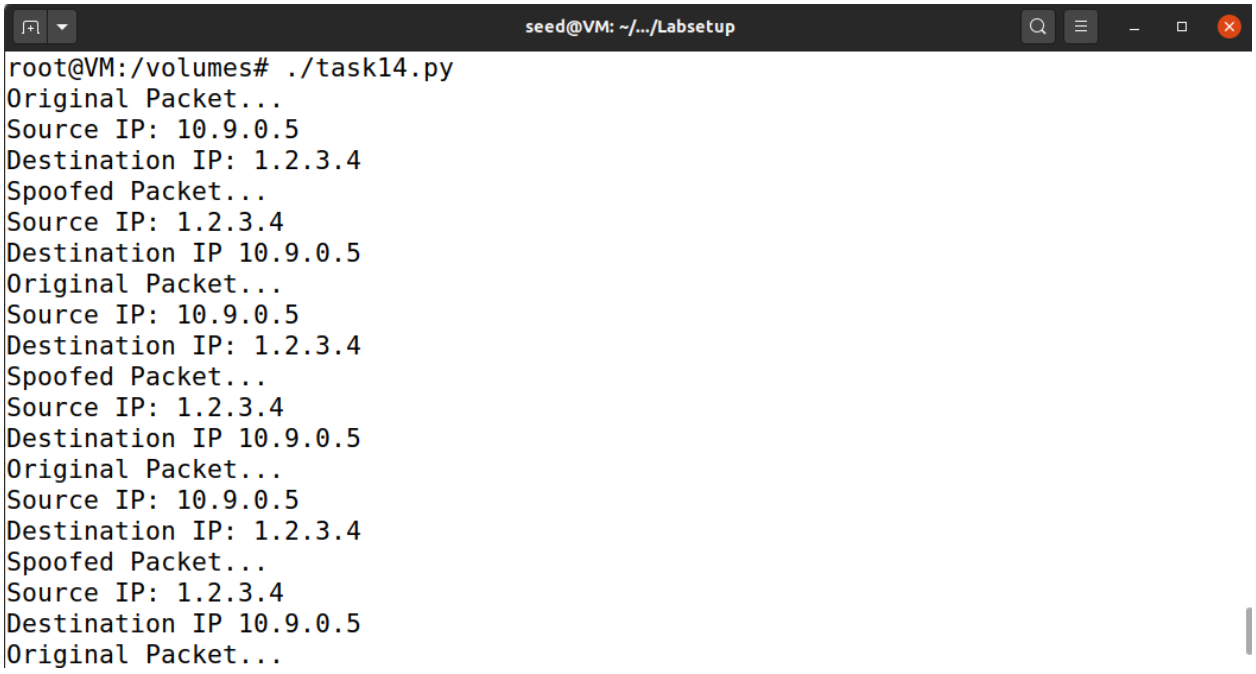
By incrementally adjusting the TTL field in outgoing packets, the traceroute-like program identifies each router in the path to the destination. Observations of ICMP Time Exceeded messages in Wireshark confirm the discovery of intermediary router IPs.

Task 1.4: Sniffing and-then Spoofing

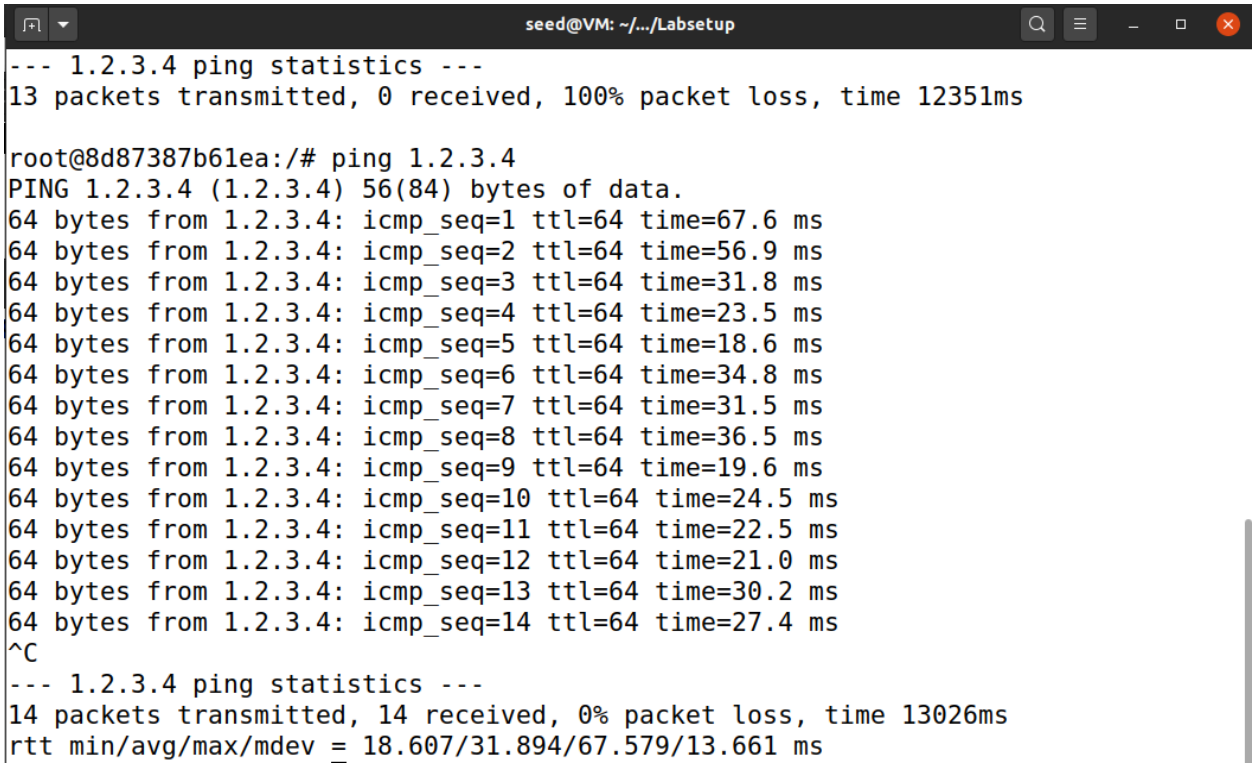
In this task, we need to create a program using Scapy that can sniff and spoof network traffic. The program will watch for ICMP Echo Requests (ping requests which checks if the packet is an ICMP Echo Request has type == 8) on the network and send fake replies, even if the target IP isn't active. It will run on the VM, listen for ping requests and automatically send fake ICMP Echo Replies to the sender, pretending to be the target IP.

1. ping 1.2.3.4 # a non-existing host on the Internet





To generate the results above, on Host A we ping 1.2.3.4:

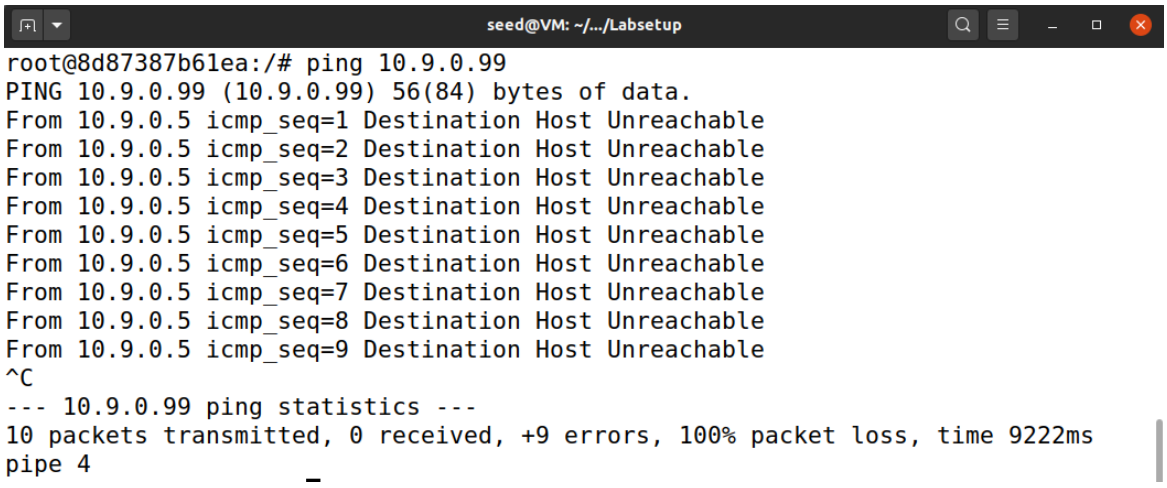


When we ping 1.2.3.4, our program catches the ping request and sends a fake reply. Normally, the ping would fall because 1.2.3.4 isn't reachable. But with our program, it always sends a fake reply, making it seem like the destination is responding.

2. ping 10.9.0.99 # a non-existing host on the LAN

Running the file with the updated filter:

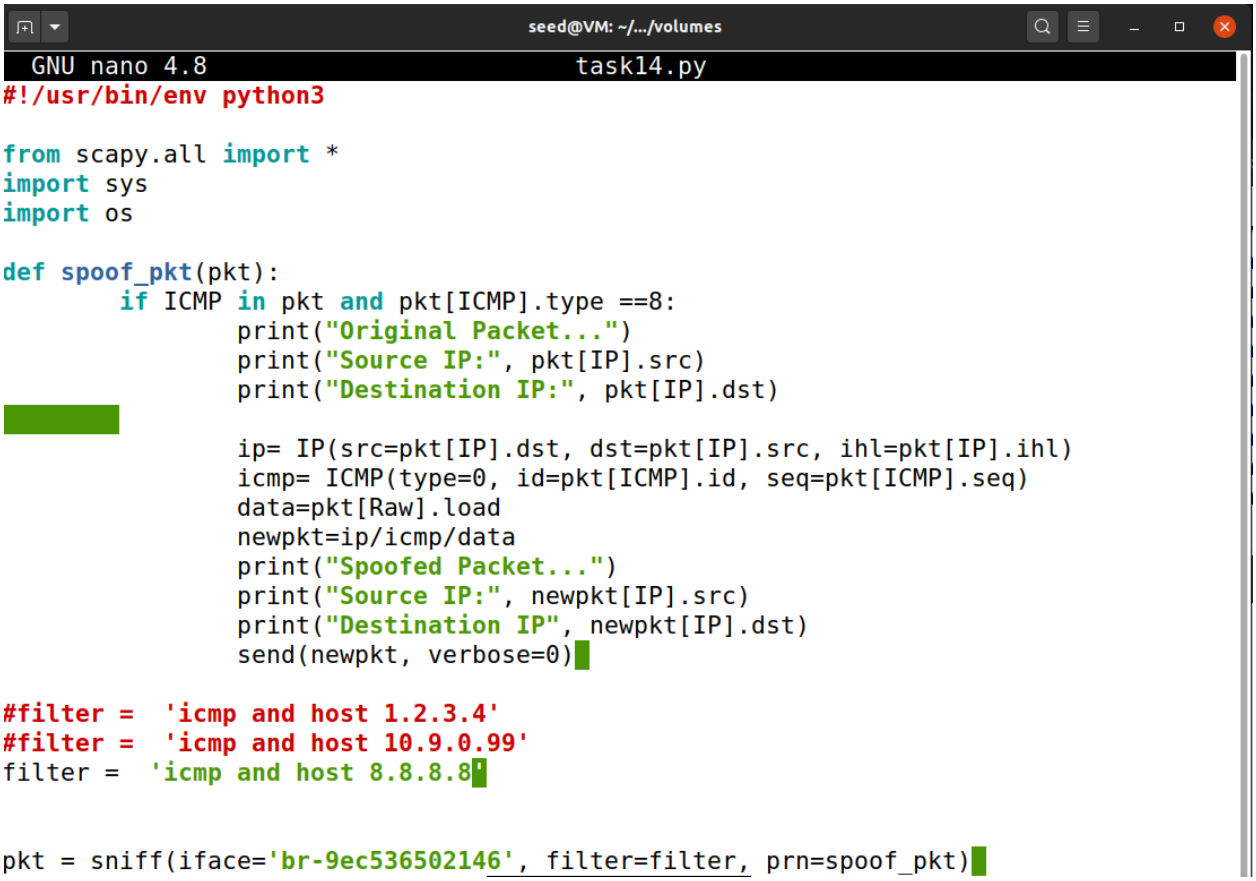
On host A we ping 10.9.0.99 to which there is no output above:

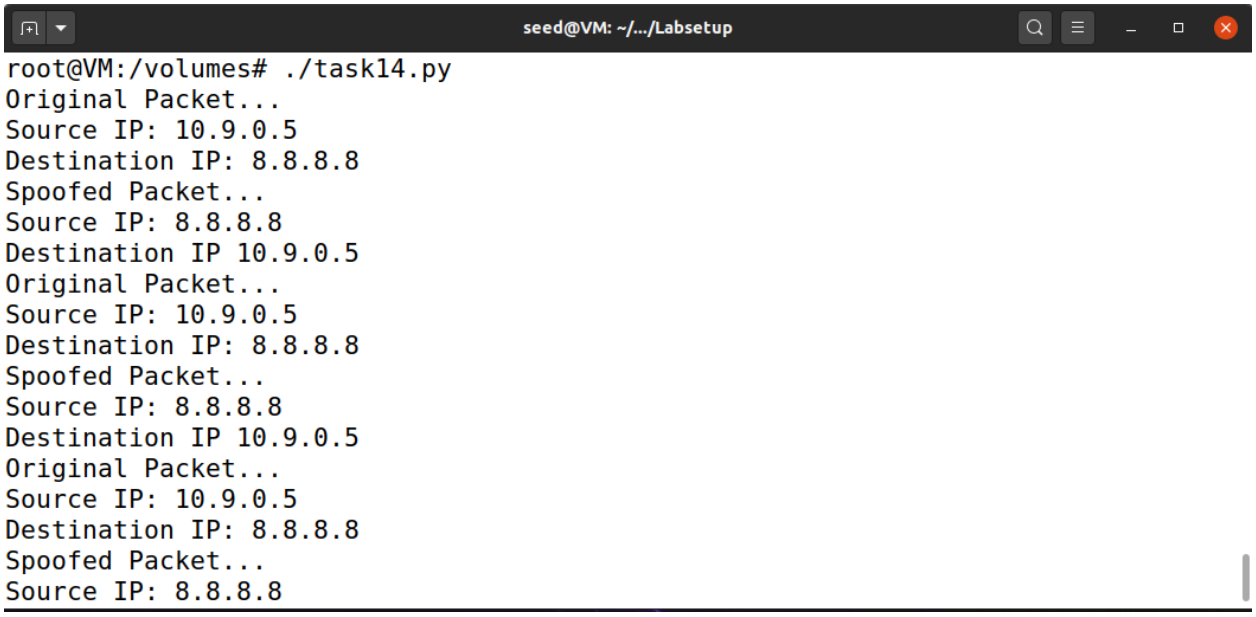


The lack of output indicates that no ICMP Echo Request packets were captured by the sniffing program. This result aligns with how ARP and ICMP interact in network communication:

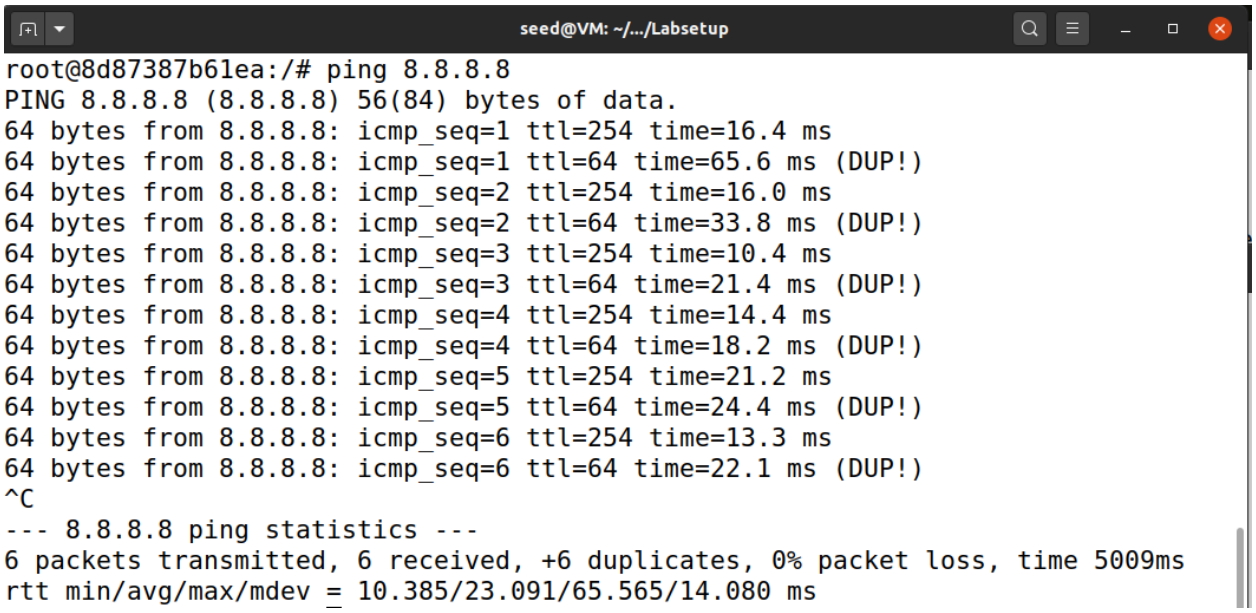
- When Host A pings a non-existing host 10.9.0.99, it first sends an ARP request to resolve the MAC address associated with 10.9.0.99.
- Since the target host does not exist, no ARP reply is received, and thus the ICMP Echo Request packet is never sent.
- Without the ARP resolution, the ICMP packet never gets generated, so the program has nothing to intercept and spoof.

3. ping 8.8.8.8 # an existing host on the Internet





When with ping 8.8.8.8 on host A:



Duplicate Echo Replies are visible, showing both legitimate and spoofed responses.

Here, both the real reply from 8.8.8.8 and the fake reply from our program are received. Normally, the ARP protocol routes packets by linking IP addresses to MAC addresses. But our program skips this by directly creating and sending fake replies with the chosen source and

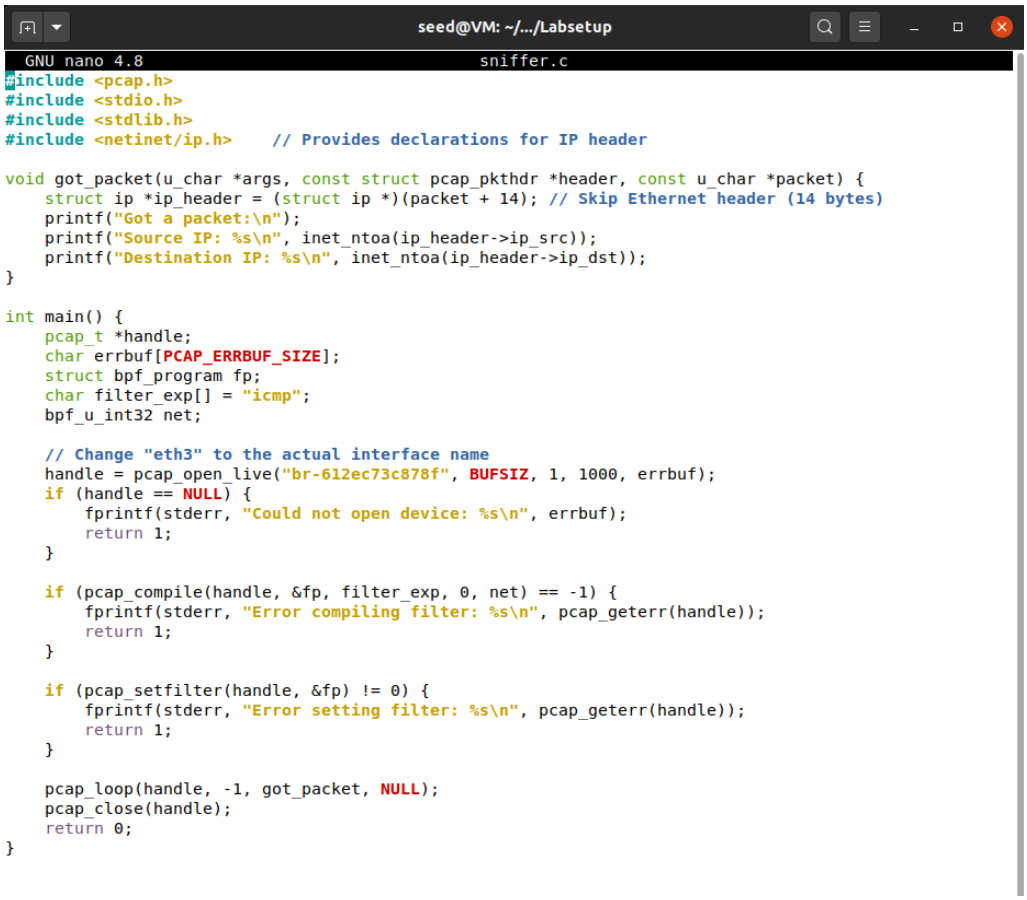
destination IP. This makes the ping tool in the user container see the fake reply almost instantly, even before the real one making it seem like the target is always responding.

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

Task 2.1A: Writing Packet Sniffing Program

We developed a sniffer program using the pcap library (application programming interface (API) for capturing network traffic) to capture network traffic and display the source and destination IP addresses. The program employs BPF (Berkeley Packet Filter) syntax to filter for ICMP packets specifically. When a packet is captured, the program checks whether the header is of the IPv4 type. If this condition is met, it prints the source and destination addresses from the IP header.

Code:

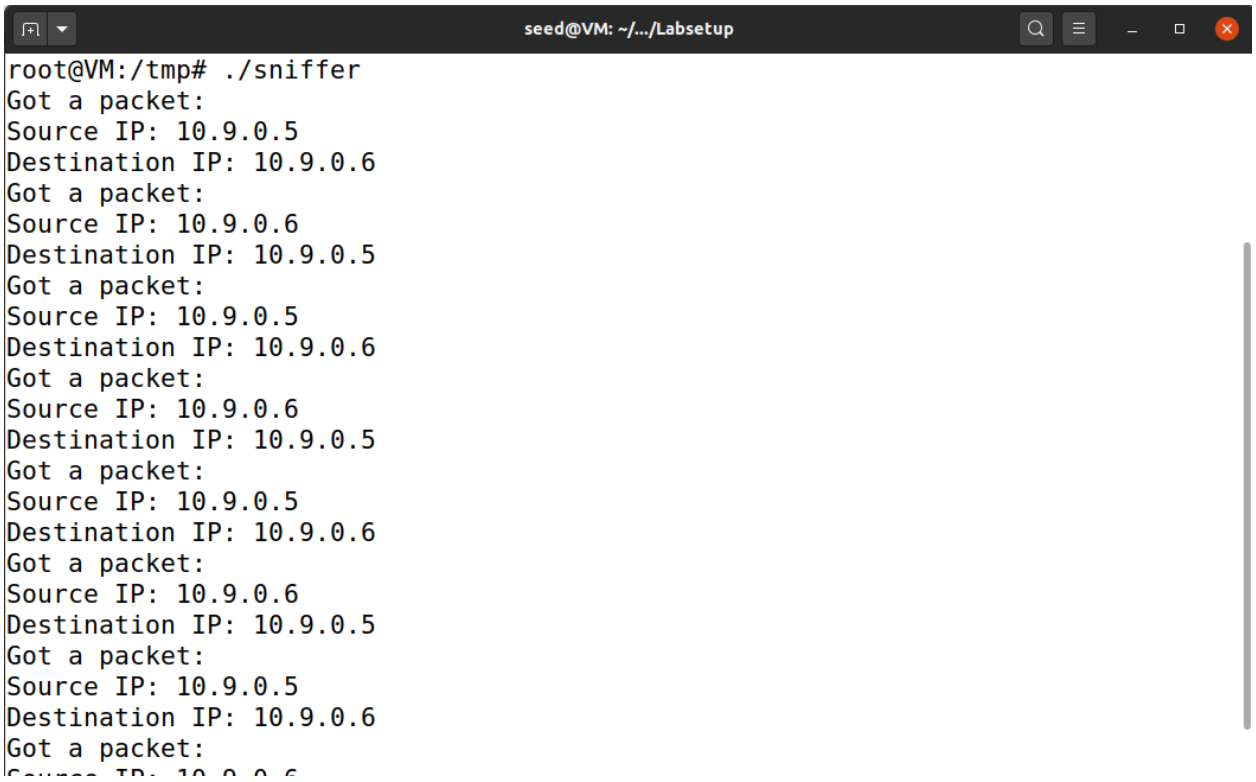


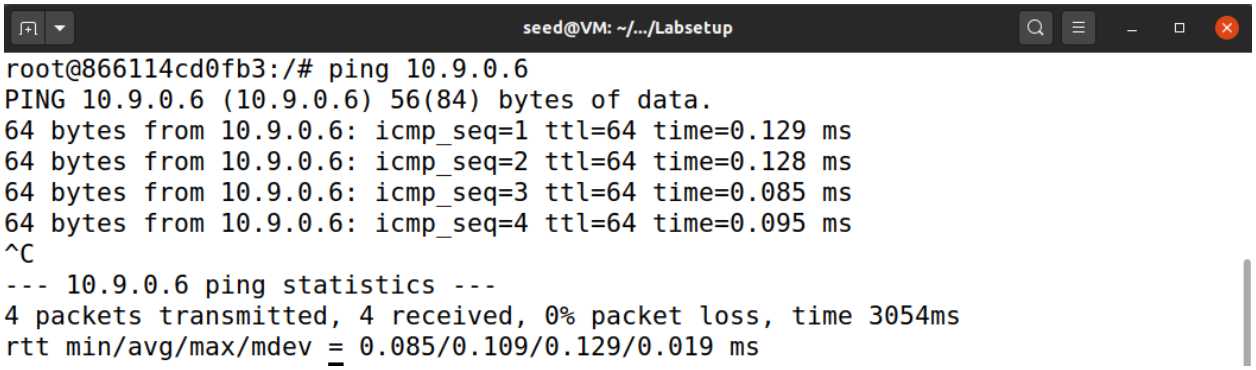
Here we ran the command *ip -d link show dev br-f2478ef59744* to check whether we are running with the interface's promiscuous mode on or off.

```
[11/17/24]seed@VM:~/.../Labsetup$ ip -d link show dev br-612ec73c878f
9: br-612ec73c878f: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT gro
up default
    link/ether 02:42:c4:03:97:9c brd ff:ff:ff:ff:ff:ff promiscuity 1 minmtu 68 maxmtu 65535
    bridge forward_delay 1500 hello_time 200 max_age 2000 ageing_time 30000 stp_state 0 priority 32768
 vlan_filtering 0 vlan_protocol 802.1Q bridge_id 8000.2:42:c4:3:97:9c designated_root 8000.2:42:c4:3:9
7:9c root_port 0 root_path_cost 0 topology_change 0 topology_change_detected 0 hello_timer    0.00 tcn
_timer    0.00 topology_change_timer    0.00 gc_timer    0.00 vlan_default_pvid 1 vlan_stats_enabled 0
 vlan_stats_per_port 0 group_fwd_mask 0 group_address 01:80:c2:00:00:00 mcast_snooping 1 mcast_router
1 mcast_query_use_ifaddr 0 mcast_querier 0 mcast_hash_elasticity 16 mcast_hash_max 4096 mcast_last_mem
ber_count 2 mcast_startup_query_count 2 mcast_last_member_interval 100 mcast_membership_interval 26000
 mcast_querier_interval 25500 mcast_query_interval 12500 mcast_query_response_interval 1000 mcast_star
tup_query_interval 3124 mcast_stats_enabled 0 mcast_igmp_version 2 mcast_mld_version 1 nf_call_iptable
s 0 nf_call_ip6tables 0 nf_call_arptables 0 addrgenmode eui64 numtxqueues 1 numrxqueues 1 gso_max_size
 65536 gso_max_segs 65535
```

Result:

We tested the program by sending a ping to an IP address, and it successfully sniffed the packet, displaying the correct source and destination IPs.





Question 1

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

To create a simple packet sniffer, two libraries are utilized:

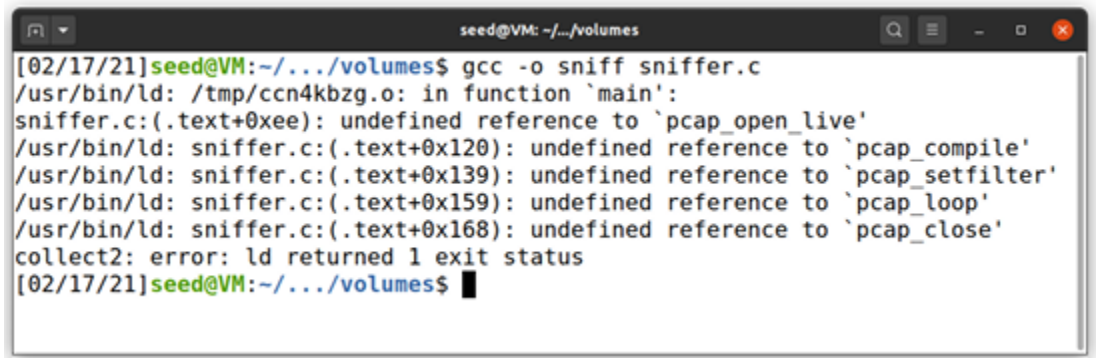
- stdio.h: This is the standard library for input and output operations in C.
- pcap.h: This library is specifically designed for packet capture.

The pcap library facilitates the handling of sockets required for network communication. It simplifies the process of sniffing packets, eliminating the need to manually code packet structures. By using pcap, we can open a raw socket in promiscuous mode, allowing it to capture all traffic on the network. The raw socket is opened with the pcap_open_live() function. Additionally, we can apply filters to focus on specific types of traffic using the pcap_compile() and pcap_setfilter() functions. Packet capture is performed using the pcap_loop() function.

Question 2

Why do you need the root privilege to run a sniffer program?

The sniffer program must operate the network interface card in promiscuous mode, a capability that is restricted to the superuser (root). Attempting to run the program without root privileges results in an error, typically indicating a lack of access rights. Specifically, the program will encounter a failure when invoking the pcap_open_live function to set up a socket with the NIC enp0s3 in promiscuous mode, as this operation is not permitted for standard user programs.



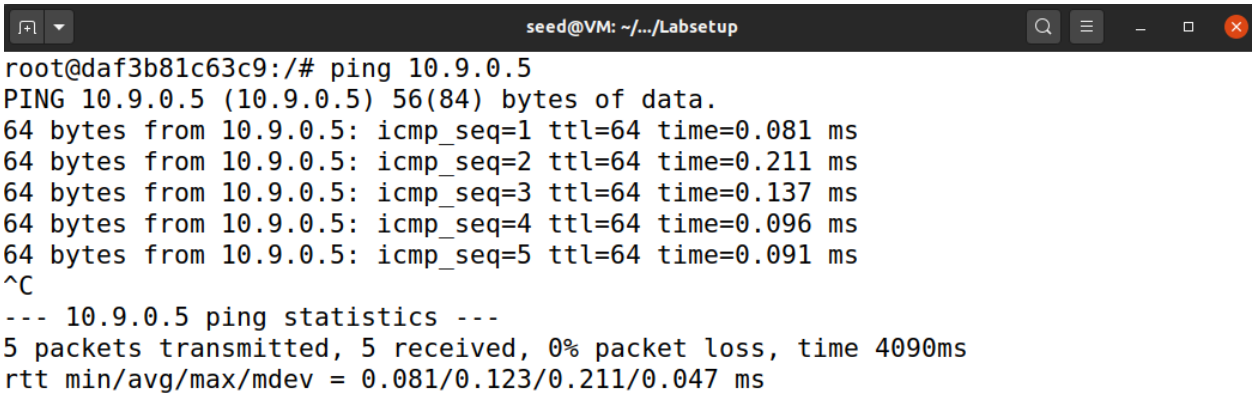
Question 3.

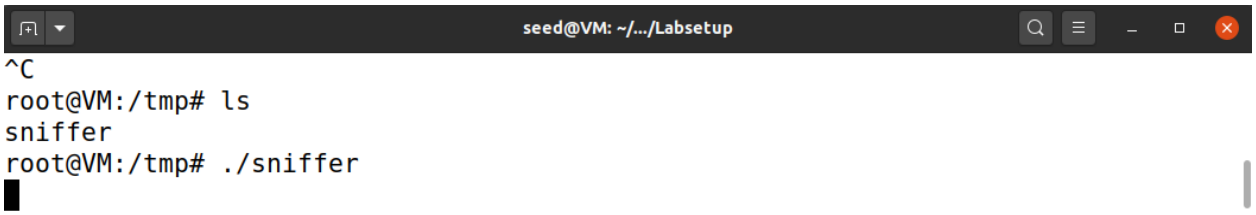
Please turn on and turn off the promiscuous mode in your sniffer program. The value 1 of the third parameter in pcap open live() turns on the promiscuous mode (use 0 to turn it off). Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this. You can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

In Task 2.1A, we enabled promiscuous mode by setting the third parameter of the pcap_open_live function to 1. This allowed us to capture network traffic and observe packets transmitted by other users. Now, we will change the value of the third parameter to 0 to disable promiscuous mode and repeat the same activity as before.

```
[11/17/24]seed@VM:~/.../Labsetup$ gcc -o sniffer sniffer.c -lpcap
[11/17/24]seed@VM:~/.../Labsetup$ docker cp sniffer 51238ec2ba02:/tmp
[11/17/24]seed@VM:~/.../Labsetup$ ip -d link show dev br-612ec73c878f
9: br-612ec73c878f: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT gro
up default
    link/ether 02:42:c4:03:97:9c brd ff:ff:ff:ff:ff:ff promiscuity 0 minmtu 68 maxmtu 65535
    bridge forward_delay 1500 hello_time 200 max_age 2000 ageing_time 30000 stp_state 0 priority 32768
 vlan_filtering 0 vlan_protocol 802.1Q bridge_id 8000.2:42:c4:3:97:9c designated_root 8000.2:42:c4:3:9
7:9c root_port 0 root_path_cost 0 topology_change 0 topology_change_detected 0 hello_timer    0.00 tcn
_timer    0.00 topology_change_timer    0.00 gc_timer  180.20 vlan_default_pvid 1 vlan_stats_enabled 0
 vlan_stats_per_port 0 group_fwd_mask 0 group_address 01:80:c2:00:00:00 mcast_snooping 1 mcast_router
1 mcast_query_use_ifaddr 0 mcast_querier 0 mcast_hash_elasticity 16 mcast_hash_max 4096 mcast_last_mem
ber_count 2 mcast_startup_query_count 2 mcast_last_member_interval 100 mcast_membership_interval 26000
 mcast_querier_interval 25500 mcast_query_interval 12500 mcast_query_response_interval 1000 mcast_star
tup_query_interval 3124 mcast_stats_enabled 0 mcast_igmp_version 2 mcast_mld_version 1 nf_call_iptable
s 0 nf_call_ip6tables 0 nf_call_arptables 0 addrgenmode eui64 numtxqueues 1 numrxqueues 1 gso_max_size
 65536 gso_max_segs 65535
```


```
root@daf3b81c63c9:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.081 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.211 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.137 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.096 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=64 time=0.091 ms
^C
--- 10.9.0.5 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4090ms
rtt min/avg/max/mdev = 0.081/0.123/0.211/0.047 ms
```





We observe that we can no longer capture packets. This occurs because, in the current configuration, we can only sniff packets that are intended for the attacker VM and not for any other host. For instance, when we ping from HostA to HostB, the sniffer in the seed-attacker container will not capture those packets since they are not directed to the attacker's network interface.

On the other hand, when promiscuous mode is enabled, the sniffer captures all traffic on the network, allowing us to see every packet that our device encounters, regardless of its intended recipient. In this mode, the sniffer can capture ICMP packets exchanged between HostA and HostB, as well as other traffic on the same network segment. For example, when we ping from HostA to HostB, the sniffer will display these ICMP packets even though they are not addressed to the seed-attacker container.

Task 2.1B: Writing Filters

Capture the ICMP packets between two specific hosts.

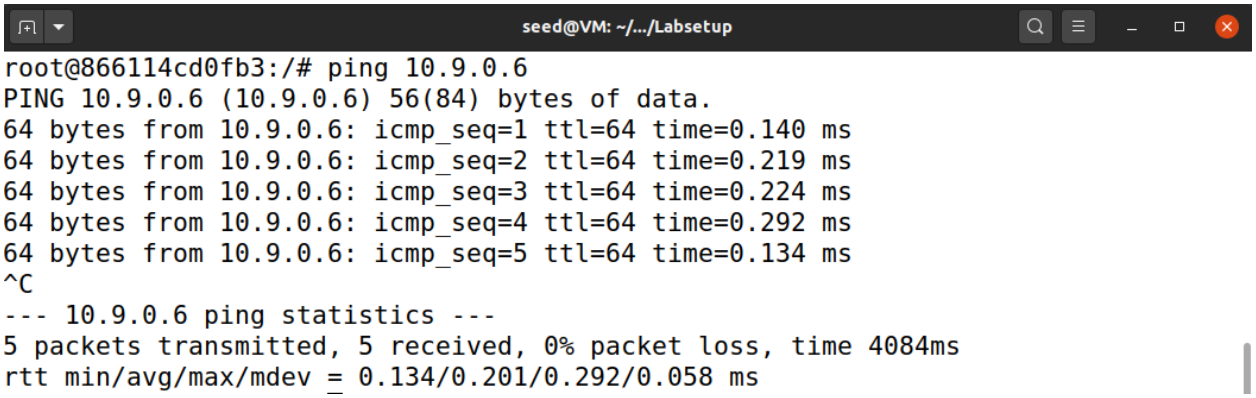
We changed the pcap filter is now: "icmp and src host 10.9.0.5 and host 10.9.0.6"

Code:

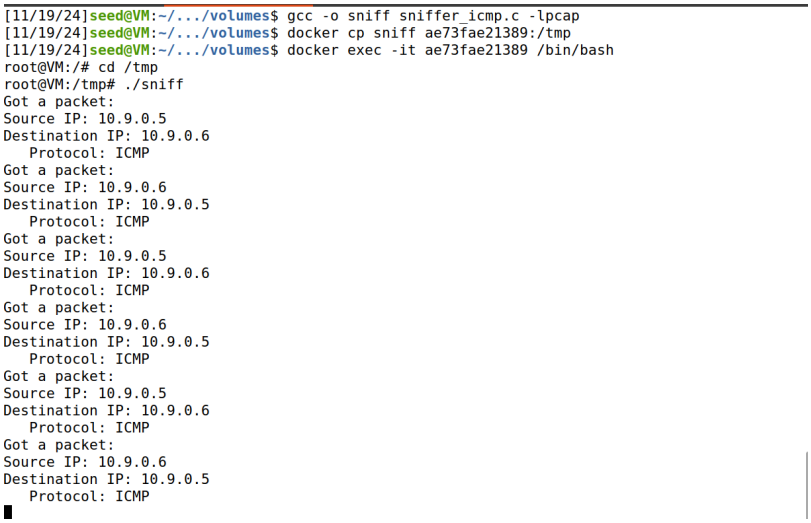
spoofUDP.c | sniff-snoof.c | *snifffer_icmp.c

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

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    struct ip *ip_header = (struct ip *)(packet + 14);

    printf("Got a packet:\n");
    printf("Source IP: %s\n", inet_ntoa(ip_header->ip_src));
    printf("Destination IP: %s\n", inet_ntoa(ip_header->ip_dst));

    switch(ip_header->ip_p) {
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            break;
        default:
            printf("   Protocol: Others\n");
            break;
    }
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;

    // Filter expression to capture ICMP packets between two specific hosts
    char filter_exp[] = "icmp and host 10.9.0.5 and host 10.9.0.6";

    bpf_u_int32 net;

    handle = pcap_open_live("br-b985f1959048", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Could not open device: %s\n", errbuf);
        return 1;
    }

    // Compile the filter expression into BPF pseudo-code
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Error compiling filter: %s\n", pcap_geterr(handle));
        return 1;
    }

    // Set the filter on the pcap handle
    if (pcap_setfilter(handle, &fp) != 0) {
        fprintf(stderr, "Error setting filter: %s\n", pcap_geterr(handle));
        return 1;
    }

    // Start capturing packets
    pcap_loop(handle, -1, got_packet, NULL);

    // Close the handle when done
    pcap_close(handle);

    return 0;
}
```

Result:



Our sniffer program is correctly capturing and displaying the relevant information for ICMP packets exchanged between the two specified hosts.



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

We changed the pcap filter now "proto TCP and dst portrange 10-100".

Code:

spoofUDP.c | sniff-snoof.c | sniffer_icmp.c | *sniffer_tcp.c

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
    struct ip *ip_header = (struct ip *)(packet + 14);

    printf("Got a packet:\n");
    printf("Source IP: %s\n", inet_ntoa(ip_header->ip_src));
    printf("Destination IP: %s\n", inet_ntoa(ip_header->ip_dst));

    switch(ip_header->ip_p) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            struct tcphdr *tcp_header = (struct tcphdr *)(packet + 14 + ip_header->ip_hl * 4); // Skip IP header
            printf("   Source Port: %d\n", ntohs(tcp_header->source));      // Source port
            printf("   Destination Port: %d\n", ntohs(tcp_header->dest));  // Destination port
            break;
        default:
            printf("   Protocol: Others\n");
            break;
    }
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;

    // Filter expression to capture TCP packets in a specific port range
    char filter_exp[] = "tcp and dst portrange 10-100";

    bpf_u_int32 net;

    handle = pcap_open_live("br-b985f1959048", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Could not open device: %s\n", errbuf);
        return 1;
    }

    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Error compiling filter: %s\n", pcap_geterr(handle));
        return 1;
    }

    // Set the filter on the pcap handle
    if (pcap_setfilter(handle, &fp) != 0) {
        fprintf(stderr, "Error setting filter: %s\n", pcap_geterr(handle));
        return 1;
    }

    // Start capturing packets
    pcap_loop(handle, -1, got_packet, NULL);

    // Close the handle when done
    pcap_close(handle);

    return 0;
}
```

Result:

This output verifies that the sniffer program is effectively capturing and displaying pertinent information about TCP packets exchanged between hosts.

We utilized Telnet to capture the TCP packet and transmit it to another VM, specifically HostB. The program successfully captures the packet and displays its contents.

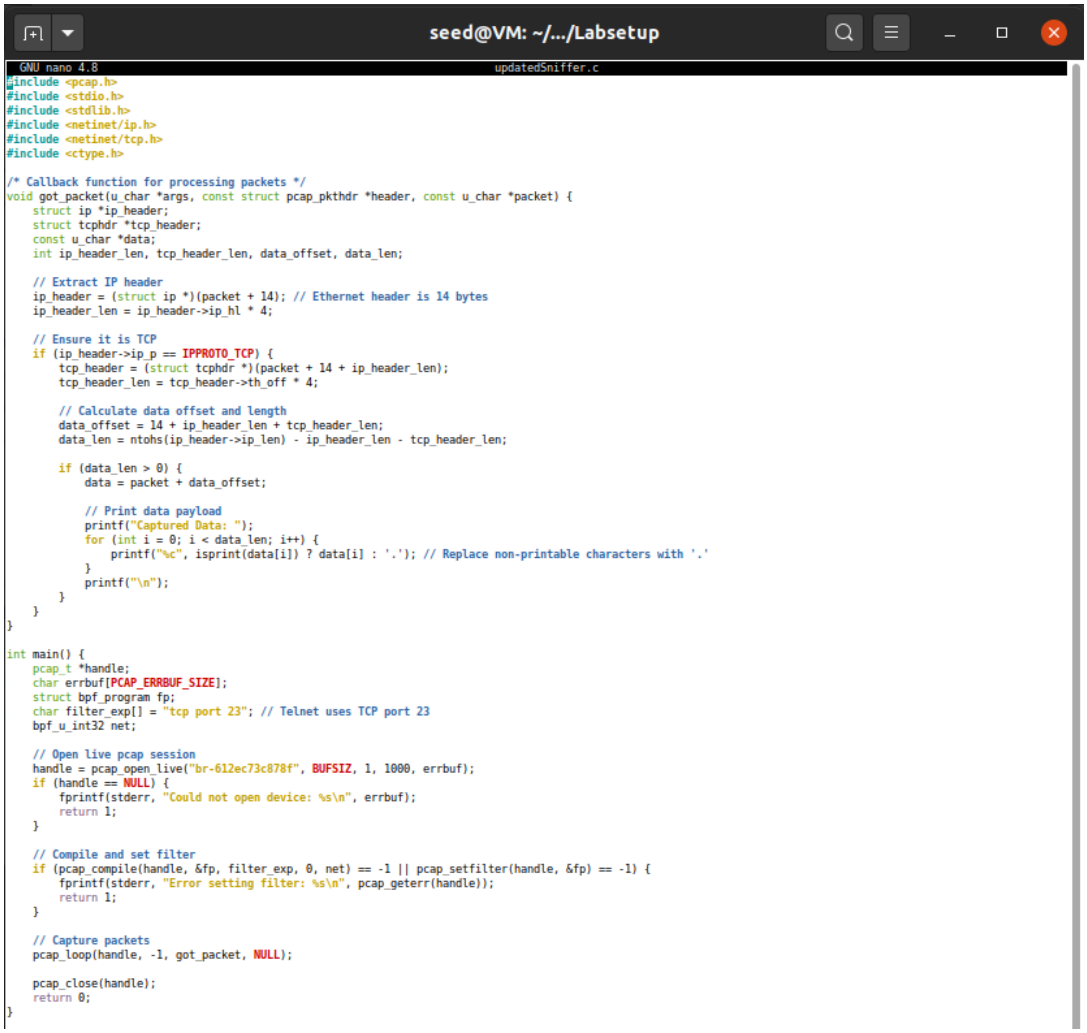
```
[11/19/24]seed@VM:~/.../volumes$ gcc -o sniff sniffer_tcp.c -lpcap
[11/19/24]seed@VM:~/.../volumes$ docker cp sniff ae73fae21389:/tmp
[11/19/24]seed@VM:~/.../volumes$ docker exec -it ae73fae21389 /bin/bash
root@VM:/# cd /tmp
root@VM:/tmp# ./sniff
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
   Protocol: TCP
   Source Port: 50304
   Destination Port: 23
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
   Protocol: TCP
   Source Port: 50304
   Destination Port: 23
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
   Protocol: TCP
   Source Port: 50304
   Destination Port: 23
Got a packet:
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
   Protocol: TCP
   Source Port: 50304
   Destination Port: 23
Got a packet:
```

```
root@b84d8785a75e:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
df26c92c6a68 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: Mon Nov 18 04:33:18 UTC 2024 from hostA-10.9.0.5.net-10.9.0.0 on pts/2
seed@df26c92c6a68:~$
```

Task 2.1C: Sniffing Passwords

We changed the pcap filter to "tcp post 23".

Code:



Result:

```
root@866114cd0fb3:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
daf3b81c63c9 login: roxie
Password:

Login incorrect
daf3b81c63c9 login: ^[[A^[[A^[[A
Password:

Login incorrect
daf3b81c63c9 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.
```

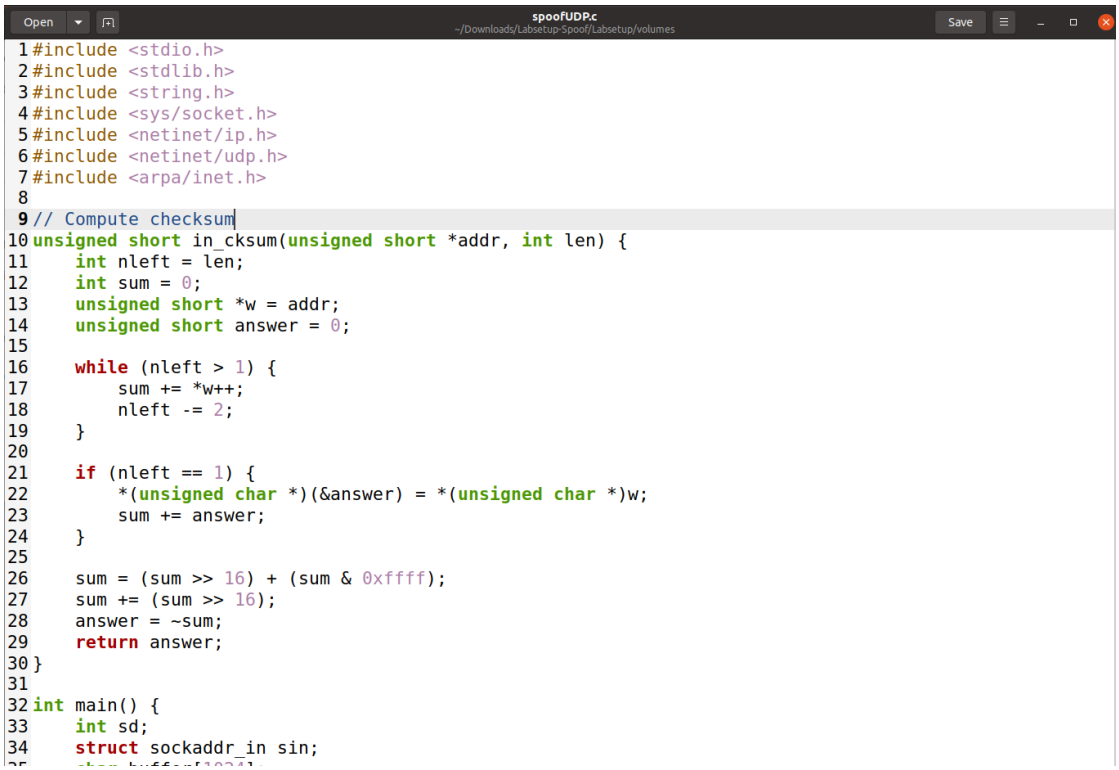
Observation:
In this scenario, VM1 is executing the sniffer program while VM2 establishes a Telnet connection to VM3. The program captures and displays the data being transmitted within these packets. As the sniffer is actively monitoring TCP packets, it successfully captures the Telnet traffic, revealing the payload in clear text. Consequently, whenever any data (such as a username or password) is entered on VM2, it appears on VM1. This occurs because Telnet transmits data in clear text over the network, making it susceptible to packet sniffing.

Task 2.2: Spoofing

Task 2.2A: Write a spoofing program

We developed a spoofing program that transmits a UDP packet to host 10.9.0.6 on port 9090, containing the message “Hello, UDP!”. We then set up a UDP server on 10.9.0.6 that listens on port 2020, after which we execute the program on the attacker machine.

Code:
The program was created using the pcap library and modified the IP headers to designate the source IP as 10.9.0.5 and the destination IP as 10.9.0.6.




```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <arpa/inet.h>

// Compute checksum
unsigned short in_cksum(unsigned short *addr, int len) {
    int nleft = len;
    int sum = 0;
    unsigned short *w = addr;
    unsigned short answer = 0;

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

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

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

int main() {
    int sd;
    struct sockaddr_in sin;
```

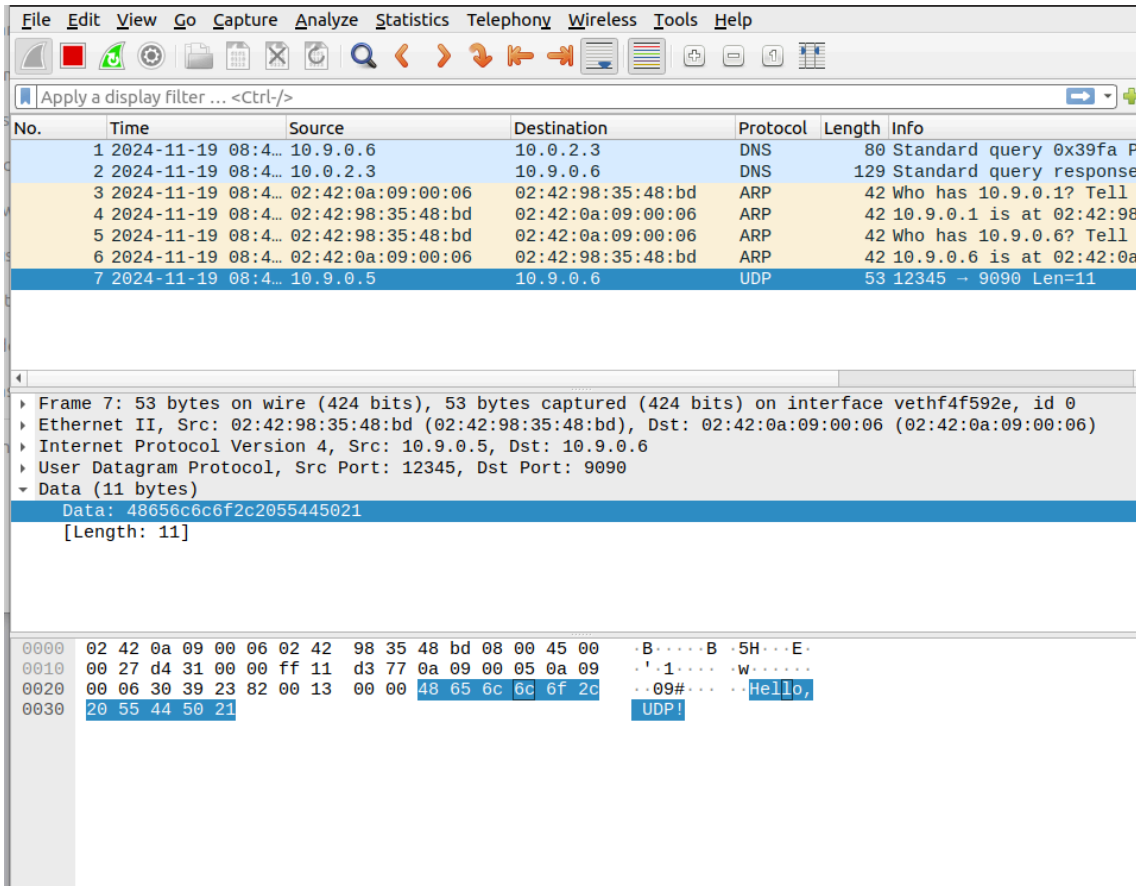
```c
    char buffer[1024];

    // Create a raw socket
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sd < 0) {
        perror("socket() error");
        exit(-1);
    }

    // Set IP_HDRINCL
    int one = 1;
    const int *val = &one;
    if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one)) < 0) {
        perror("setsockopt() error");
        exit(-1);
    }

    // Construct the packet
    struct ip *ip = (struct ip *) buffer;
    struct udphdr *udp = (struct udphdr *) (buffer + sizeof(struct ip));
    char *data = buffer + sizeof(struct ip) + sizeof(struct udphdr);

    // IP header
    ip->ip_v = 4;
    ip->ip_hl = 5;
    ip->ip_tos = 0;
    ip->ip_len = htons(sizeof(struct ip) + sizeof(struct udphdr) + strlen("Hello, UDP!"));
    ip->ip_id = htons(54321);
    ip->ip_off = 0;
    ip->ip_ttl = 255;
    ip->ip_p = IPPROTO_UDP;
    ip->ip_sum = 0;
    ip->ip_src.s_addr = inet_addr("10.9.0.5");  // Spoofed source IP
    ip->ip_dst.s_addr = inet_addr("10.9.0.6");  // Destination IP

    // UDP header
    udp->uh_sport = htons(12345);  // Source port
    udp->uh_dport = htons(9090);   // Destination port
    udp->uh_ulen = htons(sizeof(struct udphdr) + strlen("Hello, UDP!"));
    udp->uh_sum = 0;

    // UDP data
    strcpy(data, "Hello, UDP!");

    // Destination info
    sin.sin_family = AF_INET;
    sin.sin_port = udp->uh_dport;
    sin.sin_addr.s_addr = ip->ip_dst.s_addr;

    // Send the packet
    if (sendto(sd, buffer, ntohs(ip->ip_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        perror("sendto() error");
        exit(-1);
    }

    printf("Spoofed UDP packet sent successfully\n");
    return 0;
}
```

C Tab Width: 8 Ln 34, Col 5 INS

```
root@VM:/volumes# ./spoof
Spoofed UDP packet sent successfully
root@VM:/volumes#
```

```
exit
[11/19/24]seed@VM:~/.../Labsetup$ docksh hostB-10.9.0.6
root@df26c92c6a68:/# nc -luv 9090
Bound on 0.0.0.0 9090
Connection received on hostA-10.9.0.5.net-10.9.0.0 12345
Hello, UDP!
```

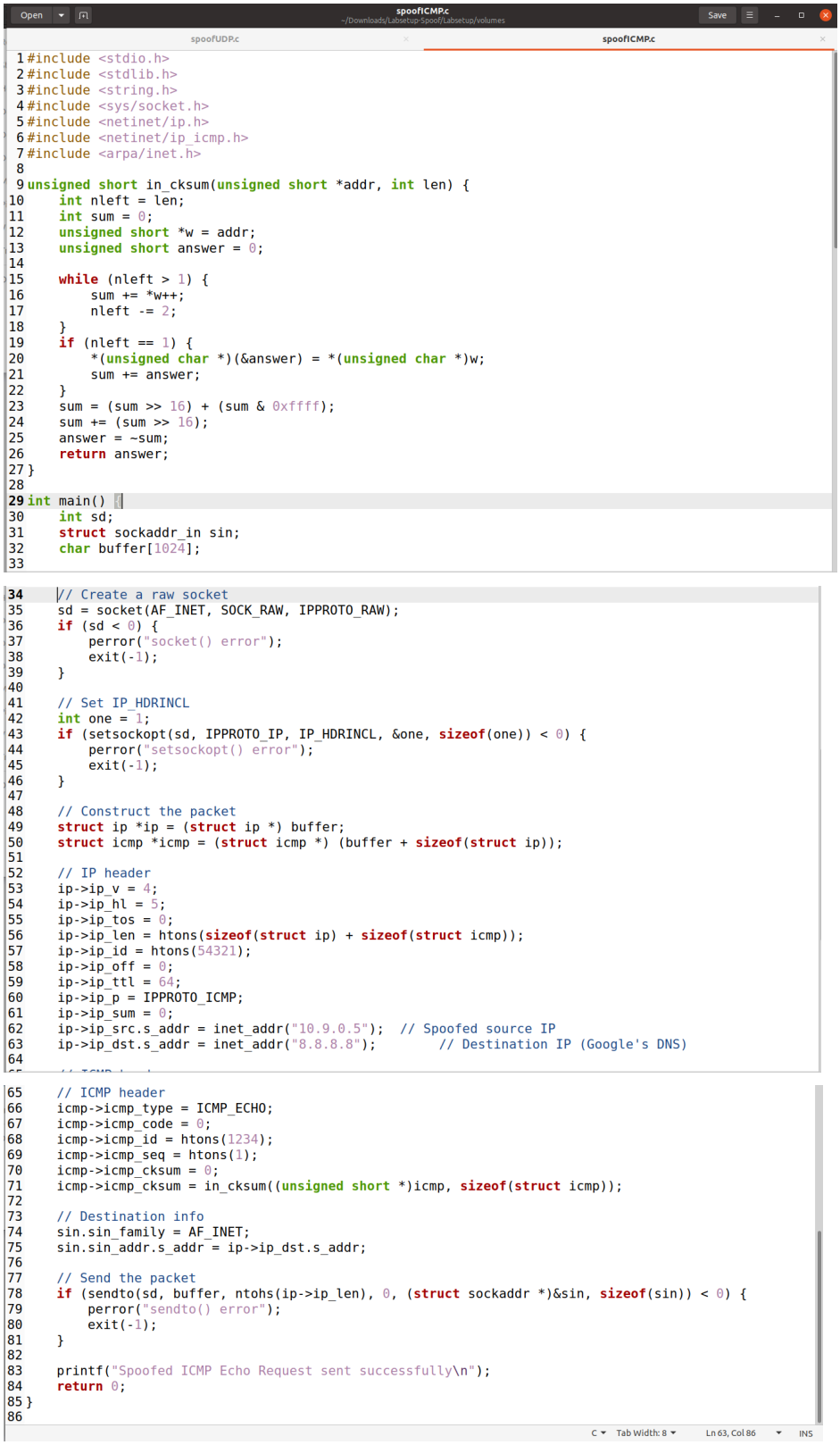
Observation:
We see that as soon as we execute the program, VM2 immediately displays the string "Hello, UDP!". The packet was successfully transmitted using raw sockets, demonstrating our ability to effectively send spoofed UDP packets.



Task 2.2B: Spoof an ICMP Echo Request

We developed a program that sent a fake ping (ICMP Echo Request) to Google's DNS server at 8.8.8.8. This ping pretended to come from the IP address 10.9.0.5, which wasn't its real source. Google's server received this ping and, thinking it was genuine, sent a reply back to 10.9.0.5. This reply made its way back to our network, probably because 10.9.0.5 is an address within our local network range.
This explanation describes how the program successfully spoofed an ICMP packet and received a response, demonstrating the basics of IP spoofing and how network routing handles such packets.

Code:




```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>

unsigned short in_cksum(unsigned short *addr, int len) {
    int nleft = len;
    int sum = 0;
    unsigned short *w = addr;
    unsigned short answer = 0;

    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }
    if (nleft == 1) {
        *(unsigned char *)(&answer) = *(unsigned char *)w;
        sum += answer;
    }
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    answer = ~sum;
    return answer;
}

int main() {
    int sd;
    struct sockaddr_in sin;
    char buffer[1024];

    // Create a raw socket
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sd < 0) {
        perror("socket() error");
        exit(-1);
    }

    // Set IP_HDRINCL
    int one = 1;
    if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &one, sizeof(one)) < 0) {
        perror("setsockopt() error");
        exit(-1);
    }

    // Construct the packet
    struct ip *ip = (struct ip *) buffer;
    struct icmp *icmp = (struct icmp *) (buffer + sizeof(struct ip));

    // IP header
    ip->ip_v = 4;
    ip->ip_hl = 5;
    ip->ip_tos = 0;
    ip->ip_len = htons(sizeof(struct ip) + sizeof(struct icmp));
    ip->ip_id = htons(54321);
    ip->ip_off = 0;
    ip->ip_ttl = 64;
    ip->ip_p = IPPROTO_ICMP;
    ip->ip_sum = 0;
    ip->ip_src.s_addr = inet_addr("10.9.0.5");  // Spoofed source IP
    ip->ip_dst.s_addr = inet_addr("8.8.8.8");        // Destination IP (Google's DNS)

    // ICMP header
    icmp->icmp_type = ICMP_ECHO;
    icmp->icmp_code = 0;
    icmp->icmp_id = htons(1234);
    icmp->icmp_seq = htons(1);
    icmp->icmp_cksum = 0;
    icmp->icmp_cksum = in_cksum((unsigned short *)icmp, sizeof(struct icmp));

    // Destination info
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = ip->ip_dst.s_addr;

    // Send the packet
    if (sendto(sd, buffer, ntohs(ip->ip_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        perror("sendto() error");
        exit(-1);
    }

    printf("Spoofed ICMP Echo Request sent successfully\n");
    return 0;
}
```

We look up Google's DNS here:

```
root@b84d8785a75e:/# nslookup dns.google
Server:         127.0.0.11
Address:        127.0.0.11#53

Non-authoritative answer:
Name:   dns.google
Address: 8.8.4.4
Name:   dns.google
Address: 8.8.8.8
Name:   dns.google
Address: 2001:4860:4860::8888
Name:   dns.google
Address: 2001:4860:4860::8844
```

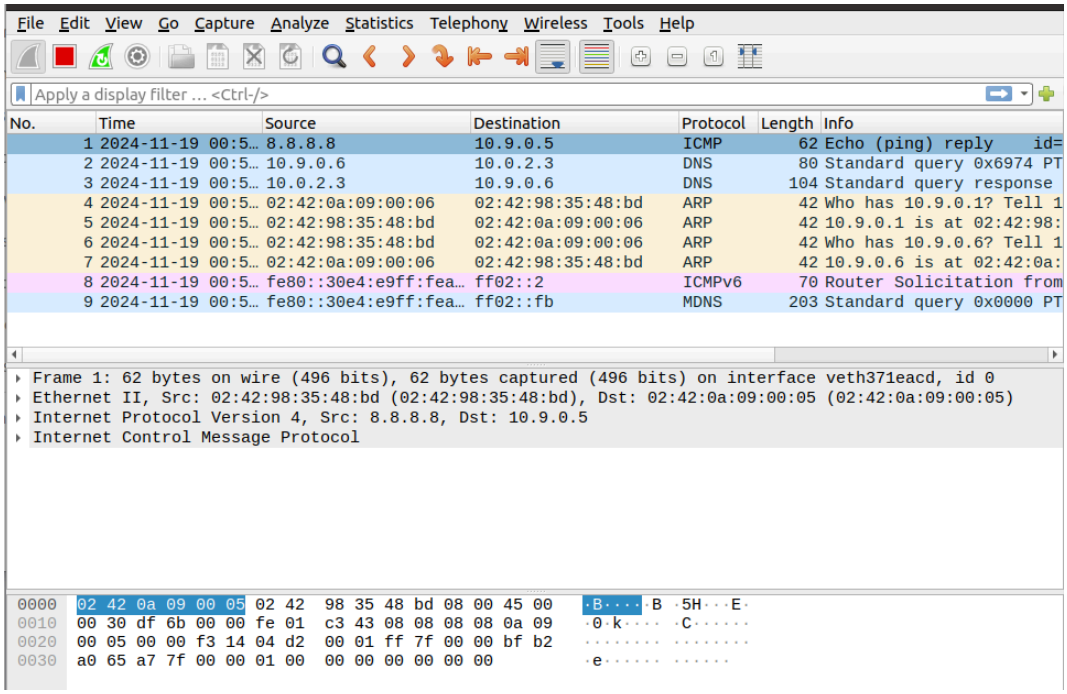
Running the program:

```
root@VM:/volumes# ./spoof_icmp
Spoofed ICMP Echo Request sent successfully
root@VM:/volumes#
```

```
root@df26c92c6a68:/# tcpdump -i eth0 icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
04:39:41.142719 IP hostA-10.9.0.5.net-10.9.0.0 > df26c92c6a68: ICMP echo request, id 1234
, seq 1, length 28
04:39:41.142754 IP df26c92c6a68 > hostA-10.9.0.5.net-10.9.0.0: ICMP echo reply, id 1234,
seq 1, length 28
```

```
root@b84d8785a75e:/# tcpdump -i eth0 icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
04:39:41.142768 IP hostB-10.9.0.6.net-10.9.0.0 > b84d8785a75e: ICMP echo reply, id 1234,
seq 1, length 28
```

Observation:
This packet is the ICMP Echo Reply sent by Google's DNS server (8.8.8.8) in response to our spoofed ICMP Echo Request. This means the spoofing was successful.



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

The IP packet length field can be set to any value greater than or equal to 20. Values below 20 cause the sendto() function to throw an error due to invalid arguments, as 20 bytes is the minimum size for an IP packet header without payload. While values above 20 allow the packet to be sent, the system overwrites the total length to the actual packet size during transmission.

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

Regarding checksums for the IP header in raw socket programming, manual calculation is unnecessary. By default, setting ip_check = 0 allows the kernel to handle checksum calculation. However, if a different value is set, a custom checksum method must be implemented.

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?

Root privileges are required for programs using raw sockets because they need to set the Network Interface Card (NIC) to promiscuous mode. This mode allows capture of all network packets. Without root privileges, the program fails at the socket creation stage, producing an error indicating that the operation is not permitted. This occurs because non-privileged users cannot enable promiscuous mode on the NIC.

Task 2.3: Sniff and then Spoof

Here we sniff ICMP packets on the network and spoof back packets using the spoofing program.

spoofUDP.c | sniff-snoof.c

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PACKET_LEN 1500

// Compute checksum function
unsigned short in_cksum(unsigned short *addr, int len) {
    int nleft = len;
    int sum = 0;
    unsigned short *w = addr;
    unsigned short answer = 0;

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

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

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

// Send spoofed ICMP echo reply
void send_echo_reply(struct ip *ip_header, struct icmp *icmp_header) {
    char buffer[PACKET_LEN];
    struct ip *reply_ip = (struct ip *)buffer;
    struct icmp *reply_icmp = (struct icmp *)(buffer + sizeof(struct ip));
    struct sockaddr_in dest_info;
    int enable = 1;

    // Construct IP header
    memset(buffer, 0, PACKET_LEN);
    reply_ip->ip_v = 4;
    reply_ip->ip_hl = 5;
    reply_ip->ip_tos = 0;
    reply_ip->ip_len = htons(sizeof(struct ip) + sizeof(struct icmp));
    reply_ip->ip_id = htons(54321);
    reply_ip->ip_off = 0;
    reply_ip->ip_ttl = 64;
    reply_ip->ip_p = IPPROTO_ICMP;
    reply_ip->ip_sum = 0;
    reply_ip->ip_src = ip_header->ip_dst;
    reply_ip->ip_dst = ip_header->ip_src;

    // Construct ICMP header
    reply_icmp->icmp_type = ICMP_ECHOREPLY;
    reply_icmp->icmp_code = 0;
    reply_icmp->icmp_id = icmp_header->icmp_id;
    reply_icmp->icmp_seq = icmp_header->icmp_seq;
    reply_icmp->icmp_cksum = 0;
    reply_icmp->icmp_cksum = in_cksum((unsigned short *)reply_icmp, sizeof(struct icmp));

```

```
    // Create raw socket
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sock < 0) {
        perror("socket error");
        exit(1);
    }

    // Set socket option
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

    // Set destination info
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = reply_ip->ip_dst;

    // Send the packet
    sendto(sock, buffer, ntohs(reply_ip->ip_len), 0, (struct sockaddr *)&dest_info,
  sizeof(dest_info));

    close(sock);
    printf("Spoofed ICMP Echo Reply sent to %s\n", inet_ntoa(reply_ip->ip_dst));
}

// Packet processing callback function
void packet_handler(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char *packet) {
    struct ip *ip_header = (struct ip *)(packet + 14); // Skip Ethernet header
    struct icmp *icmp_header = (struct icmp *)(packet + 14 + ip_header->ip_hl * 4);

    if (ip_header->ip_p == IPPROTO_ICMP && icmp_header->icmp_type == ICMP_ECHO) {
        printf("ICMP Echo Request detected from %s\n", inet_ntoa(ip_header->ip_src));
        send_echo_reply(ip_header, icmp_header);
    }
}

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

    // Open the network interface for packet capture
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device: %s\n", errbuf);
        return 2;
    }

    // Compile and set the filter
    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter %s: %s\n", filter_exp, pcap_geterr(handle));
        return 2;
    }
    if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter %s: %s\n", filter_exp, pcap_geterr(handle));
        return 2;
    }

    printf("Sniffing for ICMP Echo Requests...\n");
    pcap_loop(handle, -1, packet_handler, NULL);

    pcap_close(handle);
    return 0;
}
```

C Tab Width: 8 Ln 64, Col 90 INS

We run the program on attacker machine and start the Wireshark to see the packet transmission:

```
root@VM:/volumes# ./sniff-snoof
Sniffing for ICMP Echo Requests...
ICMP Echo Request detected from 10.0.2.15
Spoofed ICMP Echo Reply sent to 10.0.2.15
ICMP Echo Request detected from 10.0.2.15
Spoofed ICMP Echo Reply sent to 10.0.2.15
ICMP Echo Request detected from 10.0.2.15
Spoofed ICMP Echo Reply sent to 10.0.2.15
```

We start a ping from HostA to the IP address 10.0.9.6:

```
root@b84d8785a75e:/# ping 10.0.9.6
PING 10.0.9.6 (10.0.9.6) 56(84) bytes of data.
^C
--- 10.0.9.6 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 1999ms

root@b84d8785a75e:/#
```

As soon as the ping command is initiated, our program immediately detects the echo request and generates a spoofed echo response. Since the target host is operational, Wireshark displays duplicate echo replies.

Our attacker machine was set to promiscuous mode, allowing its network interface card (NIC) to capture all incoming packets. When we ran our spoofing program, it processed these captured packets by swapping the source and destination addresses. The program then constructs a new packet with these modified details and transmits it. The victim machine receives this spoofed packet, believing it to be a legitimate ICMP echo reply.

This process demonstrates our successful spoofing of the ICMP echo request, effectively intercepting and manipulating network traffic.